

Chapter 1. Introducing Android

This chapter covers

- Exploring Android, the open source phone and tablet platform
- Android `Intents`, the way things work
- Sample application

You've heard about Android. You've read about Android. Now it's time to begin unlocking Android.

Android is a software platform that's revolutionizing the global cell phone market. It's the first open source mobile application platform that's moved the needle in major mobile markets around the globe. When you're examining Android, there are a number of technical and market-related dimensions to consider. This first section introduces the platform and provides context to help you better understand Android and where it fits in the global cell phone scene. Moreover, Android has eclipsed the cell phone market, and with the release of Android 3.X has begun making inroads into the tablet market as well. This book focuses on using SDKs from 2.0 to 3.X.

Android is primarily a Google effort, in collaboration with the Open Handset Alliance. Open Handset Alliance is an alliance of dozens of organizations committed to bringing a "better" and more "open" mobile phone to market. Considered a novelty at first by some, Android has grown to become a market-changing player in a few short years, earning both respect and derision alike from peers in the industry.

This chapter introduces Android—what it is, and, equally important, what it's not. After reading this chapter, you'll understand how Android is constructed, how it compares with other offerings in the market, and what its foundational technologies are, plus you'll get a preview of Android application architecture. More specifically, this chapter takes a look at the Android platform and its relationship to the popular Linux operating system, the Java programming language, and the runtime environment known as the Dalvik virtual machine (VM).

Java programming skills are helpful throughout the book, but this chapter is more about setting the stage than about coding specifics. One coding element introduced in this chapter is the `Intent` class. Having a good understanding of and comfort level with the `Intent` class is essential for working with the Android platform.

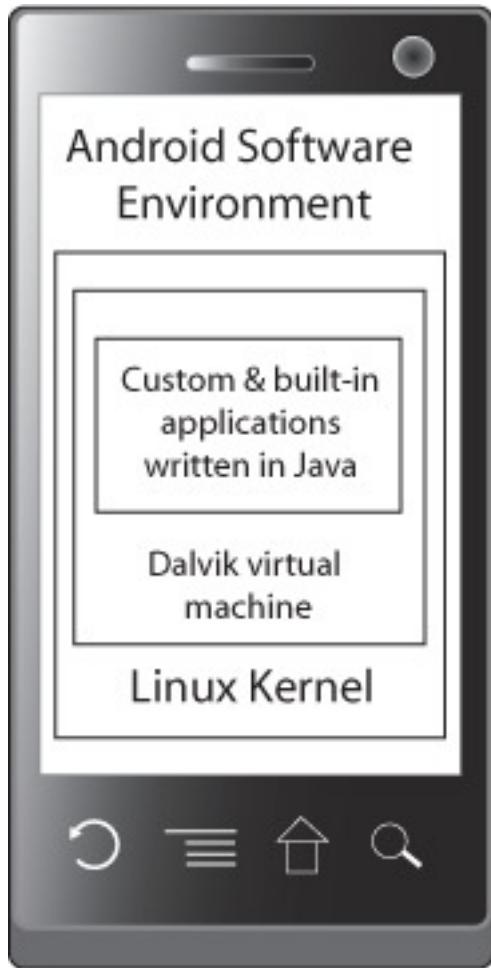
In addition to `Intent`, this chapter introduces the four main application components: `Activity`, `Service`, `ContentProvider`, and `BroadcastReceiver`. The chapter concludes with a simple Android application to get you started quickly.

1.1. THE ANDROID PLATFORM

Android is a software environment built for mobile devices. It's not a hardware platform. Android includes a Linux kernel-based OS, a rich UI, end-user applications, code libraries, application frameworks, multimedia support, and much more. And, yes, even telephone functionality is included! Whereas components of the underlying OS are written in C or C++, user applications are built for Android in Java. Even the built-in applications are written in Java. With the exception of some Linux exploratory exercises in [chapter 13](#) and the Native Developer Kit (NDK) in [chapter 19](#), all the code examples in this book are written in Java, using the Android software development kit (SDK).

One feature of the Android platform is that there's no difference between the built-in applications and applications that you create with the SDK. This means that you can write powerful applications to tap into the resources available on the device. [Figure 1.1](#) shows the relationship between Android and the hardware it runs on. The most notable feature of Android might be that it's open source; missing elements can and will be provided by the global developer community. Android's Linux kernel-based OS doesn't come with a sophisticated shell environment, but because the platform is open, you can write and install shells on a device. Likewise, multimedia codecs can be supplied by third-party developers and don't need to rely on Google or anyone else to provide new functionality. That's the power of an open source platform brought to the mobile market.

Figure 1.1. Android is software only. By leveraging its Linux kernel to interface with the hardware, Android runs on many different devices from multiple cell phone manufacturers. Developers write applications in Java.



Platform vs. Device Throughout this book, wherever code must be tested or exercised on a device, a software-based emulator is typically employed. An exception is in [chapter 14](#) where Bluetooth and Sensors are exercised. See [chapter 2](#) for information on how to set up and use the Android emulator.

The term *platform* refers to Android itself—the software—including all the binaries, code libraries, and tool chains. This book focuses on the Android platform; the Android emulators available in the SDK are simply components of the Android platform.

With all of that as a backdrop, creating a successful mobile platform is clearly a nontrivial task involving numerous players. Android is an ambitious undertaking, even for Google, a company of seemingly boundless resources and moxie—and they’re getting the job done. Within a span of three years, Android has seen numerous major software releases, the release of multiple handsets across most major mobile carriers in the global market, and most recently the introduction of Android-powered tablets.

Now that you've got an introduction to what Android is, let's look at the why and where of Android to provide some context and set the perspective for Android's introduction to the marketplace. After that, it's on to exploring the platform itself!

1.2. UNDERSTANDING THE ANDROID MARKET

Android promises to have something for everyone. It aims to support a variety of hardware devices, not just high-end ones typically associated with expensive smartphones. Of course, Android users will enjoy improved performance on a more powerful device, considering that it sports a comprehensive set of computing features. But how well can Android scale up and down to a variety of markets and gain market and mind share? How quickly can the smartphone market become the standard? Some folks are still clinging to phone-only devices, even though smartphones are growing rapidly in virtually every demographic. Let's look at Android from the perspective of a few existing players in the marketplace. When you're talking about the cellular market, the place to start is at the top, with the carriers, or as they're sometimes referred to, the *mobile operators*.

1.2.1. Mobile operators

Mobile operators (the cell phone companies such as AT&T and Verizon) are in the business, first and foremost, of selling subscriptions to their services. Shareholders want a return on their investment, and it's hard to imagine an industry where there's a larger investment than in a network that spans such broad geographic territory. To the mobile operator, cell phones are simultaneously a conduit for services, a drug to entice subscribers, and an annoyance to support and lock down.

Some mobile operators are embracing Android as a platform to drive new data services across the excess capacity operators have built into their networks. Data services represent high-premium services and high-margin revenues for the operator. If Android can help drive those revenues for the mobile operator, all the better.

Other mobile operators feel threatened by Google and the potential of "free wireless," driven by advertising revenues and an upheaval of the market. Another challenge for mobile operators is that they want the final say on what services are enabled across their networks. Historically, handset manufacturers complain that their devices are handicapped and don't exercise all the features designed into them because mobile operators lack the capability or willingness to support those features. An encouraging sign is that there are mobile operators involved in the Open Handset Alliance.

Let's move on to a comparison of Android and existing cell phones on the market today.

1.2.2. Android vs. the feature phones

The majority of cell phones on the market continue to be consumer flip phones and *feature phones*—phones that aren’t smartphones.¹ These phones are the ones consumers get when they walk into the retailer and ask what can be had for free. These consumers are the “I just want a phone” customers. Their primary interest is a phone for voice communications, an address book, and increasingly, texting. They might even want a camera. Many of these phones have additional capabilities such as mobile web browsing, but because of relatively poor user experience, these features aren’t employed heavily. The one exception is text messaging, which is a dominant application no matter the classification of device. Another increasingly in-demand category is location-based services, which typically use the *Global Positioning System (GPS)*.

¹ About 25% of phones sold in the second quarter of 2011 were smartphones: <http://www.gartner.com/it/page.jsp?id=1764714>.

Android’s challenge is to scale down to this market. Some of the bells and whistles in Android can be left out to fit into lower-end hardware. One of the big functionality gaps on these lower-end phones is the web experience the user gets. Part of the problem is screen size, but equally challenging is the browser technology itself, which often struggles to match the rich web experience of desktop computers. Android features the market-leading WebKit browser engine, which brings desktop-compatible browsing to the mobile arena. [Figure 1.2](#) shows WebKit in action on Android. If a rich web experience can be effectively scaled down to feature phone class hardware, it would go a long way toward penetrating this end of the market. [Chapter 16](#) takes a close look at using web development skills for creating Android applications.

Figure 1.2. Android’s built-in browser technology is based on WebKit’s browser engine.



WebKit The WebKit (www.webkit.org) browser engine is an open source project that powers the browser found in Macs (Safari) and is the engine behind Mobile Safari, which is the browser on the iPhone. It's not a stretch to say that the browser experience is one of a few features that made the iPhone popular out of the gate, so its inclusion in Android is a strong plus for Android's architecture.

Software at the lower end of the market generally falls into one of two camps:

- *Qualcomm's BREW environment*—BREW stands for Binary Runtime Environment for Wireless. For a high-volume example of BREW technology,

consider Verizon's Get It Now-capable devices, which run on this platform. The challenge for software developers who want to gain access to this market is that the bar to get an application on this platform is high, because everything is managed by the mobile operator, with expensive testing and revenue-sharing fee structures. The upside to this platform is that the mobile operator collects the money and disburses it to the developer after the sale, and often these sales recur monthly. Just about everything else is a challenge to the software developer.

Android's open application environment is more accessible than BREW.

- *Java ME*, or *Java Platform, Micro Edition*—A popular platform for this class of device. The barrier to entry is much lower for software developers. Java ME developers will find a same-but-different environment in Android. Android isn't strictly a Java ME-compatible platform, but the Java programming environment found in Android is a plus for Java ME developers. There are some projects underway to create a bridge environment, with the aim of enabling Java ME applications to be compiled and run for Android. Gaming, a better browser, and anything to do with texting or social applications present fertile territory for Android at this end of the market.

Although the majority of cell phones sold worldwide are not considered smartphones, the popularity of Android (and other capable platforms) has increased demand for higher-function devices. That's what we're going to discuss next.

1.2.3. Android vs. the smartphones

Let's start by naming the major smartphone players: Symbian (big outside North America), BlackBerry from Research in Motion, iPhone from Apple, Windows (Mobile, SmartPhone, and now Phone 7), and of course, the increasingly popular Android platform.

One of the major concerns of the smartphone market is whether a platform can synchronize data and access Enterprise Information Systems for corporate users. Device-management tools are also an important factor in the enterprise market. The browser experience is better than with the lower-end phones, mainly because of larger displays and more intuitive input methods, such as a touch screen, touch pad, slide-out keyboard, or jog dial.

Android's opportunity in this market is to provide a device and software that people want. For all the applications available for the iPhone, working with Apple can be a challenge; if the core device doesn't suit your needs, there's little room to maneuver because of the limited models available and historical carrier exclusivity. Now that email, calendaring, and contacts can sync with Microsoft Exchange, the corporate environment is more accessible, but Android will continue to fight the battle of scaling the Enterprise walls. Later Android releases have added improved support for the Microsoft Exchange platform, though third-party solutions still out-perform the built-in offerings. BlackBerry is dominant because of its intuitive email capabilities, and the

Microsoft platforms are compelling because of tight integration to the desktop experience and overall familiarity for Windows users. iPhone has surprisingly good integration with Microsoft Exchange—for Android to compete in this arena, it must maintain parity with iPhone on Enterprise support.

You've seen how Android stacks up next to feature phones and smartphones. Next, we'll see whether Android, the open source mobile platform, can succeed as an open source project.

1.2.4. Android vs. itself

Android will likely always be an open source project, but to succeed in the mobile market, it must sell millions of units and stay fresh. Even though Google briefly entered the device fray with its Nexus One and Nexus S phones, it's not a hardware company. Historically, Android-powered devices have been brought to market by others such as HTC, Samsung, and Motorola, to name the larger players. Starting in mid-2011, Google began to further flex its muscles with the acquisition of Motorola's mobile business division. Speculation has it that Google's primary interest is in Motorola's patent portfolio, because the intellectual property scene has heated up considerably. A secondary reason may be to acquire the Motorola Xoom platform as Android continues to reach beyond cell phones into tablets and beyond.

When a manufacturer creates an Android-powered device, they start with the Android Open Source Platform (AOSP) and then extend it to meet their need to differentiate their offerings. Android isn't the first open source phone, but it's the first from a player with the market-moving weight of Google leading the charge. This market leadership position has translated to impressive unit sales across multiple manufacturers and markets around the globe. With a multitude of devices on the market, can Android keep the long-anticipated fragmentation from eroding consumer and investor confidence?

Open source is a double-edged sword. On one hand, the power of many talented people and companies working around the globe and around the clock to deliver desirable features is a force to be reckoned with, particularly in comparison with a traditional, commercial approach to software development. This topic has become trite because the benefits of open source development are well documented. On the other hand, how far will the competing manufacturers extend and potentially split Android? Depending on your perspective, the variety of Android offerings is a welcome alternative to a more monolithic iPhone device platform where consumers have few choices available.

Another challenge for Android is that the licensing model of open source code used in commercial offerings can be sticky. Some software licenses are more restrictive than others, and some of those restrictions pose a challenge to the open source label. At the same time, Android licensees need to protect their investment, so licensing is an important topic for the commercialization of Android.

1.2.5. Licensing Android

Android is released under two different open source licenses. The Linux kernel is released under the *GNU General Public License (GPL)* as is required for anyone licensing the open source OS kernel. The Android platform, excluding the kernel, is licensed under the *Apache Software License (ASL)*. Although both licensing models are open source-oriented, the major difference is that the Apache license is considered friendlier toward commercial use. Some open source purists might find fault with anything but complete openness, source-code sharing, and noncommercialization; the ASL attempts to balance the goals of open source with commercial market forces. So far there has been only one notable licensing hiccup impacting the Android mod community, and that had more to do with the gray area of full system images than with a manufacturer's use of Android on a mainstream product release. Currently, Android is facing intellectual property challenges; both Microsoft and Apple are bringing litigation against Motorola and HTC for the manufacturer's Android-based handsets.

The high-level, market-oriented portion of the book has now concluded! The remainder of this book is focused on Android application development. Any technical discussion of a software environment must include a review of the layers that compose the environment, sometimes referred to as a *stack* because of the layer-upon-layer construction. Next up is a high-level breakdown of the components of the Android stack.

Selling applications

A mobile platform is ultimately valuable only if there are applications to use and enjoy on that platform. To that end, the topic of buying and selling applications for Android is important and gives us an opportunity to highlight a key difference between Android and the iPhone. The Apple App Store contains software titles for the iPhone—lots of them. But Apple's somewhat draconian grip on the iPhone software market requires that all applications be sold through its venue. Although Apple's digital rights management (DRM) is the envy of the market, this approach can pose a challenging environment for software developers who might prefer to make their application available through multiple distribution channels.

Contrast Apple's approach to application distribution with the freedom Android developers enjoy to ship applications via traditional venues such as freeware and shareware, and commercially through various marketplaces, including their own website! For software publishers who want the focus of an on-device shopping experience, Google has launched and continues to mature the Android Market. For software developers who already have titles for other platforms such as Windows Mobile, Palm, and BlackBerry, traditional software markets such as Handango (www.Handango.com) also support selling Android applications. Handango and its ilk are important outlets; consumers new to Android will likely visit sites such as Handango

because that might be where they first purchased one of their favorite applications for their prior device.

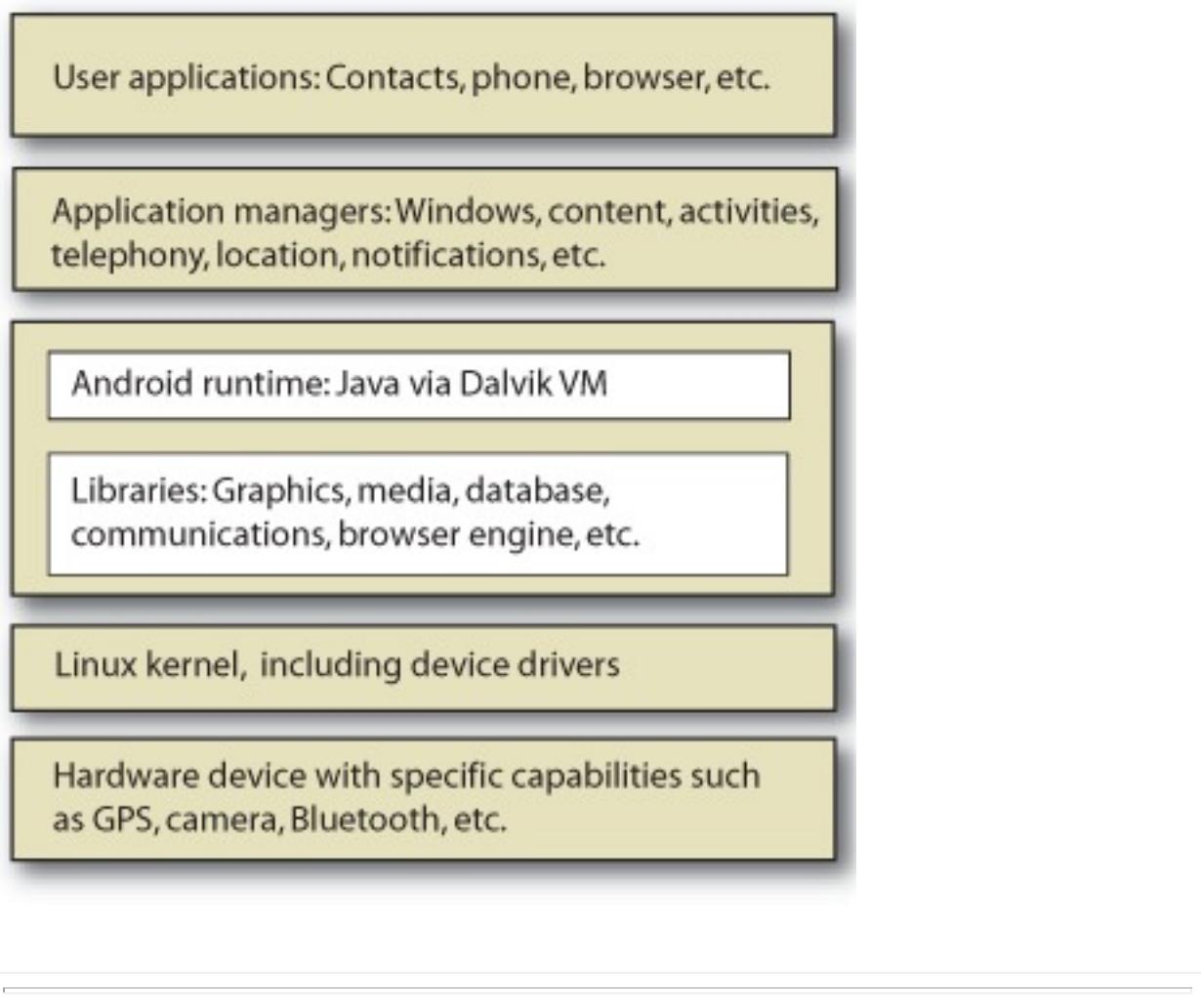
1.3. THE LAYERS OF ANDROID

The Android stack includes an impressive array of features for mobile applications. In fact, looking at the architecture alone, without the context of Android being a platform designed for mobile environments, it would be easy to confuse Android with a general computing environment. All the major components of a computing platform are there. Here's a quick rundown of prominent components of the Android stack:

- A *Linux kernel* that provides a foundational hardware abstraction layer, as well as core services such as process, memory, and filesystem management. The kernel is where hardware-specific drivers are implemented—capabilities such as Wi-Fi and Bluetooth are here. The Android stack is designed to be flexible, with many optional components that largely rely on the availability of specific hardware on a given device. These components include features such as touch screens, cameras, GPS receivers, and accelerometers.
- *Prominent code libraries*, including the following:
 - Browser technology from WebKit, the same open source engine powering Mac's Safari and the iPhone's Mobile Safari browser. WebKit has become the de facto standard for most mobile platforms.
 - Database support via SQLite, an easy-to-use SQL database.
 - Advanced graphics support, including 2D, 3D, animation from Scalable Games Language (SGL), and OpenGL ES.
 - Audio and video media support from PacketVideo's OpenCORE, and Google's own Stagefright media framework.
 - Secure Sockets Layer (SSL) capabilities from the Apache project.
- *An array of managers* that provide services for
 - Activities and views
 - Windows
 - Location-based services
 - Telephony
 - Resources
- The Android runtime, which provides
 - Core Java packages for a nearly full-featured Java programming environment. Note that this isn't a Java ME environment.
 - The Dalvik VM, which employs services of the Linux-based kernel to provide an environment to host Android applications.

Both core applications and third-party applications (such as the ones you'll build in this book) run in the Dalvik VM, atop the components we just listed. You can see the relationship among these layers in [figure 1.3](#).

Figure 1.3. The Android stack offers an impressive array of technologies and capabilities.



Tip

Without question, Android development requires Java programming skills. To get the most out of this book, be sure to brush up on your Java programming knowledge. There are many Java references on the internet, and no shortage of Java books on the market. An excellent source of Java titles can be found at www.manning.com/catalog/java.

Now that we've shown you the obligatory stack diagram and introduced all the layers, let's look more in depth at the runtime technology that underpins Android.

1.3.1. Building on the Linux kernel

Android is built on a Linux kernel and on an advanced, optimized VM for its Java applications. Both technologies are crucial to Android. The Linux kernel component of the Android stack promises agility and portability to take advantage of numerous hardware options for future Android-equipped phones. Android's Java environment is key: it makes Android accessible to programmers because of both the number of Java software developers and the rich environment that Java programming has to offer.

Why use Linux for a phone? Using a full-featured platform such as the Linux kernel provides tremendous power and capabilities for Android. Using an open source foundation unleashes the capabilities of talented individuals and companies to move the platform forward. Such an arrangement is particularly important in the world of mobile devices, where products change so rapidly. The rate of change in the mobile market makes the general computer market look slow and plodding. And, of course, the Linux kernel is a proven core platform. Reliability is more important than performance when it comes to a mobile phone, because voice communication is the primary use of a phone. All mobile phone users, whether buying for personal use or for a business, demand voice reliability, but they still want cool data features and will purchase a device based on those features. Linux can help meet this requirement.

Speaking to the rapid rate of phone turnover and accessories hitting the market, another advantage of using Linux as the foundation of the Android platform stack is that it provides a hardware abstraction layer; the upper levels remain unchanged despite changes in the underlying hardware. Of course, good coding practices demand that user applications fail gracefully in the event a resource isn't available, such as a camera not being present in a particular handset model. As new accessories appear on the market, drivers can be written at the Linux level to provide support, just as on other Linux platforms. This architecture is already demonstrating its value; Android devices are already available on distinct hardware platforms. HTC, Motorola, and others have released Android-based devices built on their respective hardware platforms. User applications, as well as core Android applications, are written in Java and are compiled into *byte codes*. Byte codes are interpreted at runtime by an interpreter known as a *virtual machine* (VM).

1.3.2. Running in the Dalvik VM

The Dalvik VM is an example of the need for efficiency, the desire for a rich programming environment, and even some intellectual property constraints, colliding, with innovation as the result. Android's Java environment provides a rich application platform and is accessible because of the popularity of Java itself. Also, application

performance, particularly in a low-memory setting such as you find in a mobile phone, is paramount for the mobile market. But this isn't the only issue at hand.

Android isn't a Java ME platform. Without commenting on whether this is ultimately good or bad for Android, there are other forces at play here. There's the matter of Java VM licensing from Oracle. From a high level, Android's code environment is Java. Applications are written in Java, which is compiled to Java byte codes and subsequently translated to a similar but different representation called *dex files*. These files are logically equivalent to Java byte codes, but they permit Android to run its applications in its own VM that's both (arguably) free from Oracle's licensing clutches and an open platform upon which Google, and potentially the open source community, can improve as necessary. Android is facing litigation challenges from Oracle about the use of Java.

Note

From the mobile application developer's perspective, Android is a Java environment, but the runtime isn't strictly a Java VM. This accounts for the incompatibilities between Android and proper Java environments and libraries. If you have a code library that you want to reuse, your best bet is to assume that your code is nearly *source compatible*, attempt to compile it into an Android project, and then determine how close you are to having usable code.

The important things to know about the Dalvik VM are that Android applications run inside it and that it relies on the Linux kernel for services such as process, memory, and filesystem management.

Now that we've discussed the foundational technologies in Android, it's time to focus on Android application development. The remainder of this chapter discusses high-level Android application architecture and introduces a simple Android application. If you're not comfortable or ready to begin coding, you might want to jump to [chapter 2](#), where we introduce the development environment step-by-step.

1.4. THE INTENT OF ANDROID DEVELOPMENT

Let's jump into the fray of Android development, focus on an important component of the Android platform, and expand to take a broader view of how Android applications are constructed.

An important and recurring theme of Android development is the `Intent`. An `Intent` in Android describes what you want to do. An `Intent` might look like “I want to look up a contact record” or “Please launch this website” or “Show the order confirmation screen.” `Intents` are important because they not only facilitate navigation in an innovative way, as we’ll discuss next, but also represent the most important aspect of Android coding. Understand the `Intent` and you’ll understand Android.

Note

Instructions for setting up the Eclipse development environment are in [appendix A](#). This environment is used for all Java examples in this book. [Chapter 2](#) goes into more detail on setting up and using the development tools.

The code examples in this chapter are primarily for illustrative purposes. We reference and introduce classes without necessarily naming specific Java packages. Subsequent chapters take a more rigorous approach to introducing Android-specific packages and classes.

Next, we’ll look at the foundational information about why `Intents` are important, and then we’ll describe how `Intents` work. Beyond the introduction of the `Intent`, the remainder of this chapter describes the major elements of Android application development, leading up to and including the first complete Android application that you’ll develop.

1.4.1. Empowering intuitive UIs

The power of Android’s application framework lies in the way it brings a web mindset to mobile applications. This doesn’t mean the platform has only a powerful browser and is limited to clever JavaScript and server-side resources, but rather it goes to the core of how the Android platform works and how users interact with the mobile device. The power of the internet is that everything is just a click away. Those clicks are known as *Uniform Resource Locators (URLs)*, or alternatively, *Uniform Resource Identifiers (URIs)*. Using effective URIs permits easy and quick access to the information users need and want every day. “Send me the link” says it all.

Beyond being an effective way to get access to data, why is this URI topic important, and what does it have to do with `Intents`? The answer is nontechnical but crucial: the way a mobile user navigates on the platform is crucial to its commercial success. Platforms that replicate the desktop experience on a mobile device are acceptable to only a small

percentage of hardcore power users. Deep menus and multiple taps and clicks are generally not well received in the mobile market. The mobile application, more than in any other market, demands intuitive ease of use. A consumer might buy a device based on cool features that were enumerated in the marketing materials, but that same consumer is unlikely to even touch the instruction manual. A UI's usability is highly correlated with its market penetration. UIs are also a reflection of the platform's data access model, so if the navigation and data models are clean and intuitive, the UI will follow suit.

Now we're going to introduce `Intents` and `IntentFilters`, Android's innovative navigation and triggering mechanisms.

1.4.2. Intents and how they work

`Intents` and `IntentFilters` bring the “click it” paradigm to the core of mobile application use (and development) for the Android platform:

- An `Intent` is a declaration of need. It's made up of a number of pieces of information that describe the desired action or service. We're going to examine the requested action and, generically, the data that accompanies the requested action.
- An `IntentFilter` is a declaration of capability and interest in offering assistance to those in need. It can be generic or specific with respect to which `Intents` it offers to service.

The action attribute of an `Intent` is typically a verb: for example `VIEW`, `PICK`, or `EDIT`. A number of built-in `Intent` actions are defined as members of the `Intent` class, but application developers can create new actions as well. To view a piece of information, an application employs the following `Intent` action:

```
android.content.Intent.ACTION_VIEW
```

The data component of an `Intent` is expressed in the form of a URI and can be virtually any piece of information, such as a contact record, a website location, or a reference to a media clip. Table 1.1 lists some Android URI examples.

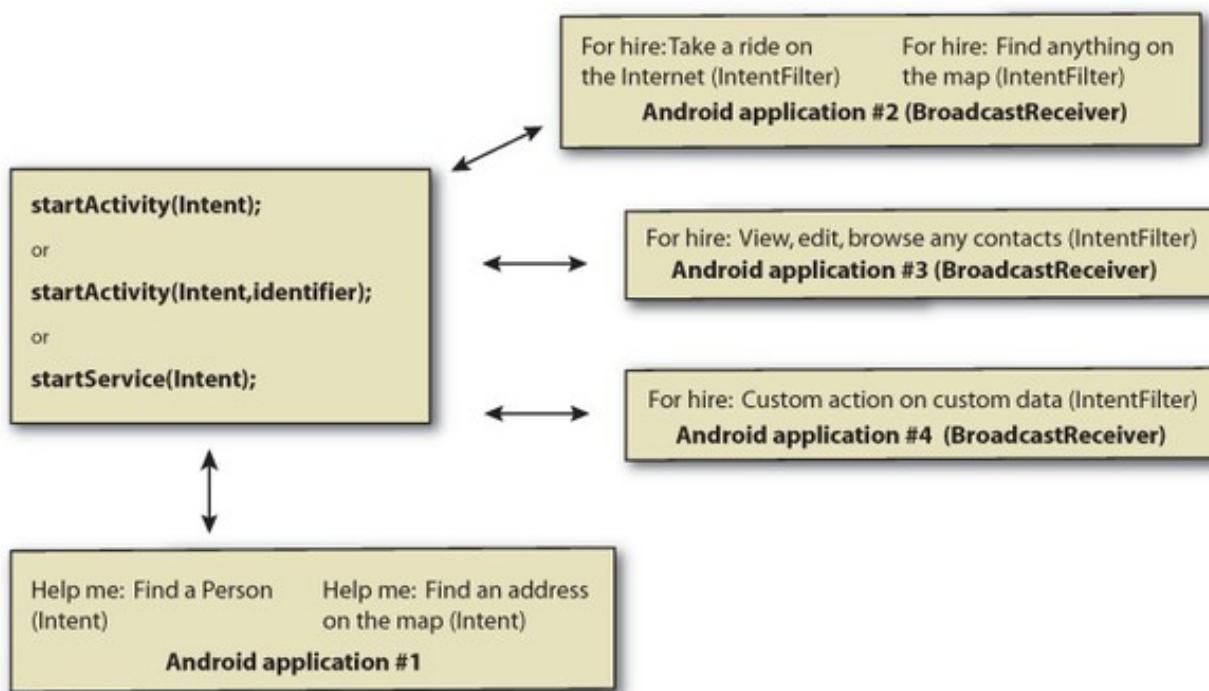
Table 1.1. Commonly employed URIs in Android

Type of information	URI data
Contact lookup	<code>content://contacts/people</code>
Map lookup/search	<code>Geo:o,o?q=23+Route+206+Stanhope+NJ</code>
Browser launch to a specific website	<code>http://www.google.com/</code>

The `IntentFilter` defines the relationship between the `Intent` and the application. `IntentFilters` can be specific to the data portion of the `Intent`, the action portion, or both. `IntentFilters` also contain a field known as a *category*. The category helps classify the action. For example, the category named `CATEGORY_LAUNCHER` instructs Android that the `Activity` containing this `IntentFilter` should be visible in the main application launcher or home screen.

When an `Intent` is dispatched, the system evaluates the available `Activities`, `Services`, and registered `BroadcastReceivers` (more on these in [section 1.5](#)) and dispatches the `Intent` to the most appropriate recipient. [Figure 1.4](#) depicts this relationship among `Intents`, `IntentFilters`, and `BroadcastReceivers`.

Figure 1.4. `Intents` are distributed to Android applications, which register themselves by way of the `IntentFilter`, typically in the `AndroidManifest.xml` file.



`IntentFilters` are often defined in an application's `AndroidManifest.xml` file with the `<intent-filter>` tag. The `AndroidManifest.xml` file is essentially an application descriptor file, which we'll discuss later in this chapter.

A common task on a mobile device is looking up a specific contact record for the purpose of initiating a call, sending a text message, or looking up a snail-mail address when you're standing in line at the neighborhood pack-and-ship store. Or a user might want to view a specific piece of information, say a contact record for user 1234. In these cases, the action is `ACTION_VIEW` and the data is a specific contact record identifier. To carry out these kinds of tasks, you create an `Intent` with the action set

to `ACTION_VIEW` and a URI that represents the specific person of interest. Here are some examples:

- The URI that you would use to contact the record for user 1234: `content://contacts/people/1234`
- The URI for obtaining a list of all contacts: `content://contacts/people`

The following code snippet shows how to `PICK` a contact record:

```
Intent pickIntent = new Intent(Intent.ACTION_PICK,Uri.parse("content://contacts/people"));

startActivity(pickIntent);
```

An `Intent` is evaluated and passed to the most appropriate handler. In the case of picking a contact record, the recipient would likely be a built-in Activity named `com.google.android.phone.Dialer`. But the best recipient of this `Intent` might be an `Activity` contained in the same custom Android application (the one you build), a built-in application (as in this case), or a third-party application on the device. Applications can leverage existing functionality in other applications by creating and dispatching an `Intent` that requests existing code to handle the `Intent` rather than writing code from scratch. One of the great benefits of employing `Intents` in this manner is that the same UIs get used frequently, creating familiarity for the user. *This is particularly important for mobile platforms where the user is often neither tech-savvy nor interested in learning multiple ways to accomplish the same task, such as looking up a contact on the phone.*

The `Intents` we've discussed thus far are known as *implicit Intents*, which rely on the `IntentFilter` and the Android environment to dispatch the `Intent` to the appropriate recipient. Another kind of `Intent` is the *explicit Intent*, where you can specify the exact class that you want to handle the `Intent`. Specifying the exact class is helpful when you know exactly which `Activity` you want to handle the `Intent` and you don't want to leave anything to chance in terms of what code is executed. To create an explicit `Intent`, use the overloaded `Intent` constructor, which takes a class as an argument:

```
public void onClick(View v) {

    try {
        startActivityForResult(new Intent(v.getContext(),RefreshJobs.class),0);
    } catch (Exception e) {
        . . .
    }
}
```

}

These examples show how an Android developer creates an `Intent` and asks for it to be handled. Similarly, an Android application can be deployed with an `IntentFilter`, indicating that it responds to `Intents` that were already defined on the system, thereby publishing new functionality for the platform. This facet alone should bring joy to independent software vendors (ISVs) who've made a living by offering better contact managers and to-do list management software titles for other mobile platforms.

The power and the complexity of Intents

It's not hard to imagine that an absolutely unique user experience is possible with Android because of the variety of `Activities` with specific `IntentFilters` that are installed on any given device. It's architecturally feasible to upgrade various aspects of an Android installation to provide sophisticated functionality and customization. Though this might be a desirable characteristic for the user, it can be troublesome for someone providing tech support who has to navigate a number of components and applications to troubleshoot a problem.

Because of the potential for added complexity, this approach of ad hoc system patching to upgrade specific functionality should be entertained cautiously and with your eyes wide open to the potential pitfalls associated with this approach.

`Intent` resolution, or *dispatching*, takes place at runtime, as opposed to when the application is compiled. You can add specific `Intent`-handling features to a device, which might provide an upgraded or more desirable set of functionality than the original shipping software. This runtime dispatching is also referred to as *late binding*.

Thus far, this discussion of `Intents` has focused on the variety of `Intents` that cause UI elements to be displayed. Other `Intents` are more event-driven than task-oriented, as our earlier contact record example described. For example, you also use the `Intent` class to notify applications that a text message has arrived. `Intents` are a central element to Android; we'll revisit them on more than one occasion.

Now that we've explained `Intents` as the catalyst for navigation and event flow on Android, let's jump to a broader view and discuss the Android application lifecycle and the key components that make Android tick. The `Intent` will come into better focus as we further explore Android throughout this book.

1.5. FOUR KINDS OF ANDROID COMPONENTS

Let's build on your knowledge of the `Intent` and `IntentFilter` classes and explore the four primary components of Android applications, as well as their relation to the Android process model. We'll include code snippets to provide a taste of Android application development. We're going to leave more in-depth examples and discussion for later chapters.

Note

A particular Android application might not contain all of these elements but will have at least one of these elements, and could have all of them.

1.5.1. Activity

An application might have a UI, but it doesn't have to have one. If it has a UI, it'll have at least one `Activity`.

The easiest way to think of an Android `Activity` is to relate it to a visible screen, because more often than not there's a one-to-one relationship between an `Activity` and a UI screen. This relationship is similar to that of a controller in the MVC paradigm.

Android applications often contain more than one `Activity`. Each `Activity` displays a UI and responds to system- and user-initiated events. The `Activity` employs one or more `Views` to present the actual UI elements to the user. The `Activity` class is extended by user classes, as shown in the following listing.

Listing 1.1. A basic `Activity` in an Android application

```
package com.msi.manning.chapter1;  
  
import android.app.Activity;  
  
import android.os.Bundle;  
  
public class Activity1 extends Activity {  
  
    @Override  
  
    public void onCreate(Bundle savedInstanceState) {
```

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

The `Activity` class is part of the `android.app` Java package, found in the Android runtime. The Android runtime is deployed in the `android.jar` file. The class `Activity1` extends the class `Activity`, which we'll examine in detail in [chapter 3](#). One of the primary tasks an `Activity` performs is displaying UI elements, which are implemented as `Views` and are typically defined in XML layout files. [Chapter 3](#) goes into more detail on `Views` and `Resources`.

You say Intent; I say Intent

The `Intent` class is used in similar sounding but very different scenarios.

Some `Intents` are used to assist in navigating from one `Activity` to the next, such as the example given earlier of viewing a contact record. Activities are the targets of these kinds of `Intents`, which are used with the `startActivity` and `startActivityForResult` methods.

Also, a `Service` can be started by passing an `Intent` to the `startService` method.

`BroadcastReceivers` receive `Intents` when responding to system-wide events, such as a ringing phone or an incoming text message.

Moving from one `Activity` to another is accomplished with the `startActivity()` method or the `startActivityForResult()` method when you want a synchronous call/result paradigm. The argument to these methods is an instance of an `Intent`.

The `Activity` represents a visible application component within Android. With assistance from the `View` class, which we'll cover in [chapter 3](#), the `Activity` is the most commonly employed Android application component. Android 3.0 introduced a new kind of application component, the `Fragment`. `Fragments`, which are related to `Activity`s and have their own life cycle, provide more granular application control than `Activity`s. `Fragments` are covered in [Chapter 20](#). The next topic of interest is the `Service`, which runs in the background and doesn't generally present a direct UI.

1.5.2. Service

If an application is to have a long lifecycle, it's often best to put it into a `Service`. For example, a background data-synchronization utility should be implemented as a `Service`. A best practice is to launch `Services` on a periodic or as-needed basis, triggered by a system alarm, and then have the `Service` terminate when its task is complete.

Like the `Activity`, a `Service` is a class in the Android runtime that you should extend, as shown in the following listing. This example extends a `Service` and periodically publishes an informative message to the Android log.

Listing 1.2. A simple example of an Android `Service`

```
package com.msi.manning.chapter1;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
public class Service1 extends Service implements Runnable {
    public static final String tag = "service1";
    private int counter = 0;
    @Override
    protected void onCreate() {           ← ② Initialization
        super.onCreate();
        Thread aThread = new Thread (this);
        aThread.start();
    }
    public void run() {
        while (true) {
            try {
                Log.i(tag,"service1 firing : # " + counter++);
                Thread.sleep(10000);
            } catch(Exception ee) {
                Log.e(tag,ee.getMessage());
            }
        }
    }
    @Override
    public IBinder onBind(Intent intent) {   ← ③ Handle
        return null;                      binding request
    }
}
```

Diagram annotations:

- ① Extend Service class: An arrow points from the `Service1` class name to the `extends Service` line.
- ② Initialization: An arrow points from the `onCreate()` method to the `super.onCreate()` call.
- ③ Handle binding request: An arrow points from the `onBind()` method to the `return null;` statement.

This example requires that the package `android.app.Service` be imported. This package contains the `Service` class. This example also demonstrates Android's logging

mechanism `android.util.Log`, which is useful for debugging purposes. (Many examples in this book include using the logging facility. We'll discuss logging in more depth in [chapter 2](#).) The `Service1` class ① extends the `Service` class. This class implements the `Runnable` interface to perform its main task on a separate thread.

The `onCreate` method ② of the `Service` class permits the application to perform initialization-type tasks. We're going to talk about the `onBind()` method ③ in further detail in [chapter 4](#), when we'll explore the topic of interprocess communication in general.

Services are started with the `startService(Intent)` method of the abstract `Context` class. Note that, again, the `Intent` is used to initiate a desired result on the platform.

Now that the application has a UI in an `Activity` and a means to have a background task via an instance of a `Service`, it's time to explore the `BroadcastReceiver`, another form of Android application that's dedicated to processing `Intents`.

1.5.3. BroadcastReceiver

If an application wants to receive and respond to a global event, such as a ringing phone or an incoming text message, it must register as a `BroadcastReceiver`. An application registers to receive `Intents` in one of the following ways:

- The application can implement a `<receiver>` element in the `AndroidManifest.xml` file, which describes the `BroadcastReceiver`'s class name and enumerates its `IntentFilters`. Remember, the `IntentFilter` is a descriptor of the `Intent` an application wants to process. If the receiver is registered in the `AndroidManifest.xml` file, the application doesn't need to be running in order to be triggered. When the event occurs, the application is started automatically upon notification of the triggering event. Thankfully, all this housekeeping is managed by the Android OS itself.
- An application can register at runtime via the `Context` class's `registerReceiver` method.

Like `Services`, `BroadcastReceivers` don't have a UI. Even more important, the code running in the `onReceive` method of a `BroadcastReceiver` should make no assumptions about persistence or long-running operations. If the `BroadcastReceiver` requires more than a trivial amount of code execution, it's recommended that the code initiate a request to a `Service` to complete the requested functionality because the `Service` application component is designed for longer-running operations whereas the `BroadcastReceiver` is meant for responding to various triggers.

Note

The familiar `Intent` class is used in triggering `BroadcastReceivers`. The parameters will differ, depending on whether you're starting an `Activity`, a `Service`, or a `BroadcastReceiver`, but it's the same `Intent` class that's used throughout the Android platform.

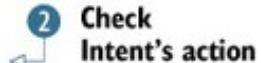
A `BroadcastReceiver` implements the abstract method `onReceive` to process incoming `Intents`. The arguments to the method are a `Context` and an `Intent`. The method returns `void`, but a handful of methods are useful for passing back results, including `setResult`, which passes back to the invoker an integer return code, a `String` return value, and a `Bundle` value, which can contain any number of objects.

The following listing is an example of a `BroadcastReceiver` triggering upon receipt of an incoming text message.

Listing 1.3. A sample `BroadcastReceiver`

```
package com.msi.manning.unlockingandroid;
import android.content.Context;
import android.content.Intent;
import android.util.Log;
import android.content.BroadcastReceiver
public class MySMSMailBox extends BroadcastReceiver {
    public static final String tag = "MySMSMailBox";
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i(tag, "onReceive");
        if (intent.getAction().equals
            ("android.provider.Telephony.SMS_RECEIVED")) {
            Log.i(tag, "Found our Event!");
        }
    }
}
```

 1 Tag used in logging

 2 Check Intent's action

We need to discuss a few items in this listing. The class `MySMSMailBox` extends the `BroadcastReceiver` class. This subclass approach is the most straightforward way to employ a `BroadcastReceiver`. (Note the class name `MySMSMailBox`; it'll be used in the

AndroidManifest.xml file, shown in [listing 1.4](#).) The `tag` variable 1 is used in conjunction with the logging mechanism to assist in labeling messages sent to the console log on the emulator. Using a tag in the log enables you to filter and organize log messages in the console. (We discuss the log mechanism in more detail in [chapter 2](#).) The `onReceive` method is where all the work takes place in a `BroadcastReceiver`; you must implement this method. A given `BroadcastReceiver` can register

multiple `IntentFilters`. A `BroadcastReceiver` can be instantiated for an arbitrary number of `Intents`.

It's important to make sure that the application handles the appropriate `Intent` by checking the action of the incoming `Intent` ②. When the application receives the desired `Intent`, it should carry out the specific functionality that's required. A common task in an SMS-receiving application is to parse the message and display it to the user via the capabilities found in the `NotificationManager`. (We'll discuss notifications in [chapter 8](#).) In [listing 1.3](#), you simply record the action to the log.

In order for this `BroadcastReceiver` to fire and receive this `Intent`, the `BroadcastReceiver` is listed in the `AndroidManifest.xml` file, along with an appropriate `intentfilter` tag, as shown in the following listing. This listing contains the elements required for the application to respond to an incoming text message.

Listing 1.4. `AndroidManifest.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.unlockingandroid">
    <uses-permission android:name="android.permission.RECEIVE_SMS" /> ①
    <application android:icon="@drawable/icon">
        <activity android:name=".Activity1" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".MySMSMailBox" > ② Receiver tag;
            <intent-filter>                                note dot prefix
                <action android:name="android.provider.Telephony.SMS_RECEIVED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Certain tasks within the Android platform require the application to have a designated privilege. To give an application the required permissions, use

the `<usespermission>` tag ①. (We'll discuss this tag in detail in [section 1.6](#).) The `<receiver>` tag contains the class name of the class implementing the `BroadcastReceiver`. In this example, the class name is `MySMSMailBox`, from the package `com.msi.manning.unlockingandroid`. Be sure to note the dot that precedes the name ②. This dot is required. If your application isn't behaving as expected, one of the first places to check is your `Android.xml` file, and look for the dot before the class name!

The `IntentFilter` is defined in the `<intent-filter>` tag. The desired action in this example is `android.provider.Telephony.SMS_RECEIVED`. The Android SDK contains the available actions for the standard `Intents`. Also, remember that user applications can define their own `Intents`, as well as listen for them.

Testing SMS

The emulator has a built-in set of tools for manipulating certain telephony behavior to simulate a variety of conditions, such as in-network and out-of-network coverage and placing phone calls.

To send an SMS message to the emulator, telnet to port 5554 (the port number might vary on your system), which will connect to the emulator, and issue the following command at the prompt:

```
sms send <sender's phone number> <body of text message>
```

To learn more about available commands, type `help` at the prompt.

We'll discuss these tools in more detail in [chapter 2](#).

Now that we've introduced `Intents` and the Android classes that process or handle `Intents`, it's time to explore the next major Android application topic: the `ContentProvider`, Android's preferred data-publishing mechanism.

1.5.4. ContentProvider

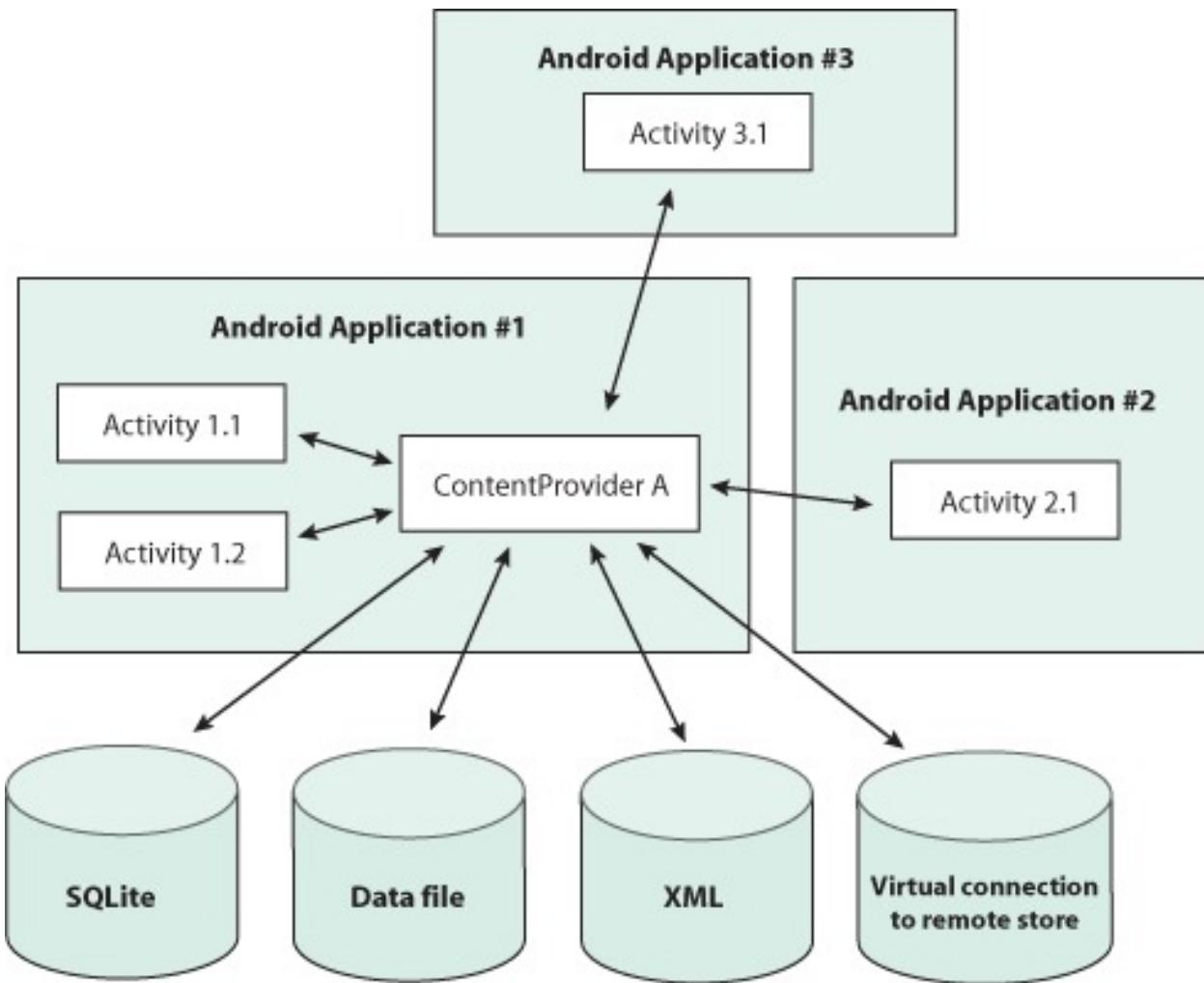
If an application manages data and needs to expose that data to other applications running in the Android environment, you should consider a `ContentProvider`. If an application component (`Activity`, `Service`, or `BroadcastReceiver`) needs to access data from another application, the component accesses the other application's `ContentProvider`. The `ContentProvider` implements a standard set of methods to permit an application to access a data store. The access might be for read or write operations, or for both. A `ContentProvider` can provide data to an `Activity` or `Service` in the same containing application, as well as to an `Activity` or `Service` contained in other applications.

A `ContentProvider` can use any form of data-storage mechanism available on the Android platform, including files, SQLite databases, or even a memory-based hash map

if data persistence isn't required. The `ContentProvider` is a data layer that provides data abstraction for its clients and centralizing storage and retrieval routines in a single place.

Sharing files or databases directly is discouraged on the Android platform, and is enforced by the underlying Linux security system, which prevents ad hoc file access from one application space to another without explicitly granted permissions. Data stored in a `ContentProvider` can be traditional data types, such as integers and strings. Content providers can also manage binary data, such as image data. When binary data is retrieved, the suggested best practice is to return a string representing the filename that contains the binary data. If a filename is returned as part of a `ContentProvider` query, the application shouldn't access the file directly; you should use the helper class, `ContentResolver`'s `openInputStream` method, to access the binary data. This approach navigates the Linux process and security hurdles, as well as keeps all data access normalized through the `ContentProvider`. [Figure 1.5](#) outlines the relationship among `ContentProviders`, data stores, and their clients.

Figure 1.5. The content provider is the data tier for Android applications and is the prescribed manner in which data is accessed and shared on the device.



A ContentProvider's data is accessed by an Android application through a Content URI. A ContentProvider defines this URI as a public static final String. For example, an application might have a data store managing material safety data sheets. The Content URI for this ContentProvider might look like this:

```
public static final Uri CONTENT_URI =
Uri.parse("content://com.msi.manning.provider.unlockingandroid/datasheets");
```

From this point, accessing a ContentProvider is similar to using Structured Query Language (SQL) in other platforms, though a complete SQL statement isn't employed. A query is submitted to the ContentProvider, including the columns desired and optional `where` and `Order By` clauses. Similar to parameterized queries in traditional SQL, parameter substitution is also supported when working with the ContentProvider class. Where do the results from the query go? In a Cursor class, naturally. We'll provide a detailed ContentProvider example in [chapter 5](#).

Note

In many ways, a `ContentProvider` acts like a database server. Although an application could contain only a `ContentProvider` and in essence be a database server, a `ContentProvider` is typically a component of a larger Android application that hosts at least one `Activity`, `Service`, or `BroadcastReceiver`.

This concludes our brief introduction to the major Android application classes. Gaining an understanding of these classes and how they work together is an important aspect of Android development. Getting application components to work together can be a daunting task. For example, have you ever had a piece of software that just didn't work properly on your computer? Perhaps you copied it from another developer or downloaded it from the internet and didn't install it properly. Every software project can encounter environment-related concerns, though they vary by platform. For example, when you're connecting to a remote resource such as a database server or FTP server, which username and password should you use? What about the libraries you need to run your application? All these topics are related to software deployment.

Before we discuss anything else related to deployment or getting an Android application to run, we need to discuss the Android file named `AndroidManifest.xml`, which ties together the necessary pieces to run an Android application on a device. A one-to-one relationship exists between an Android application and its `AndroidManifest.xml` file.

1.6. UNDERSTANDING THE ANDROIDMANIFEST.XML FILE

In the preceding sections, we introduced the common elements of an Android application. A fundamental fact of Android development is that an Android application contains at least one `Activity`, `Service`, `BroadcastReceiver`, or `ContentProvider`. Some of these elements advertise the `Intents` they're interested in processing via the `IntentFilter` mechanism. All these pieces of information need to be tied together for an Android application to execute. The glue mechanism for this task of defining relationships is the `AndroidManifest.xml` file.

The `AndroidManifest.xml` file exists in the root of an application directory and contains all the design-time relationships of a specific application and `Intents`. `AndroidManifest.xml` files act as deployment descriptors for Android applications. The following listing is an example of a simple `AndroidManifest.xml` file.

Listing 1.5. `AndroidManifest.xml` file for a basic Android application

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

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.msi.manning.unlockingandroid">

    <application android:icon="@drawable/icon">

        <activity android:name=".Activity1" 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>
```

Looking at this simple `AndroidManifest.xml` file, you see that the `manifest` element contains the obligatory namespace, as well as the Java package name containing this application. This application contains a single `Activity`, with the class name `Activity1`. Note also the `@string` syntax. Any time an `@` symbol is used in an `AndroidManifest.xml` file, it references information stored in one of the resource files. In this case, the `label` attribute is obtained from the string resource identified as `app_name`. (We discuss resources in further detail later in [chapter 3](#).) This application's lone `Activity` contains a single `IntentFilter` definition. The `IntentFilter` used here is the most common `IntentFilter` seen in Android applications. The `action android.intent.action.MAIN` indicates that this is an entry point to the application. The `category android.intent.category.LAUNCHER` places this `Activity` in the launcher window, as shown in [figure 1.6](#). It's possible to have multiple `Activity` elements in a manifest file (and thereby an application), with zero or more of them visible in the launcher window.

Figure 1.6. Applications are listed in the launcher based on their `IntentFilter`. In this example, the application `Where Do You Live` is available in the `LAUNCHER` category.



In addition to the elements used in the sample manifest file shown in [listing 1.5](#), other common tags are as follows:

- The `<service>` tag represents a `Service`. The attributes of the `<service>` tag include its class and label. A `Service` might also include the `<intent-filter>` tag.
- The `<receiver>` tag represents a `BroadcastReceiver`, which might have an explicit `<intent-filter>` tag.
- The `<uses-permission>` tag tells Android that this application requires certain security privileges. For example, if an application requires access to the contacts on a device, it requires the following tag in its `AndroidManifest.xml` file:
`<uses-permission android:name="android.permission.READ_CONTACTS" />`

We'll revisit the `AndroidManifest.xml` file a number of times throughout the book because we need to add more details about certain elements and specific coding scenarios.

Now that you have a basic understanding of the Android application and the `AndroidManifest.xml` file, which describes its components, it's time to discuss how and where an Android application executes. To do that, we need to talk about the relationship between an Android application and its Linux and Dalvik VM runtime.

1.7. MAPPING APPLICATIONS TO PROCESSES

Android applications each run in a single Linux process. Android relies on Linux for process management, and the application itself runs in an instance of the Dalvik VM. The OS might need to unload, or even kill, an application from time to time to accommodate resource allocation demands. The system uses a hierarchy or sequence to select the victim during a resource shortage. In general, the system follows these rules:

- Visible, running activities have top priority.
- Visible, nonrunning activities are important, because they're recently paused and are likely to be resumed shortly.
- Running services are next in priority.
- The most likely candidates for termination are processes that are empty (loaded perhaps for performance-caching purposes) or processes that have dormant `Activity`s.

```
ps -a
```

The Linux environment is complete, including process management. You can launch and kill applications directly from the shell on the Android platform, but this is a

developer's debugging task, not something the average Android handset user is likely to carry out. It's nice to have this option for troubleshooting application issues. It's a relatively recent phenomenon to be able to touch the metal of a mobile phone in this way. For more in-depth exploration of the Linux foundations of Android, see [chapter 13](#).

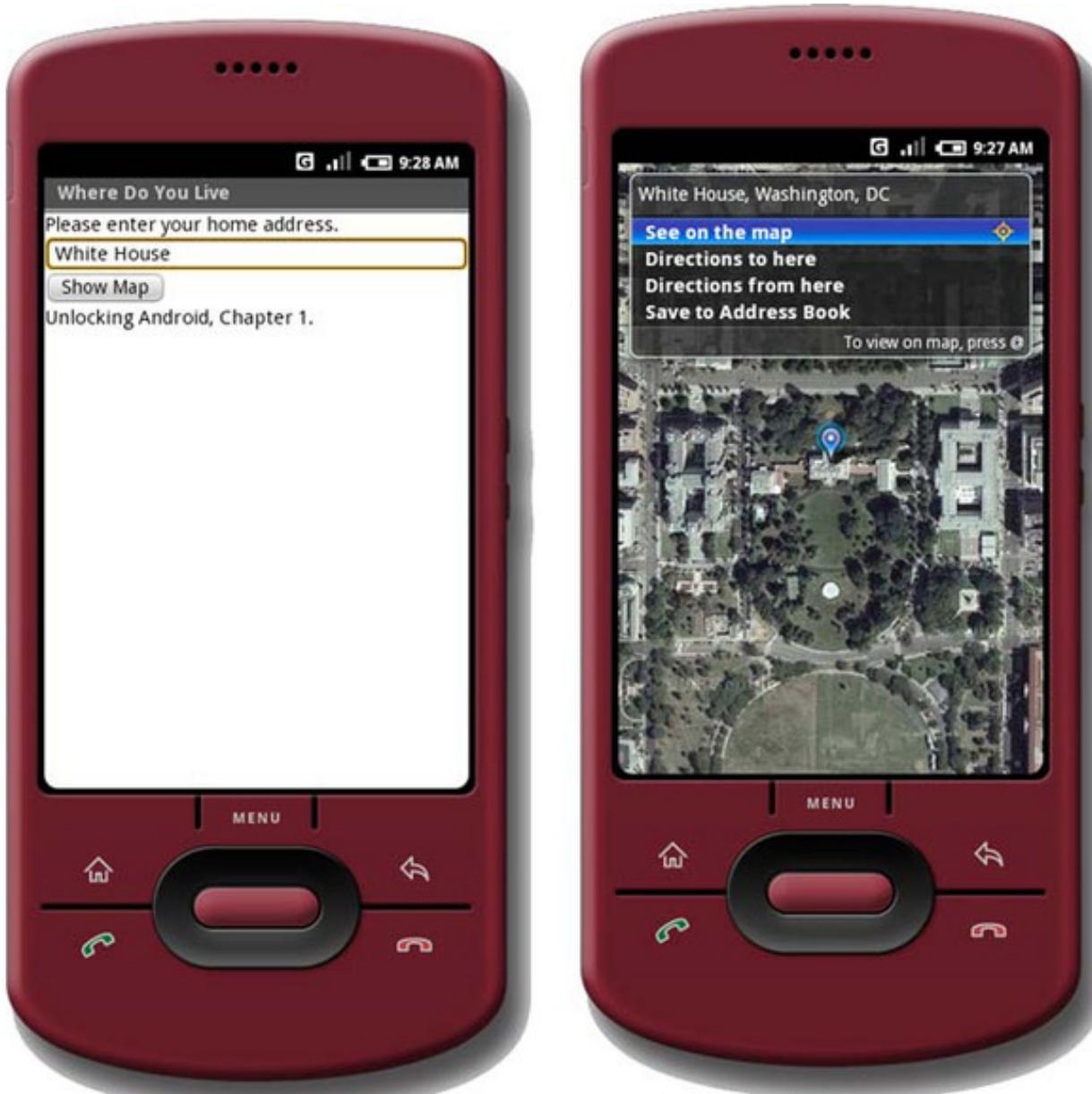


Let's apply some of what you've learned by building your first Android application.

1.8. CREATING AN ANDROID APPLICATION

Let's look at a simple Android application consisting of a single `Activity`, with one `View`. The `Activity` collects data (a street address) and creates an `Intent` to find this address. The `Intent` is ultimately dispatched to Google Maps. [Figure 1.7](#) is a screen shot of the application running on the emulator. The name of the application is Where Do You Live.

Figure 1.7. This Android application demonstrates a simple `Activity` and an `Intent`.



As we previously stated, the `AndroidManifest.xml` file contains the descriptors for the application components of the application. This application contains a single Activity named `WhereDoYouLive`. The application's `AndroidManifest.xml` file is shown in the following listing.

Listing 1.6. `AndroidManifest.xml` for the Where Do You Live application

```
<?xml version="1.0" encoding="utf-8"?>  
  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
  
    package="com.msi.manning.unlockingandroid">  
  
    <application android:icon="@drawable/icon">
```

```

<activity android:name=".AWhereDoYouLive"
    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>
<uses-permission android:name="android.permission.INTERNET" />
</manifest>

```

The sole Activity is implemented in the file AWhereDoYouLive.java, shown in the following listing.

Listing 1.7. Implementing the Android Activity in AWhereDoYouLive.java

```

package com.msi.manning.unlockingandroid;
// imports omitted for brevity
public class AWhereDoYouLive extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        final EditText addressfield =
            (EditText) findViewById(R.id.address);
        final Button button = (Button)
            findViewById(R.id.launchmap);
        button.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View view) {
                try {
                    String address = addressfield.getText().toString();
                    address = address.replace(' ', '+');
                    Intent geoIntent = new Intent
                        (android.content.Intent.ACTION_VIEW,
                        Uri.parse("geo:0,0?q=" + address));
                    startActivity(geoIntent);
                } catch (Exception e) {
                }
            }
        });
    }
}

```

1 Get address

2 Prepare Intent

In this example application, the `setContentView` method creates the primary UI, which is a layout defined in `main.xml` in the `/res/layout` directory. The `EditText` view collects information, which in this case is an address. The `EditText` view is a text box or edit box in generic programming parlance. The `findViewById` method connects the resource identified by `R.id.address` to an instance of the `EditText` class.

A `Button` object is connected to the `launchmap` UI element, again using the `findViewById` method. When this button is clicked, the application obtains the entered address by invoking the `getText` method of the associated `EditText` 1.

When the address has been retrieved from the UI, you need to create an `Intent` to find the entered address. The `Intent` has a `VIEW` action, and the data portion represents a geographic search query 2.

Finally, the application asks Android to perform the `Intent`, which ultimately results in the mapping application displaying the chosen address. The `startActivity` method is invoked, passing in the prepared `Intent`.

Resources are precompiled into a special class known as the `R` class, as shown in [listing 1.8](#). The final members of this class represent UI elements. You should never modify the `R.java` file manually; it's automatically built every time the underlying resources change. (We'll cover Android resources in greater depth in [chapter 3](#).)

Listing 1.8. R.java containing the R class, which has UI element identifiers

```
/* AUTO-GENERATED FILE. DO NOT MODIFY.

*
* This class was automatically generated by the
* aapt tool from the resource data it found. It
* should not be modified by hand.
*/
package com.msi.manning.unlockingandroid;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
```

```
}

public static final class id {
    public static final int address=0x7f050000;
    public static final int launchmap=0x7f050001;
}

public static final class layout {
    public static final int main=0x7f030000;
}

public static final class string {
    public static final int app_name=0x7f040000;
}

}
```

Figure 1.7 shows the sample application in action. Someone looked up the address of the White House; the result shows the White House pinpointed on the map.

The primary screen of this application is defined as a `LinearLayout` view, as shown in the following listing. It's a single layout containing one label, one text-entry element, and one button control.

Listing 1.9. Main.xml defining the UI elements for the 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:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Please enter your home address."
    />
<EditText
    android:id="@+id/address"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoText="true"
    />
<Button
    android:id="@+id/launchmap"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Show Map"
    />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Unlocking Android, Chapter 1."
    />
</LinearLayout>

```

1 ID assignment for EditText

2 ID assignment for Button

Note the use of the @ symbol in this resource's id attribute 1 and 2. This symbol causes the appropriate entries to be made in the R class via the automatically generated R.java file. These R class members are used in the calls to findViewById(), as shown in [listing 1.7](#), to tie the UI elements to an instance of the appropriate class.

A strings file and icon round out the resources in this simple application. The strings.xml file for this application is shown in the following listing. This file is used to localize string content.

Listing 1.10. strings.xml

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Where Do You Live</string>
</resources>

```

As you've seen, an Android application has a few moving pieces—though the components themselves are rather straightforward and easy to stitch together. As we progress through the book, we'll introduce additional sample applications step-by-step as we cover each of the major elements of Android development activities.

1.9. ANDROID 3.0 FOR TABLETS AND SMARTPHONES

Android 3.0 was originally introduced for tablets. But what makes the tablet different? It's the richer and more interactive application *user experience* that tablets provide. This user experience is driven by the tablet's form factor (larger screen), ease of handling, media-rich and graphical capabilities, content and application distribution support, computing power, and, as in the case of smartphones, connectivity, including offline support.

This new form factor opens the door to new application verticals such as eHealth, where ease of use and privacy issues are of primary importance, and content media distribution where content protection via DRM will play an important role.

The tablet form factor also introduces new challenges to Android developers—challenges related to UI design and development considerations not found when developing for smartphones. The larger form factor encourages touch interaction and navigation using one or both hands, and layout design that takes full advantage of landscape versus portrait. And because tablets are now part of the mobile platform family, application compatibility and portability across smartphones and tablets is an important consideration for mobile developers.

Android 3.0 isn't limited to tablets and applies to smartphones as well, but on a smaller scale. Everything in this chapter also applies to smartphones, once Android 3.0 is ported across the different platforms.

1.9.1. Why develop for Android tablets?

Mobile developers already have to deal with many different kinds of mobile platforms: iOS, mobile web, Android (and its different versions), BlackBerry, Windows Phone, Web OS, and so on. This can be overwhelming, so it's important to focus on the platforms that matter to you and your customers—in other words, the platforms with greater return on investment.

The tablet space is not only growing, but is expected to be *massive*. Driven by iOS and Android tablets, a recent 2011 Yankee Report puts total tablet device sales in the USA alone at \$7 billion.^[1] Tablets will play a major role in both the consumer and enterprise spaces. The opportunities for tablet application development seem endless.

²www.yankeegroup.com/ResearchDocument.do?id=55390

According to Gartner, 17.6 million tablets were sold in 2010, and it anticipates a significant increase with sales jumping to 69.5 million tablets in 2011. The firm's analysts anticipate in 2015 nearly 300 million devices could be sold.^[1]

³<http://mng.bz/680r>

Tablets will be a predominate mobile platform that must be considered by any developer who is serious about developing for mobile.

1.9.2. What's new in the Android 3.0 Honeycomb platform?

The new Android 3.0 platform provides all the elements for tablet application development. Android 3.0 introduces a number of UI enhancements that improve overall application usage experience on tablets. These include a new holographic theme, a new global notification bar, an application-specific action bar, a redesigned keyboard, and text selection with cut/paste capabilities. New connectivity features for Bluetooth and USB are provided, as well as updates to a number of the standard applications such as the browser, camera, and email. Because tablets are expected to play a major role in the Enterprise and businesses, new policy-management support has been introduced as well.

From the developer perspective, the changes introduced by Android 3.0 are extensive with additions and changes to many existing Java packages and three new Java packages:

- Animation (`android.animation`)
- Digital Rights Management (DRM, `android.drm`)
- High-performance 3D graphics (`android.renderscript`)

The changes to the other existing Java packages touch many aspects of the Android API layer, including the following:

- `Activitys` and `Fragments`
- The Action bar
- Drag and drop
- Custom notifications
- Loaders
- Bluetooth

This book will cover the major aspects of tablet development using Android 3.0, starting with `Activitys` and `Fragments`. Although we'll focus on tablets, note that Google TV is

Android 3.1-based, meaning that most of the content covered here is also applicable to Google TV.

1.10. SUMMARY

This chapter introduced the Android platform and briefly touched on market positioning, including what Android is up against in the rapidly changing and highly competitive mobile marketplace. In a few years, the Android SDK has been announced, released, and updated numerous times. And that's just the software. Major device manufacturers have now signed on to the Android platform and have brought capable devices to market, including a privately labeled device from Google itself. Add to that the patent wars unfolding between the major mobile players, and the stakes continue to rise—and Android's future continues to brighten.

In this chapter, we examined the Android stack and discussed its relationship with Linux and Java. With Linux at its core, Android is a formidable platform, especially for the mobile space where it's initially targeted. Although Android development is done in the Java programming language, the runtime is executed in the Dalvik VM, as an alternative to the Java VM from Oracle. Regardless of the VM, Java coding skills are an important aspect of Android development.

We also examined the Android SDK's `Intent` class. The `Intent` is what makes Android tick. It's responsible for how events flow and which code handles them. It provides a mechanism for delivering specific functionality to the platform, enabling third-party developers to deliver innovative solutions and products for Android. We introduced all the main application classes of `Activity`, `Service`, `ContentProvider`, and `BroadcastReceiver`, with a simple code snippet example for each. Each of these application classes use `Intents` in a slightly different manner, but the core facility of using `Intents` to control application behavior enables the innovative and flexible Android environment. `Intents` and their relationship with these application classes will be unpacked and unlocked as we progress through this book.

The `AndroidManifest.xml` descriptor file ties all the details together for an Android application. It includes all the information necessary for the application to run, what `Intents` it can handle, and what permissions the application requires. Throughout this book, the `AndroidManifest.xml` file will be a familiar companion as we add and explain new elements.

Finally, this chapter provided a taste of Android application development with a simple example tying a simple UI, an `Intent`, and Google Maps into one seamless and useful experience. This example is, of course, just scratching the surface of what Android can do. The next chapter takes a deeper look into the Android SDK so that you can learn more about the toolbox we'll use to unlock Android.

Chapter 2. Android's development environment

This chapter covers

- Introducing the Android SDK
- Exploring the development environment
- Building an Android application in Eclipse
- Debugging applications in the Android emulator

Building upon the foundational information presented in the first chapter, we pick up the pace by introducing the Android development environment used to construct the applications in the balance of the book. If you haven't installed the development tools, refer to [appendix A](#) for a step-by-step guide to downloading and installing the tools.

This chapter introduces the Android development tool chain and the software tools required to build Android applications, and serves as your hands-on guide to creating, testing, and even debugging applications. When you've completed this chapter, you'll be familiar with using Eclipse and the Android Development Tools (ADT) plug-in for Eclipse, navigating the Android SDK, running Android applications in the emulator, and stepping line-by-line through a sample application that you'll construct in this chapter: a simple tip calculator.

Android developers spend a significant amount of time working with the Android emulator to debug their applications. This chapter goes into detail about creating and building projects, defining Android virtual devices (emulators), setting up run configurations, and running and debugging applications on an instance of the Android emulator. If you've never constructed an Android application, please don't skip this chapter; mastering the basics demonstrated here will aide your learning throughout the rest of the book.

When embracing a new platform, the first task for a developer is gaining an understanding of the SDK and its components. Let's start by examining the core components of the Android SDK and then transition into using the SDK's tools to build and debug an application.

2.1. INTRODUCING THE ANDROID SDK

The Android SDK is a freely available download from the Android website. The first thing you should do before going any further in this chapter is make sure you have the Android SDK installed, along with Eclipse and the Android plug-in for Eclipse, also known as the *Android Development Tools*, or simply as the *ADT*. The Android SDK is required to build Android applications, and Eclipse is the preferred development environment for this book. You can download the Android SDK from <http://developer.android.com/sdk/index.html>.

Tip

The Android download page has instructions for installing the SDK, or you can refer to [appendix A](#) of this book for detailed information on installing the required development tools.

As in any development environment, becoming familiar with the class structures is helpful, so having the documentation at hand as a reference is a good idea. The Android SDK includes HTML-based documentation, which primarily consists of Javadoc-formatted pages that describe the available packages and classes. The Android SDK documentation is in the /doc directory under your SDK installation. Because of the rapidly changing nature of this platform, you might want to keep an eye out for any changes to the SDK. The most up-to-date Android SDK documentation is available at <http://developer.android.com/reference/packages.html>.

Android's Java environment can be broken down into a handful of key sections. When you understand the contents in each of these sections, the Javadoc reference material that ships with the SDK becomes a real tool and not just a pile of seemingly unrelated material. You might recall that Android isn't a strictly Java ME software environment, but there's some commonality between the Android platforms and other Java development platforms. The next few sections review some of the Java packages (core and optional) in the Android SDK and where you can use them. The remaining chapters provide a deeper look into using many of these programming topics.

2.1.1. Core Android packages

If you've ever developed in Java, you'll recognize many familiar Java packages for core functionality. These packages provide basic computational support for things such as string management, input/output controls, math, and more. The following list contains some of the Java packages included in the Android SDK:

- `java.lang`—Core Java language classes
- `java.io`—Input/output capabilities
- `java.net`—Network connections
- `java.text`—Text-handling utilities
- `java.math`—Math and number-manipulation classes
- `javax.net`—Network classes
- `javax.security`—Security-related classes
- `javax.xml`—DOM-based XML classes
- `org.apache.*`—HTTP-related classes
- `org.xml`—SAX-based XML classes

Additional Java classes are also included. Generally speaking, this book won't focus much on the core Java packages listed here, because our primary concern is Android development. With that in mind, let's look at the Android-specific functionality found in the Android SDK.

Android-specific packages are easy to identify because they start with `android` in the package name. Some of the more important packages are as follows:

- `android.app`—Android application model access
- `android.bluetooth`—Android's Bluetooth functionality
- `android.content`—Accessing and publishing data in Android
- `android.net`—Contains the `Uri` class, used for accessing content
- `android.gesture`—Creating, recognizing, loading, and saving gestures
- `android.graphics`—Graphics primitives
- `android.location`—Location-based services (such as GPS)
- `android.opengl`—OpenGL classes
- `android.os`—System-level access to the Android environment
- `android.provider`—`ContentProvider`-related classes
- `android.telephony`—Telephony capability access, including support for both Code Division Multiple Access (CDMA) and Global System for Mobile communication (GSM) devices
- `android.text`—Text layout
- `android.util`—Collection of utilities for logging and text manipulation, including XML
- `android.view`—UI elements
- `android.webkit`—Browser functionality
- `android.widget`—More UI elements

Some of these packages are core to Android application development, including `android.app`, `android.view`, and `android.content`. Other packages are used to varying degrees, depending on the type of applications that you're constructing.

2.1.2. Optional packages

Not every Android device has the same hardware and mobile connectivity capabilities, so you can consider some elements of the Android SDK as optional. Some devices support these features, and others don't. It's important that an application degrade gracefully if a feature isn't available on a specific handset. Java packages that you should pay special attention to include those that rely on specific, underlying hardware and network characteristics, such as location-based services (including GPS) and wireless technologies such as Bluetooth and Wi-Fi (802.11).

This quick introduction to the Android SDK's programming interfaces is just that—quick and at-a-glance. Upcoming chapters go into the class libraries in further detail, exercising specific classes as you learn about various topics such as UIs, graphics, location-based services, telephony, and more. For now, the focus is on the tools required to compile and run (or build) Android applications.

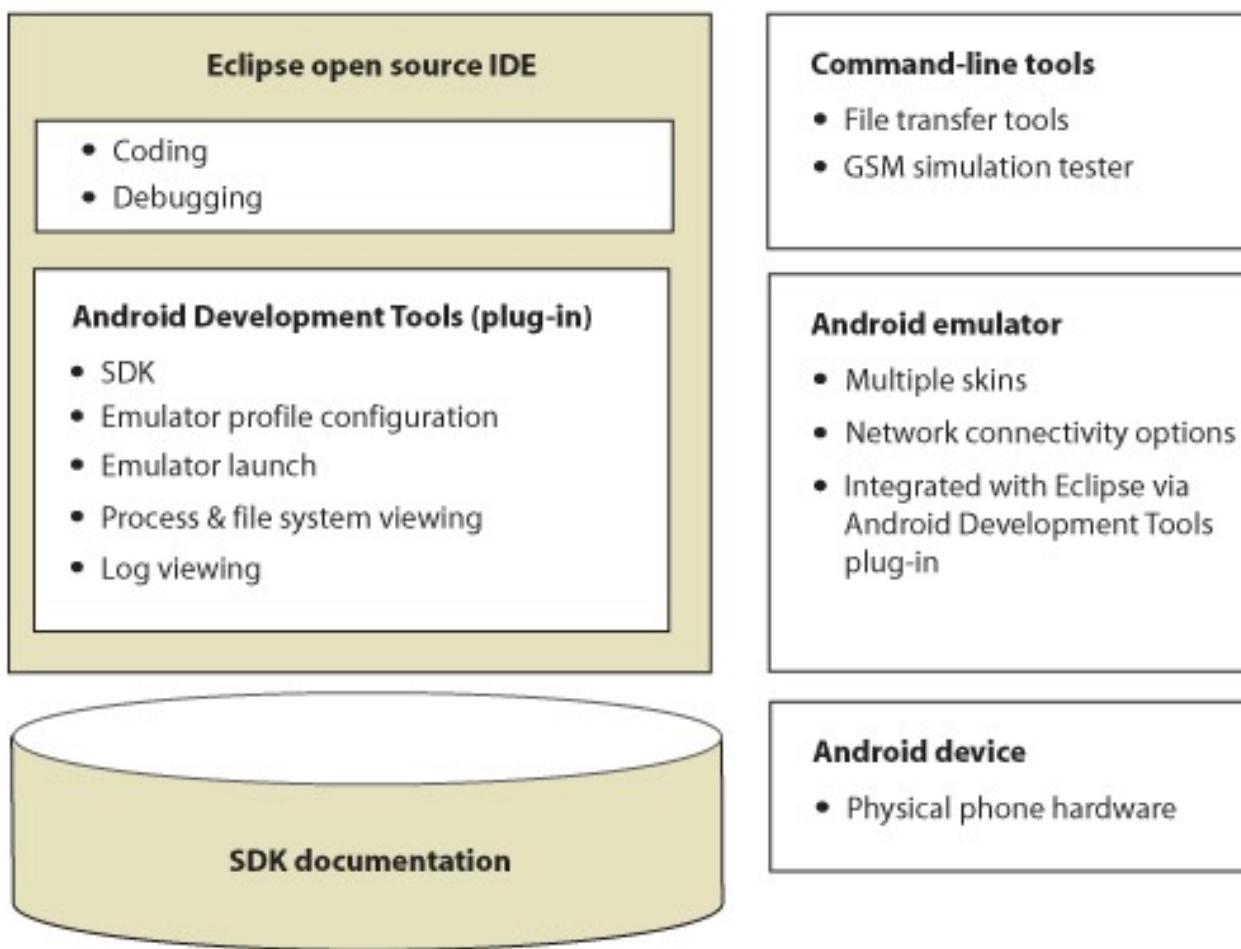
Before you build an Android application, let's examine how the Android SDK and its components fit into the Eclipse environment.

2.2. EXPLORING THE DEVELOPMENT ENVIRONMENT

After you install the Android SDK and the ADT plug-in for Eclipse, you're ready to explore the development environment. [Figure 2.1](#) depicts the typical Android development environment, including both real hardware and the useful Android emulator. Although Eclipse isn't the exclusive tool required for Android development, it can play a big role in Android development, not only because it provides a rich Java compilation and debugging environment, but also because with the ADT plug-in, you can manage and control virtually all aspects of testing your Android applications directly from the Eclipse IDE.

Figure 2.1. The development environment for building Android applications, including the popular open source Eclipse IDE

Development environment (laptop)



The following list describes key features of the Eclipse environment as it pertains to Android application development:

- A rich Java development environment, including Java source compilation, class auto-completion, and integrated Javadoc
- Source-level debugging
- AVD management and launch
- The Dalvik Debug Monitor Server (DDMS)
- Thread and heap views
- Emulator filesystem management
- Data and voice network control
- Emulator control
- System and application logging

Eclipse supports the concept of *perspectives*, where the layout of the screen has a set of related windows and tools. The windows and tools included in an Eclipse perspective are known as *views*. When developing Android applications, there are two Eclipse perspectives of primary interest: the Java perspective and the DDMS

perspective. Beyond those two, the Debug perspective is also available and useful when you're debugging an Android application; we'll talk about the Debug perspective in [section 2.5](#). To switch between the available perspectives in Eclipse, use the Open Perspective menu, under the Window menu in the Eclipse IDE.

Let's examine the features of the Java and DDMS perspectives and how you can leverage them for Android development.

2.2.1. The Java perspective

The Java perspective is where you'll spend most of your time while developing Android applications. The Java perspective boasts a number of convenient views for assisting in the development process. The Package Explorer view allows you to see the Java projects in your Eclipse workspace. [Figure 2.2](#) shows the Package Explorer listing some of the sample projects for this book.

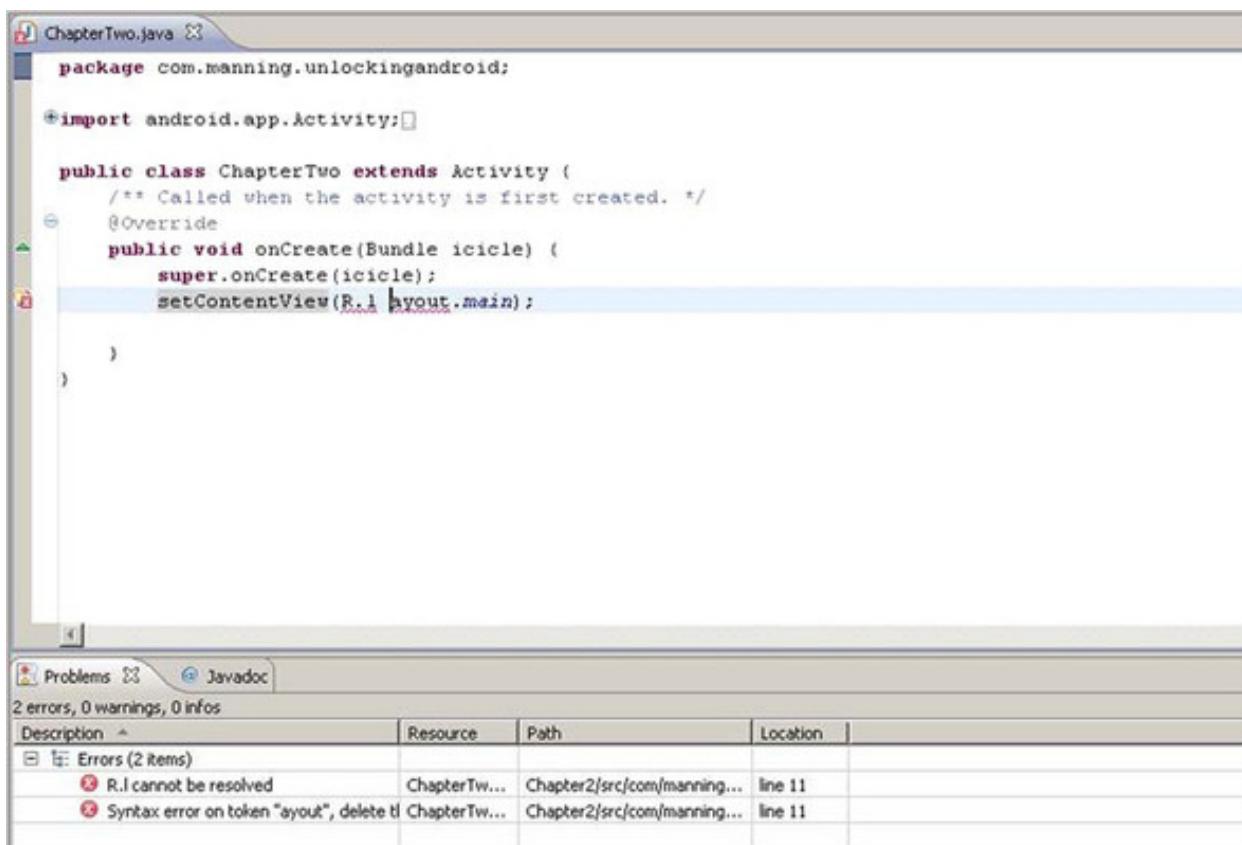
Figure 2.2. The Package Explorer allows you to browse the elements of your Android projects.



The Java perspective is where you'll edit your Java source code. Every time you save your source file, it's automatically compiled by Eclipse's Java development tools (JDT) in the background. You don't need to worry about the specifics of the JDT; the important thing to know is that it's functioning in the background to make your Java

experience as seamless and painless as possible. If there's an error in your source code, the details will show up in the Problems view of the Java perspective. [Figure 2.3](#) has an intentional error in the source code to demonstrate the Problems view. You can also put your mouse over the red x to the left of the line containing the error for a tool-tip explanation of the problem.

Figure 2.3. The Problems view shows any errors in your source code.



One powerful feature of the Java perspective in Eclipse is the integration between the source code and the Javadoc view. The Javadoc view updates automatically to provide any available documentation about a currently selected Java class or method, as shown in [figure 2.4](#). In this figure, the Javadoc view displays information about the `Activity` class.

Figure 2.4. The Javadoc view provides context-sensitive documentation, in this case for the `Activity` class.

The screenshot shows the Eclipse IDE interface. The top window displays the Java code for `ChapterTwo.java`:

```
package com.manning.unlockingandroid;
import android.app.Activity;
public class ChapterTwo extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
    }
}
```

Below the code editor is the Javadoc for `android.app.Activity`:

An activity is a single, focused thing that the user can do. Almost all activities interact with the user, so the Activity class takes care of creating a window for you in which you can place your UI windows, they can also be used in other ways: as floating windows (via a theme with `windowId` set) or embedded inside of another activity (using `ActivityGroup`). There are two method

- `onCreate(Bundle)` is where you initialize your activity. Most importantly, here you will usually call `setContentView(int)` with a layout resource defining your UI, and using `findViewById(int)`
- `onPause()` is where you deal with the user leaving your activity. Most importantly, any changes made by the user should at this point be committed (usually to the `ContentProvider` hold

To be of use with `Context.startActivity()`, all activity classes must have a corresponding `<activity>` declaration in their package's `AndroidManifest.xml`.

The Activity class is an important part of an application's overall lifecycle.

Topics covered here:

1. [Activity Lifecycle](#)
2. [Configuration Changes](#)
3. [Starting Activities and Getting Results](#)
4. [Saving Persistent State](#)
5. [Permissions](#)
6. [Process Lifecycle](#)

Activity Lifecycle

Activities in the system are managed as an *activity stack*. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always remains in the stack, even if it is not the current one.

Tip

This chapter scratches the surface in introducing the powerful Eclipse environment. To learn more about Eclipse, you might consider reading *Eclipse in Action: A Guide for Java Developers*, by David Gallardo, Ed Burnette, and Robert McGovern, published by Manning and available online at www.manning.com/gallardo.

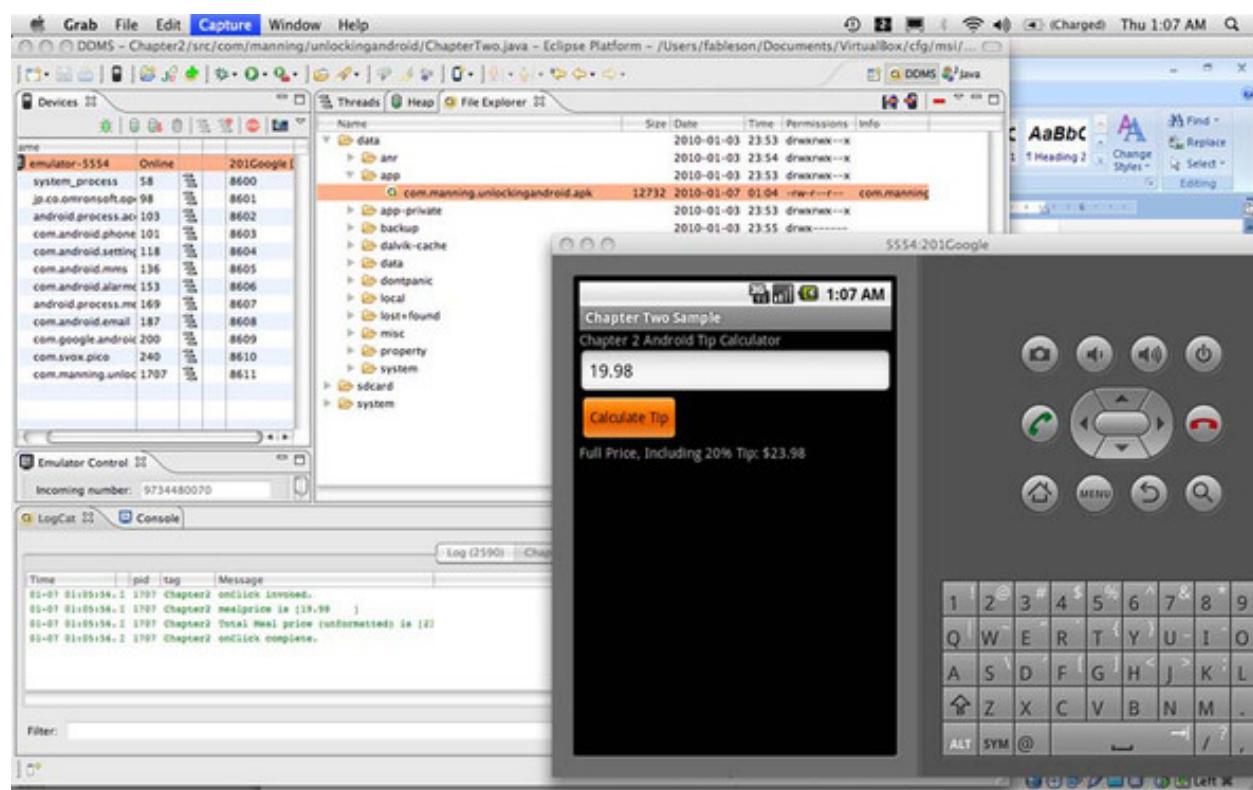
It's easy to get the views in the current perspective into a layout that isn't what you really want. If this occurs, you have a couple of choices to restore the perspective to a more useful state. You can use the Show View menu under the Window menu to display a specific view or you can select the Reset Perspective menu to restore the perspective to its default settings.

In addition to the JDT, which compiles Java source files, the ADT automatically compiles Android-specific files such as layout and resource files. You'll learn more about the underlying tools later in this chapter and again in [chapter 3](#), but now it's time to have a look at the Android-specific perspective in the DDMS.

2.2.2. The DDMS perspective

The DDMS perspective provides a dashboard-like view into the heart of a running Android device or, in this example, a running Android emulator. [Figure 2.5](#) shows the emulator running the [chapter 2](#) sample application.

Figure 2.5. DDMS perspective with an application running in the Android emulator



We'll walk through the details of the application, including how to build the application and how to start it running in the Android emulator, but first let's see what there is to learn from the DDMS with regard to our discussion about the tools available for Android development.

The Devices view in [figure 2.5](#) shows a single emulator session, titled `emulator-tcp-5554`. The title indicates that there's a connection to the Android emulator at TCP/IP port 5554. Within this emulator session, five processes are running. The one of interest to us is `com.manning.unlockingandroid`, which has the process ID 1707.



Tip

Unless you're testing a peer-to-peer application, you'll typically have only a single Android emulator session running at a time, although it is possible to have multiple instances of the Android emulator running concurrently on a single development machine. You might also have a physical Android device connected to your development machine—the DDMS interface is the same.



Logging is an essential tool in software development, which brings us to the LogCat view of the DDMS perspective. This view provides a glimpse at system and application logging taking place in the Android emulator. In [figure 2.5](#), a filter has been set up for looking at entries with a `tag` value of `Chapter2`. Using a filter on the LogCat is a helpful practice, because it can reduce the noise of all the logging entries and let you focus on your own application's entries. In this case, four entries in the list match our filter criteria. We'll look at the source code soon to see how you get your messages into the log. Note that these log entries have a column showing the process ID, or PID, of the application contributing the log entry. As expected, the PID for our log entries is 616, matching our running application instance in the emulator.

The File Explorer view is shown in the upper right of [figure 2.5](#). User applications—the ones you and I write—are deployed with a file extension of `.apk` and stored in the `/data/app` directory of the Android device. The File Explorer view also permits filesystem operations such as copying files to and from the Android emulator, as well as removing files from the emulator's filesystem. [Figure 2.6](#) shows the process of deleting a user application from the `/data/app` directory.

Figure 2.6. Delete applications from the emulator by highlighting the application file and clicking the Delete button.

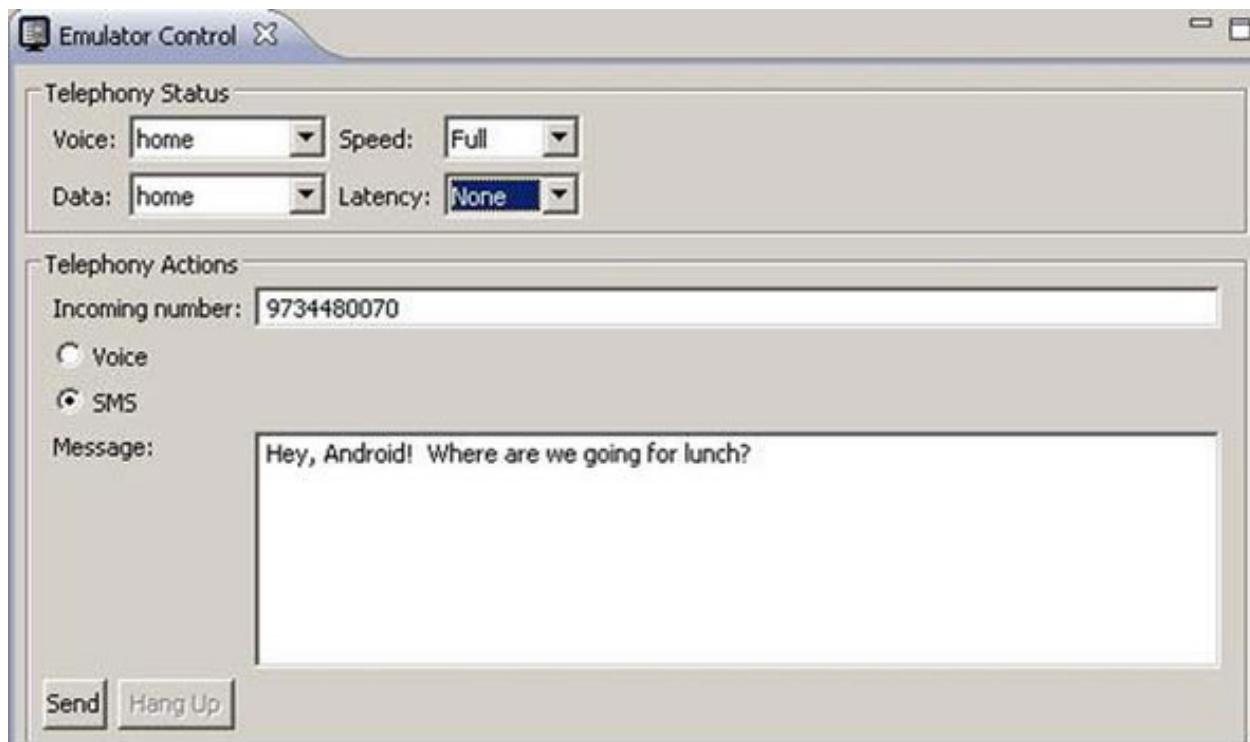
The screenshot shows the Android DDMS perspective with the 'File Explorer' tab selected. The left pane displays a tree view of the application's directory structure under the package com.manning.unlockingandroid. The right pane is a table view showing detailed information for each file and directory entry.

Name	Size	Date	Time	Permissions	Info	Action
data		2010-01-03	23:53	drwxrwx--x		
anr		2010-01-03	23:54	drwxrwx--x		
app		2010-01-03	23:53	drwxrwx--x		
com.manning.unlockingandroid.apk	12732	2010-01-07	01:04	-rw-r--r--	com.manning.unlocking...	Delete the selection
app-private		2010-01-03	23:53	drwxrwx--x		
backup		2010-01-03	23:55	drwx-----		
dalvik-cache		2010-01-03	23:53	drwxrwx--x		
data		2010-01-03	23:53	drwxrwx--x		
dontpanic		2010-01-03	23:53	drwxr-x---		
local		2010-01-03	23:53	drwxrwx--x		
lost+found		2010-01-03	23:53	drwxrwx---		
misc		2010-01-03	23:53	drwxrwx--t		
property		2010-01-03	23:53	drwx-----		
system		2010-01-03	23:54	drwxrwxr-x		
sdcard		1969-12-31	19:00	d---rwxr-x		
system		2009-11-23	15:24	drwxr-xr-x		

Obviously, being able to casually browse the filesystem of your mobile phone is a great convenience. This feature is nice to have for mobile development, where you're often relying on cryptic pop-up messages to help you along in the application development and debugging process. With easy access to the filesystem, you can work with files and readily copy them to and from your development computer platform as necessary.

In addition to exploring a running application, the DDMS perspective provides tools for controlling the emulated environment. For example, the Emulator Control view lets you test connectivity characteristics for both voice and data networks, such as simulating a phone call or receiving an incoming Short Message Service (SMS). [Figure 2.7](#) demonstrates sending an SMS message to the Android emulator.

Figure 2.7. Sending a test SMS to the Android emulator



The DDMS provides a lot of visibility into, and control over, the Android emulator, and is a handy tool for evaluating your Android applications. Before we move on to building and testing Android applications, it's helpful to understand what's happening behind the scenes and what's enabling the functionality of the DDMS.

2.2.3. Command-line tools

The Android SDK ships with a collection of command-line tools, which are located in the tools subdirectory of your Android SDK installation. Eclipse and the ADT provide a great deal of control over the Android development environment, but sometimes it's nice to exercise greater control, particularly when considering the power and convenience that scripting can bring to a development platform. Next, we're going to explore two of the command-line tools found in the Android SDK.

Tip

It's a good idea to add the tools directory to your search path. For example, if your Android SDK is installed to c:\software\google\androidsdk, you can add the Android SDK to your path by performing the following operation in a command window on your Windows computer:

```
set path=%path%;c:\software\google\androidsdk\tools;
```

Or use the following command for Mac OS X and Linux:

```
export PATH=$PATH:/path_to_Android_SDK_directory/tools
```

Android Asset Packaging Tool

You might be wondering just how files such as the layout file main.xml get processed and exactly where the R.java file comes from. Who zips up the application file for you into the apk file? Well, you might have already guessed the answer from the heading of this section—it's the *Android Asset Packaging Tool*, or as it's called from the command line, `aapt`. This versatile tool combines the functionality of pkzip or jar along with an Android-specific resource compiler. Depending on the command-line options you provide to it, `aapt` wears a number of hats and assists with your design-time Android development tasks. To learn the functionality available in `aapt`, run it from the command line with no arguments. A detailed usage message is written to the screen.

Whereas `aapt` helps with design-time tasks, another tool, the *Android Debug Bridge*, assists you at runtime to interact with the Android emulator.

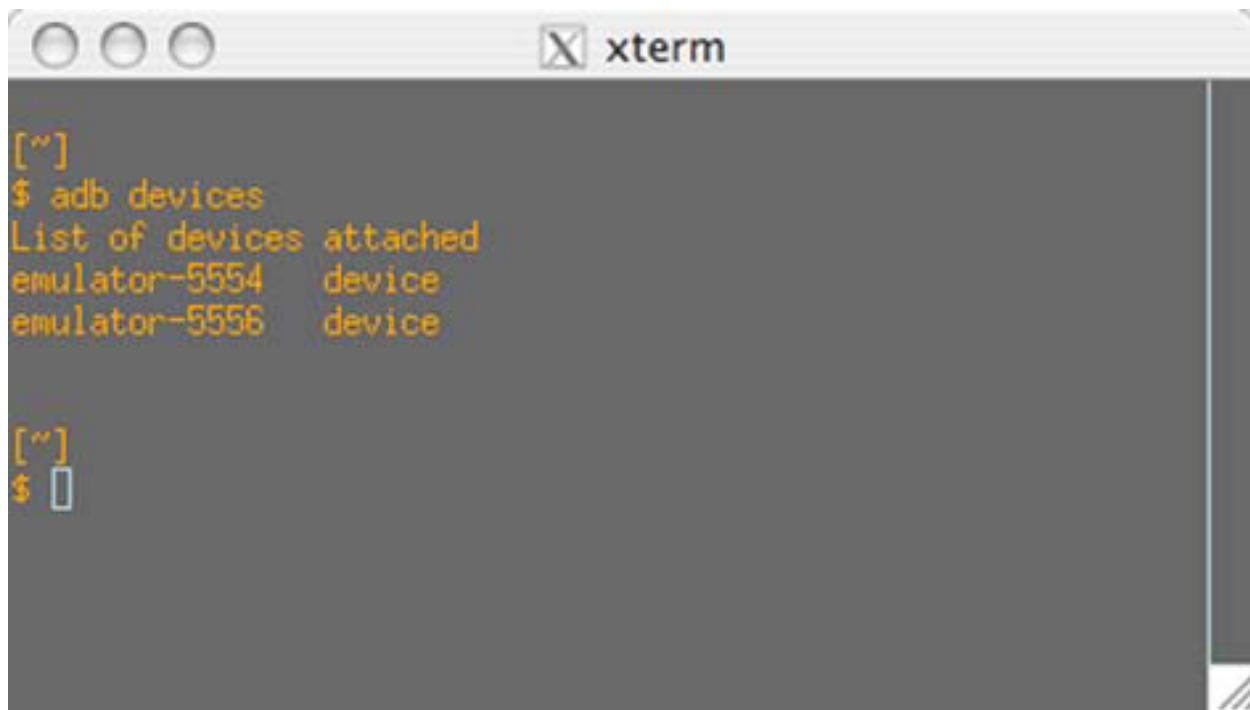
Android Debug Bridge

The *Android Debug Bridge* (`adb`) utility permits you to interact with the Android emulator directly from the command line or script. Have you ever wished you could navigate the filesystem on your smartphone? Now you can with `adb`! It works as a client/server TCP-based application. Although a couple of background processes run on the development machine and the emulator to enable your functionality, the important thing to understand is that when you run `adb`, you get access to a running instance of the Android emulator. Here are a couple of examples of using `adb`. First, let's look to see if you have any available Android emulator sessions running:

```
adb devices<return>
```

This command returns a list of available Android emulators; [figure 2.8](#) demonstrates `adb` locating two running emulator sessions.

Figure 2.8. The `adb` tool provides interaction at runtime with the Android emulator.



The screenshot shows an xterm window with the title "xterm". The terminal output is as follows:

```
[~] $ adb devices
List of devices attached
emulator-5554    device
emulator-5556    device
```

[~] \$

Let's connect to the first Android emulator session and see if your application is installed. You connect to a device or emulator with the syntax `adb shell`. You would connect this way if you had a single Android emulator session active, but because two emulators are running, you need to specify the serial number, or *identifier*, to connect to the appropriate session:

```
adb -s "serialnumber" shell
```

Figure 2.9 shows off the Android filesystem and demonstrates looking for a specific installed application, namely the `chapter2` sample application, which you'll build in [section 2.3](#).

Figure 2.9. Using the `shell` command of the `adb`, you can browse Android's filesystem.

```
[~]
$ adb devices
List of devices attached
emulator-5554    device
emulator-5556    device

[~]
$ adb -s emulator-5556 shell
#
# cd /data/app
#
# ls -l
-rw-r--r-- system   system      12732 2010-01-07 01:57 com.manning.unlockingandroid.apk
#
```

Using the shell can be handy when you want to remove a specific file from the emulator's filesystem, kill a process, or generally interact with the operating environment of the Android emulator. If you download an application from the internet, for example, you can use the `adb` command to install the application:

```
adb [-s serialnumber] shell install someapplication.apk
```

This command installs the application named `someapplication` to the Android emulator. The file is copied to the `/data/app` directory and is accessible from the Android application launcher. Similarly, if you want to remove an application, you can run `adb` to remove an application from the Android emulator. If you want to remove the `com.manning.unlockingandroid.apk` sample application from a running emulator's filesystem, for example, you can execute the following command from a terminal or Windows command window:

```
adb shell rm /data/app/com.manning.unlockingandroid.apk
```

You certainly don't need to master the command-line tools in the Android SDK to develop applications in Android, but understanding what's available and where to look for capabilities is a good skill to have in your toolbox. If you need assistance with either the `aapt` or `adb` command, enter the command at the terminal, and a fairly verbose usage/help page is displayed. You can find additional information about the tools in the Android SDK documentation.

Tip

The Android filesystem is a Linux filesystem. Though the `adb shell` command doesn't provide a rich shell programming environment, as you find on a Linux or Mac OS X system, basic commands such as `ls`, `ps`, `kill`, and `rm` are available. If you're new to Linux, you might benefit from learning some basic shell commands.

Telnet

One other tool you'll want to make sure you're familiar with is *telnet*. Telnet allows you to connect to a remote system with a character-based UI. In this case, the remote system you connect to is the Android emulator's console. You can connect to it with the following command:

```
telnet localhost 5554
```

In this case, `localhost` represents your local development computer where the Android emulator has been started, because the Android emulator relies on your computer's loopback IP address of `127.0.0.1`. Why port `5554`? Recall that when you employed `adb` to find running emulator instances, the output of that command included a name with a number at the end. The first Android emulator can generally be found at IP port `5554`.

Note

In early versions of the Android SDK, the emulator ran at port `5555` and the Android console—where you could connect via Telnet—ran at `5554`, or one number less than the number shown in DDMS. If you're having difficulty identifying which port number to connect on, be sure to run `netstat` on your development machine to assist in finding the port number. Note that a physical device listens at port `5037`.

Using a telnet connection to the emulator provides a command-line means for configuring the emulator while it's running and for testing telephony features such as calls and text messages.

So far you've learned about the Eclipse environment and some of the command-line elements of the Android tool chain. At this point, it's time to create your own Android application to exercise this development environment.

2.3. BUILDING AN ANDROID APPLICATION IN ECLIPSE

Eclipse provides a comprehensive environment for Android developers to create applications. In this section, we'll demonstrate how to build a basic Android application, step by step. You'll learn how to define a simple UI, provide code logic to support it, and create the deployment file used by all Android applications: `AndroidManifest.xml`. The goal in this section is to get a simple application under your belt. We'll leave more complex applications for later chapters; our focus is on exercising the development tools and providing a concise yet complete reference.

Building an Android application isn't much different from creating other types of Java applications in the Eclipse IDE. It all starts with choosing `File > New` and selecting an Android application as the build target.

Like many development environments, Eclipse provides a wizard interface to ease the task of creating a new application. You'll use the Android Project Wizard to get off to a quick start in building an Android application.

2.3.1. The Android Project Wizard

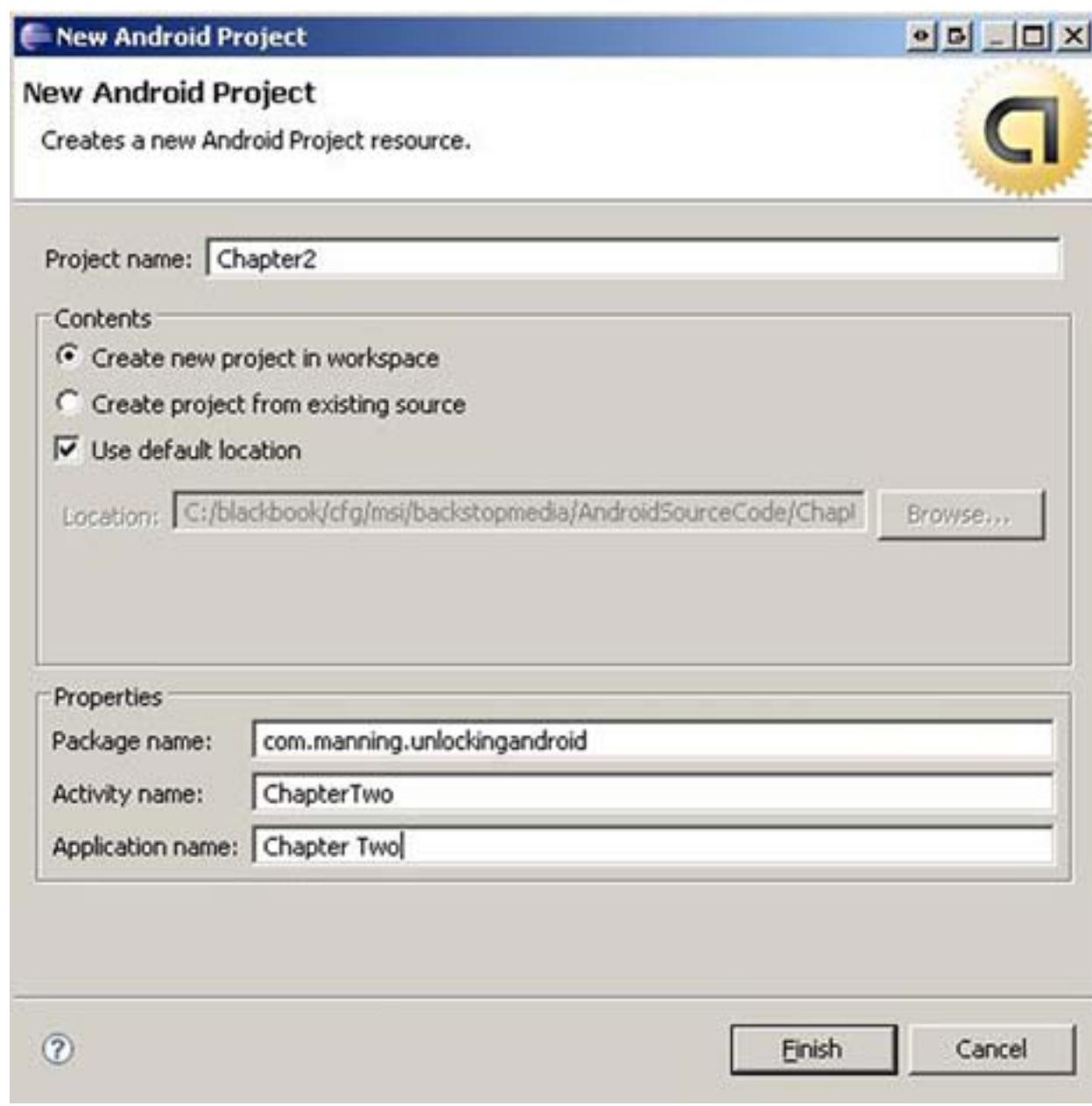
The most straightforward manner to create an Android application is to use the Android Project Wizard, which is part of the ADT plug-in. The wizard provides a simple means to define the Eclipse project name and location, the `Activity` name corresponding to the main UI class, and a name for the application. Also of importance is the Java package name under which the application is created. After you create an application, it's easy to add new classes to the project.

Note

In this example, you'll create a brand-new project in the Eclipse workspace. You can use this same wizard to import source code from another developer, such as the sample code for this book. Note also that the specific screens have changed over time as the Android tools mature. If you're following along and have a question about this chapter, be sure to post a question on the Manning Author forum for this book, available online at <http://manning.com/ableson3>.

Figure 2.10 demonstrates the creation of a new project named `Chapter2` using the wizard.

Figure 2.10. Using the Android Project Wizard, it's easy to create an empty Android application, ready for customization.



Tip

You'll want the package name of your applications to be unique from one application to the next.

Click Finish to create your sample application. At this point, the application compiles and is capable of running on the emulator—no further development steps are required. Of course, what fun would an empty project be? Let's flesh out this sample application and create an Android tip calculator.

2.3.2. Android sample application code

The Android Project Wizard takes care of a number of important elements in the Android application structure, including the Java source files, the default resource files, and the `AndroidManifest.xml` file. Looking at the Package Explorer view in Eclipse, you can see all the elements of this application. Here's a quick description of the elements included in the sample application:

- The `src` folder contains two Java source files automatically created by the wizard.
- `ChapterTwo.java` contains the main `Activity` for the application. You'll modify this file to add the sample application's tip calculator functionality.
- `R.java` contains identifiers for each of the UI resource elements in the application. Never modify this file directly. It automatically regenerates every time a resource is modified; any manual changes you make will be lost the next time the application is built.
- `Android.jar` contains the Android runtime Java classes. This reference to the `android.jar` file found in the Android SDK ensures that the Android runtime classes are accessible to your application.
- The `res` folder contains all the Android resource folders, including these:
 - `Drawables` contains image files such as bitmaps and icons. The wizard provides a default Android icon named `icon.png`.
 - `Layout` contains an XML file called `main.xml`. This file contains the UI elements for the primary view of your `Activity`. In this example, you'll modify this file but you won't make any significant or special changes—just enough to accomplish the meager UI goals for your tip calculator. We cover UI elements, including `Views`, in detail in [chapter 3](#). It's not uncommon for an Android application to have multiple XML files in the Layout section of the resources.
 - `Values` contains the `strings.xml` file. This file is used for localizing string values, such as the application name and other strings used by your application.

`AndroidManifest.xml` contains the deployment information for this project. Although `AndroidManifest.xml` files can become somewhat complex, this chapter's manifest file can run without modification because no special permissions are required. We'll visit `AndroidManifest.xml` a number of times throughout the book as we discuss new features.

Now that you know what's in the project, let's review how you're going to modify the application. Your goal with the Android tip calculator is to permit your user to enter the price of a meal and then tap a button to calculate the total cost of the meal, tip included. To accomplish this, you need to modify two files: ChapterTwo.java and the UI layout file, main.xml. Let's start with the UI changes by adding a few new elements to the primary view, as shown in the next listing.

Listing 2.1. main.xml, which contains UI elements

```
<?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:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Chapter 2 Android Tip Calculator"
        />

    <EditText
        android:id="@+id/mealprice"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:autoText="true"
        />

    <Button
        android:id="@+id/calculate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Calculate Tip"
        />

    <TextView
```

```
    android:id="@+id/answer"

    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=""

/>

</LinearLayout>
```

The layout for this application is straightforward. The overall layout is a vertical, linear layout with only four elements; all the UI controls, or *widgets*, are going to be in a vertical arrangement. A number of layouts are available for Android UI design, which we'll discuss in greater detail in [chapter 3](#).

A static `TextView` displays the title of the application. An `EditText` collects the price of the meal for this tip calculator application. The `EditText` element has an attribute of type `android:id`, with a value of `mealprice`. When a UI element contains the `android:id` attribute, it permits you to manipulate this element from your code. When the project is built, each element defined in the layout file containing the `android:id` attribute receives a corresponding identifier in the automatically generated `R.java` class file. This identifying value is used in the `findViewById` method, shown in [listing 2.2](#). If a UI element is static, such as the `TextView`, and doesn't need to be set or read from our application code, the `android:id` attribute isn't required.

A button named `calculate` is added to the view. Note that this element also has an `android:id` attribute because you need to capture click events from this UI element. A `TextView` named `answer` is provided for displaying the total cost, including tip. Again, this element has an `id` because you'll need to update it during runtime.

When you save the file `main.xml`, it's processed by the ADT plug-in, compiling the resources and generating an updated `R.java` file. Try it for yourself. Modify one of the `id` values in the `main.xml` file, save the file, and open `R.java` to have a look at the constants generated there. Remember not to modify the `R.java` file directly, because if you do, all your changes will be lost! If you conduct this experiment, be sure to change the values back as they're shown in [listing 2.1](#) to make sure the rest of the project will compile as it should. Provided you haven't introduced any syntactical errors into your `main.xml` file, your UI file is complete.

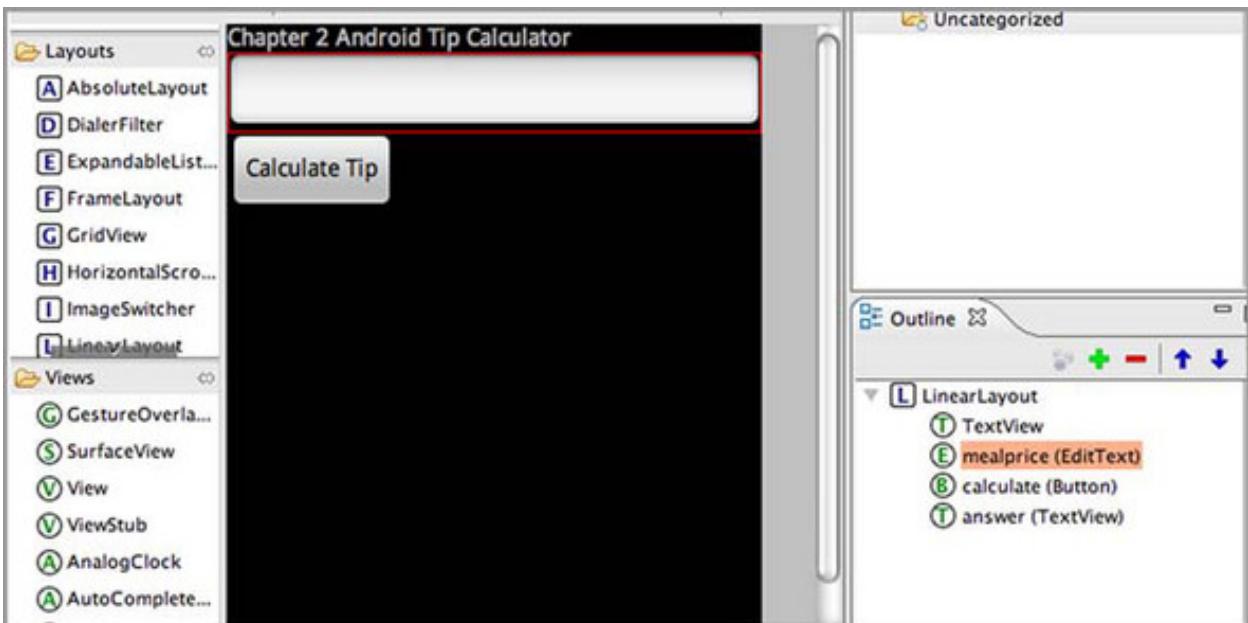
Note

This example is simple, so we jumped right into the XML file to define the UI elements. The ADT also contains an increasingly sophisticated GUI layout tool. With each release of the ADT, these tools have become more and more usable; early versions were, well, early.



Double-click the main.xml file to launch the layout in a graphical form. At the bottom of the file you can switch between the Layout view and the XML view. [Figure 2.11](#) shows the Layout tool.

Figure 2.11. Using the GUI Layout tool provided in the ADT to define the user interface elements of your application



It's time to turn our attention to the file ChapterTwo.java to implement the tip calculator functionality. ChapterTwo.java is shown in the following listing. We've omitted some imports for brevity. You can download the complete source code from the Manning website at <http://manning.com/ableson3>.

Listing 2.2. ChapterTwo.java: implements the tip calculator logic

```
package com.manning.unlockingandroid;
import com.manning.unlockingandroid.R;
import android.app.Activity;
import java.text.NumberFormat;
import android.util.Log;
// some imports omitted
public class ChapterTwo extends Activity {
    public static final String tag = "Chapter2";
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        final EditText mealpricefield =
            (EditText) findViewById(R.id.mealprice); 1 Reference  
EditText for  
mealprice
        final TextView answerfield =
            (TextView) findViewById(R.id.answer);
        final Button button = (Button) findViewById(R.id.calculate);
        button.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                try {
                    Log.i(tag,"onClick invoked.");
                    // grab the meal price from the UI
                    String mealprice =
                        mealpricefield.getText().toString(); 2 Log entry
                    Log.i(tag,"mealprice is [" + mealprice + "]");
                    String answer = "";
```

← 3 Get meal price

```

        // check to see if the meal price includes a "$"
        if (mealprice.indexOf("$") == -1) {
            mealprice = "$" + mealprice;
        }
        float fmp = 0.0F;
        // get currency formatter
        NumberFormat nf =
            java.text.NumberFormat.getCurrencyInstance();
        // grab the input meal price
        fmp = nf.parse(mealprice).floatValue();
        // let's give a nice tip -> 20%
        fmp *= 1.2;
        Log.i(tag,"Total Meal Price (unformatted) is ["
+ fmp + "]");
        // format our result
        answer = "Full Price, Including 20% Tip: "
+ nf.format(fmp);
        answerfield.setText(answer);
        Log.i(tag,"onClick complete.");
    } catch (java.text.ParseException pe) {
        Log.i(tag,"Parse exception caught");
        answerfield.setText("Failed to parse amount?");
    } catch (Exception e) {
        Log.e(tag,"Failed to Calculate Tip:" + e.getMessage());
        e.printStackTrace();
        answerfield.setText(e.getMessage());
    }
}
);
}
}

```

4 Display full price, including tip

5 Catch parse error

Let's examine this sample application. Like all but the most trivial Java applications, this class contains a statement identifying which package it belongs to: `com.manning.unlockingandroid`. This line containing the package name was generated by the Project Wizard.

You import the `com.manning.unlockingandroid.R` class to gain access to the definitions used by the UI. This step isn't required, because the `R` class is part of the same application package, but it's helpful to include this import because it makes your code easier to follow. Newcomers to Android always ask how the identifiers in the `R` class are generated. The short answer is that they're generated automatically by the ADT! Also note that you'll learn about some built-in UI elements in the `R` class later in the book as part of sample applications.

Though a number of imports are necessary to resolve class names in use, most of the import statements have been omitted from [listing 2.2](#) for the sake of brevity. One import

that's shown contains the definition for the `java.text.NumberFormat` class, which is used to format and parse currency values.

Another import shown is for the `android.util.Log` class, which is employed to make entries to the log. Calling static methods of the `Log` class adds entries to the log. You can view entries in the log via the LogCat view of the DDMS perspective. When making entries to the log, it's helpful to put a consistent identifier on a group of related entries using a common string, commonly referred to as the *tag*. You can filter on this string value so you don't have to sift through a mountain of LogCat entries to find your few debugging or informational messages.

Now let's go through the code in [listing 2.2](#). You connect the UI element

containing `mealprice` to a class-level variable of type `EditText` 1 by calling the `findViewById()` method and passing in the identifier for the `mealprice`, as defined by the automatically generated `R` class, found in `R.java`. With this reference, you can access the user's input and manipulate the meal price data as entered by the user. Similarly, you connect the UI element for displaying the calculated answer back to the user, again by calling the `findViewById()` method.

To know when to calculate the tip amount, you need to obtain a reference to the `Button` so you can add an event listener. You want to know when the button has been clicked. You accomplish this by adding a new `OnClickListener()` method named `onClick`.

When the `onClick()` method is invoked, you add the first of a few log entries using the static `i()` method of the `Log` class 2. This method adds an entry to the log with an `Information` classification. The `Log` class contains methods for adding entries to the log for different levels, including `Verbose`, `Debug`, `Information`, `Warning`, and `Error`. You can also filter the LogCat based on these levels, in addition to filtering on the process ID and tag value.

Now that you have a reference to the `mealprice` UI element, you can obtain the text entered by the user with the `getText()` method of the `EditText` class 3. In preparation for formatting the full meal price, you obtain a reference to the static currency formatter.

Let's be somewhat generous and offer a 20 percent tip. Then, using the formatter, let's format the full meal cost, including tip. Next, using the `setText()` method of the `TextView` UI element named `answerfield`, you update the UI to tell the user the total meal cost 4.

Because this code might have a problem with improperly formatted data, it's a good practice to put code logic into `try/catch` blocks so that the application behaves when the unexpected occurs .

Additional boilerplate files are in this sample project, but in this chapter we're concerned only with modifying the application enough to get basic, custom functionality working. You'll notice that as soon as you save your source files, the Eclipse IDE compiles the project in the background. If there are any errors, they're listed in the Problems view of the Java perspective; they're also marked in the left margin with a small red `x` to draw your attention to them.

Tip

Using the command-line tools found in the Android SDK, you can create batch builds of your applications without using the IDE. This approach is useful for software shops with a specific configuration-management function and a desire to conduct automated builds. In addition to the Android-specific build tools found under the tools subdirectory of your Android SDK installation, you'll also need JDK version 5.0 or later to complete command-line application builds. Creating sophisticated automated builds of Android applications is beyond the scope of this book, but you can learn more about the topic of build scripts by reading *Ant in Action: Second Edition of Java Development with Ant*, by Steve Loughran and Erik Hatcher, found at www.manning.com/loughran/.

Assuming there are no errors in the source files, your classes and UI files will compile correctly. But what needs to happen before your project can be run and tested in the Android emulator?

2.3.3. Packaging the application

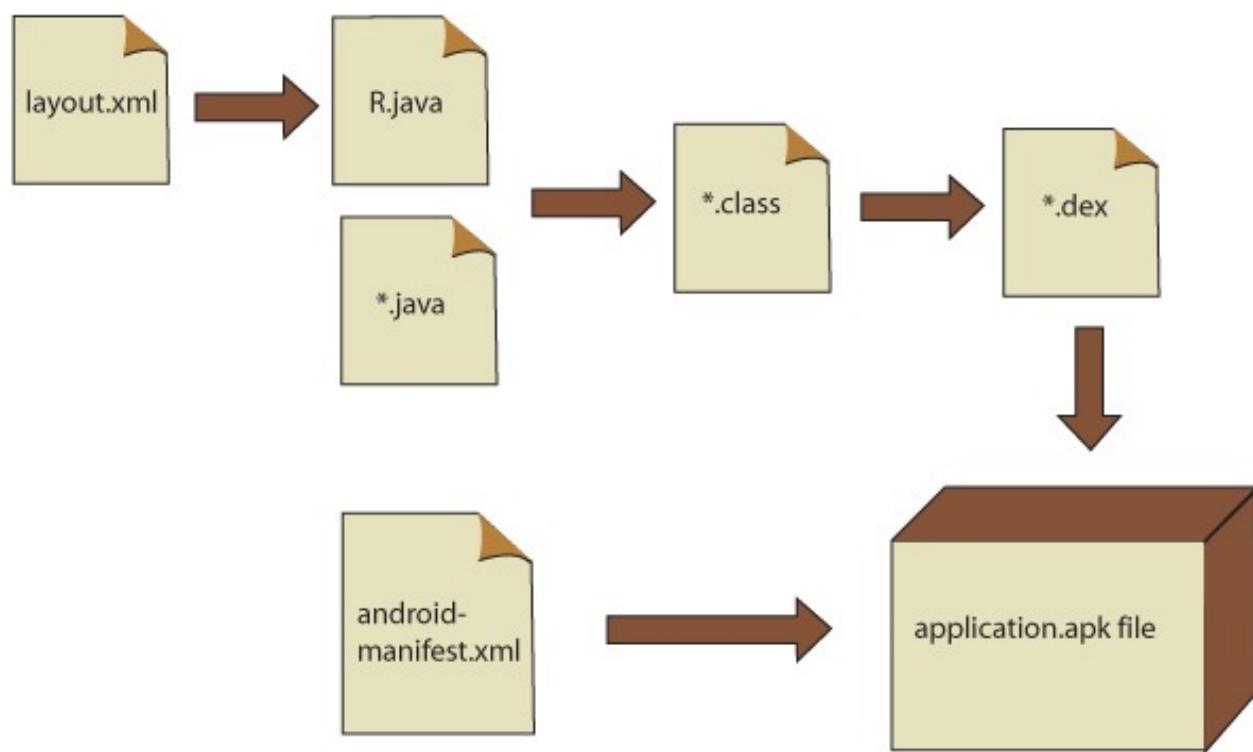
At this point, your application has compiled and is ready to be run on the device. Let's look more deeply at what happens after the compilation step. You don't need to perform these steps because the ADTs handle them for you, but it's helpful to understand what's happening behind the scenes.

Recall that despite the compile-time reliance on Java, Android applications don't run in a Java VM. Instead, the Android SDK employs the Dalvik VM. For this reason, Java byte codes created by the Eclipse compiler must be converted to the .dex file format for use in

the Android runtime. The Android SDK has tools to perform these steps, but thankfully the ADT takes care of all of this for you transparently.

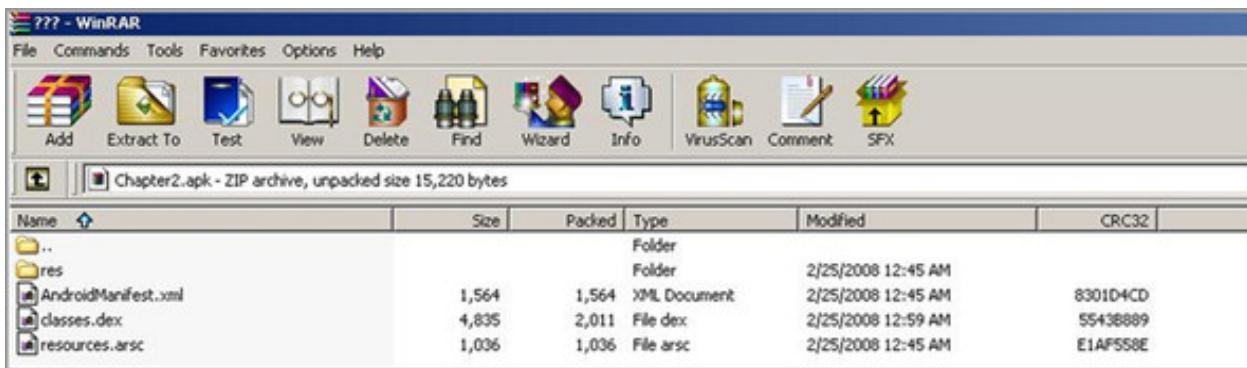
The Android SDK contains tools that convert the project files into a file ready to run on the Android emulator. [Figure 2.12](#) depicts the generalized flow of source files in the Android build process. If you recall from our earlier discussion of Android SDK tools, the tool used at design time is `aapt`. Application resource XML files are processed by `aapt`, with the `R.java` file created as a result—remember that you need to refer to the `R` class for UI identifiers when you connect your code to the UI. Java source files are first compiled to class files by your Java environment, typically Eclipse and the JDT. After they’re compiled, they’re then converted to dex files to be ready for use with Android’s Dalvik VM. Surprisingly, the project’s XML files are converted to a binary representation, not to text as you might expect. But the files retain their `.xml` extension on the device.

Figure 2.12. The ADT employs tools from the Android SDK to convert source files to a package that’s ready to run on an Android device or emulator.



The converted XML files, a compiled form of the nonlayout resources including the `Drawables` and `Values`, and the dex file (`classes.dex`) are packaged by the `aapt` tool into a file with a naming structure of *projectname.apk*. The resulting file can be read with a pkzip-compatible reader, such as WinRAR or WinZip, or the Java archiver, `jar`. [Figure 2.13](#) shows this chapter’s sample application in WinRAR.

Figure 2.13. The Android application file format is pkzip compatible.



Now you're finally ready to run your application on the Android emulator! It's important to become comfortable with working in an emulated environment when you're doing any serious mobile software development. There are many good reasons for you to have a quality emulator available for development and testing. One simple reason is that having multiple real devices with requisite data plans is an expensive proposition. A single device alone might cost hundreds of dollars. Android continues to gain momentum and is finding its way to multiple carriers with numerous devices and increasingly sophisticated capabilities. Having one of every device is impractical for all but development shops with the largest of budgets. For the rest of us, a device or two and the Android emulator will have to suffice. Let's focus on the strengths of emulator-based mobile development.

Speaking of testing applications, it's time to get the tip calculator application running!

2.4. USING THE ANDROID EMULATOR

At this point, your sample application, the Android tip calculator, has compiled successfully. Now you want to run your application in the Android emulator. Before you can run an application in the emulator, you have to configure the emulated environment. To do this, you'll learn how to create an instance of the AVD using the AVD Manager. After you've got that sorted out, you'll define a run configuration in Eclipse, which allows you to run an application in a specific AVD instance.

Tip

If you've had any trouble building the sample application, now would be a good time to go back and clear up any syntax errors that are preventing the application from building. In Eclipse, you can easily see errors because they're marked with a red x next to the project source file and on the offending lines. If you continue to have errors, make sure that your build environment is set up correctly. Refer to [appendix A](#) of this book for details on configuring the build environment.

2.4.1. Setting up the emulated environment

Setting up your emulator environment can be broken down into two logical steps. The first is to create an instance of the AVD via the AVD Manager. The second is to define a run configuration in Eclipse, which permits you to run your application in a specific AVD instance. Let's start with the AVD Manager.

Emulator vs. simulator

You might hear the words *emulator* and *simulator* thrown about interchangeably. Although they have a similar purpose—testing applications without the requirement of real hardware—those words should be used with care.

A simulator tool works by creating a testing environment that behaves as close to 100 percent in the same manner as the real environment, but it's just an approximation of the real platform. This doesn't mean that the code targeted for a simulator will run on a real device, because it's compatible only at the source-code level. Simulator code is often written to be run as a software program running on a desktop computer with Windows DLLs or Linux libraries that mimic the application programming interfaces (APIs) available on the real device. In the build environment, you typically select the CPU type for a target, and that's often x86/Simulator.

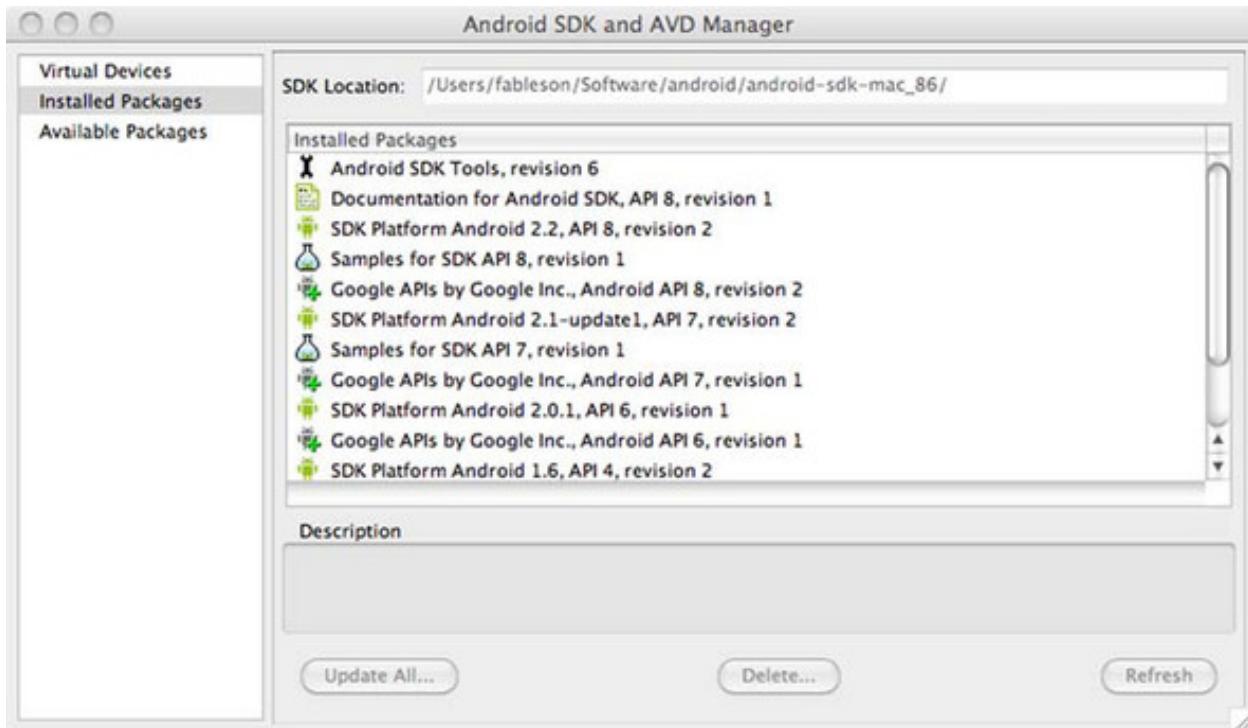
In an emulated environment, the target of your projects is compatible at the binary level. The code you write works on an emulator as well as the real device. Of course, some aspects of the environment differ in terms of how certain functions are implemented on an emulator. For example, a network connection on an emulator runs through your development machine's network interface card, whereas the network connection on a real phone runs over the wireless connection such as a GPRS, EDGE, or EVDO network. Emulators are preferred because they more reliably prepare you to run your code on real devices. Fortunately, the environment available to Android developers is an emulator, not a simulator.

Managing AVDs

Starting with version 1.6 of the Android SDK, developers have a greater degree of control over the emulated Android environment than in previous releases. The SDK and AVD Manager permit developers to download the specific platforms of interest. For

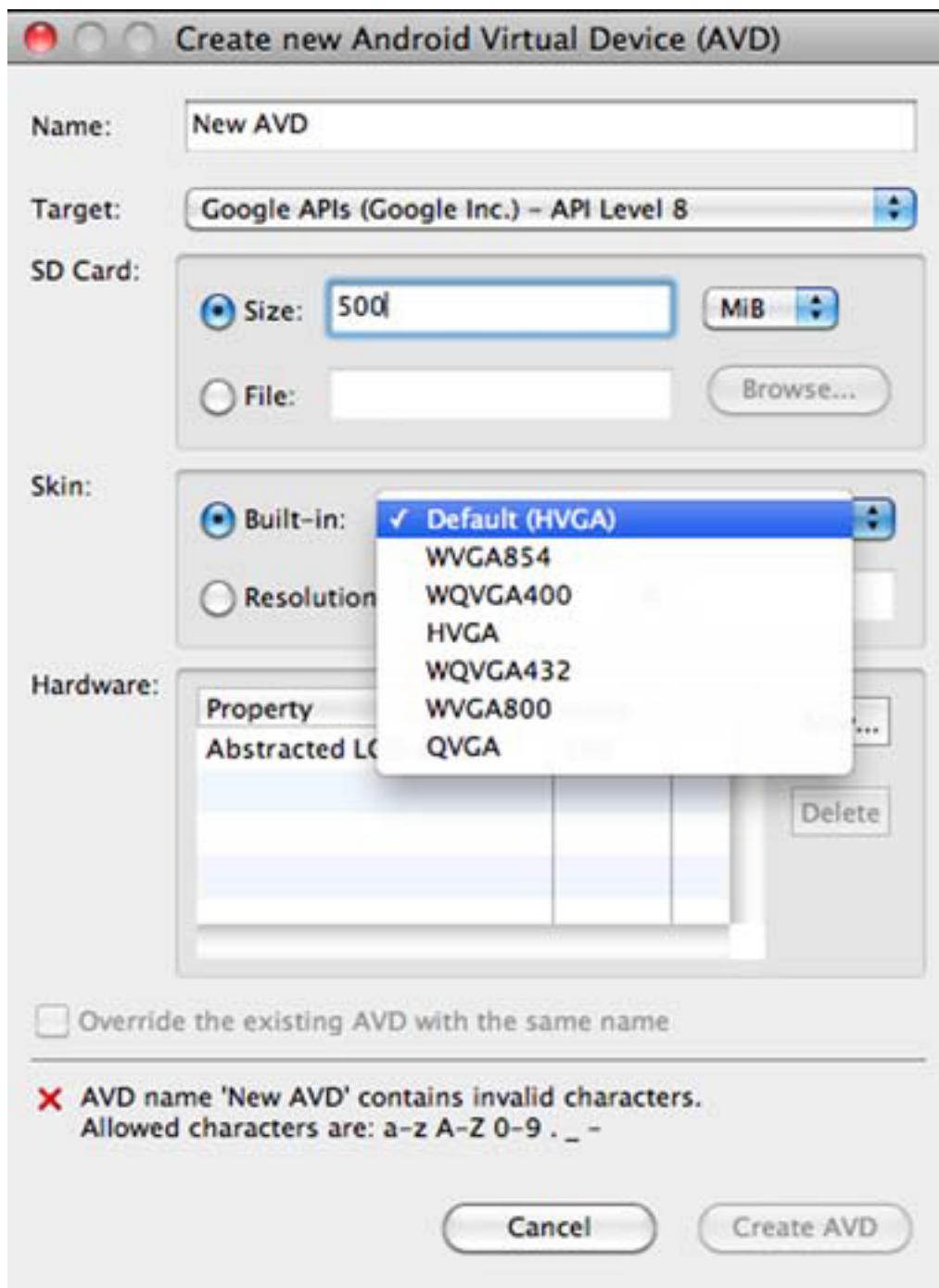
example, you might be targeting devices running version 1.5 and 2.2 of the Android platform, but you might want to add to that list as new versions become available. [Figure 2.14](#) shows the SDK and AVD Manager with a few packages installed.

Figure 2.14. The installed Android packages listed in the AVD and SDK Manager



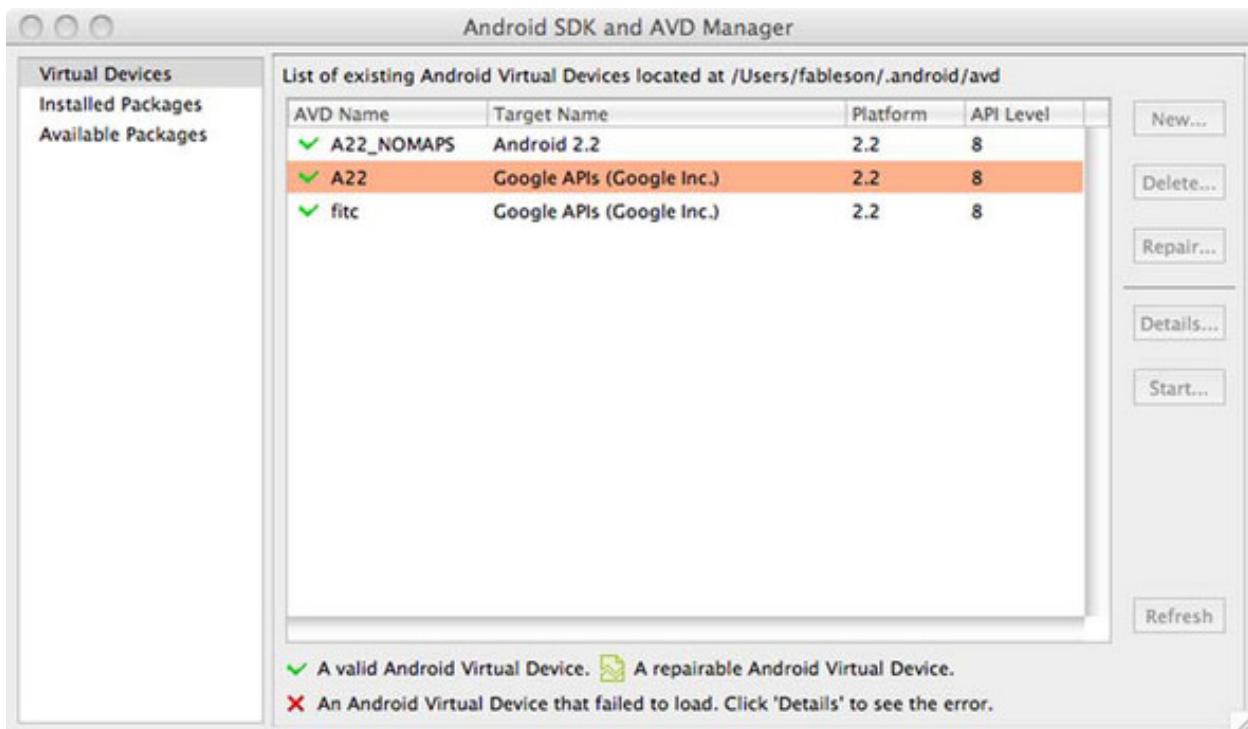
After you've installed the Android platforms that you want, you can define instances of the AVD. To define instances, select which platform you want to run on, select the device characteristics, and then create the AVD, as shown in [figure 2.15](#).

Figure 2.15. Creating a new AVD includes defining characteristics such as SD card storage capacity and screen resolution.



At this point, your AVD is created and available to be started independently. You can also use it as the target of a run configuration. [Figure 2.16](#) shows a representative list of available AVDs on a single development machine.

Figure 2.16. Available AVDs defined. You can set up as many different AVD instances as your requirements demand.



Note

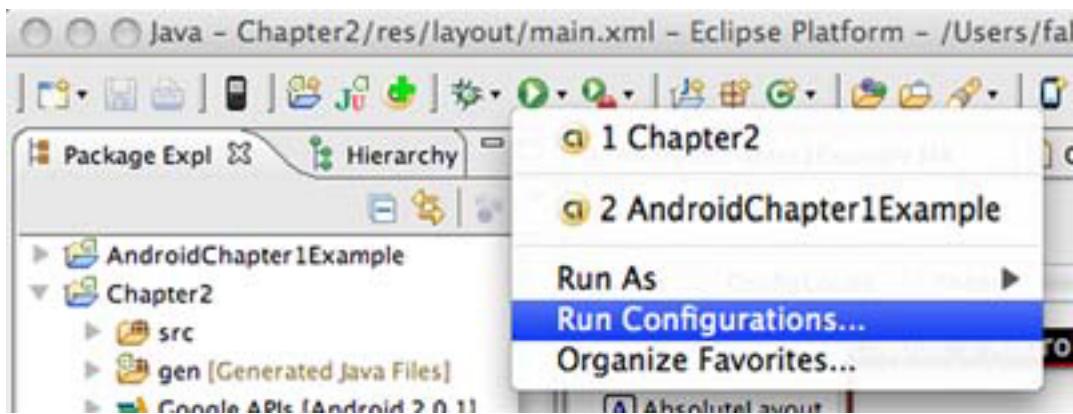
Each release of the Android platform has two versions: one with Google APIs and one without. In [Figure 2.16](#), notice that the first entry, named A22_NOMAPS, has a target of Android 2.2. The second entry, A22, has a target of Google APIs (Google Inc.). The Google version is used when you want to include application functionality such as Google Maps. Using the wrong target version is a common problem encountered by developers new to the Android platform hoping to add mapping functionality to their applications.

Now that you have the platforms downloaded and the AVDs defined, it's time to wire these things together so you can test and debug your application!

Setting up Emulator Run Configurations

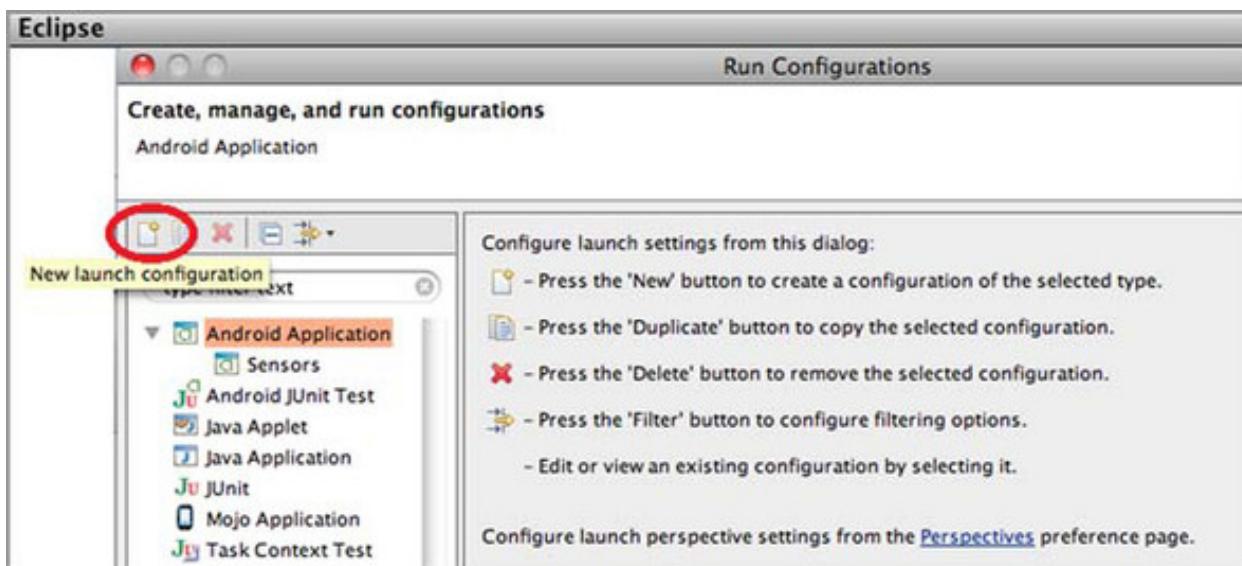
Your approach is to create a new Android emulator profile so you can easily reuse your test environment settings. The starting place is the Open Run Dialog menu in the Eclipse IDE, as shown in [figure 2.17](#). As new releases of Eclipse become available, these screen shots might vary slightly from your personal development environment.

Figure 2.17. Creating a new launch configuration for testing your Android application



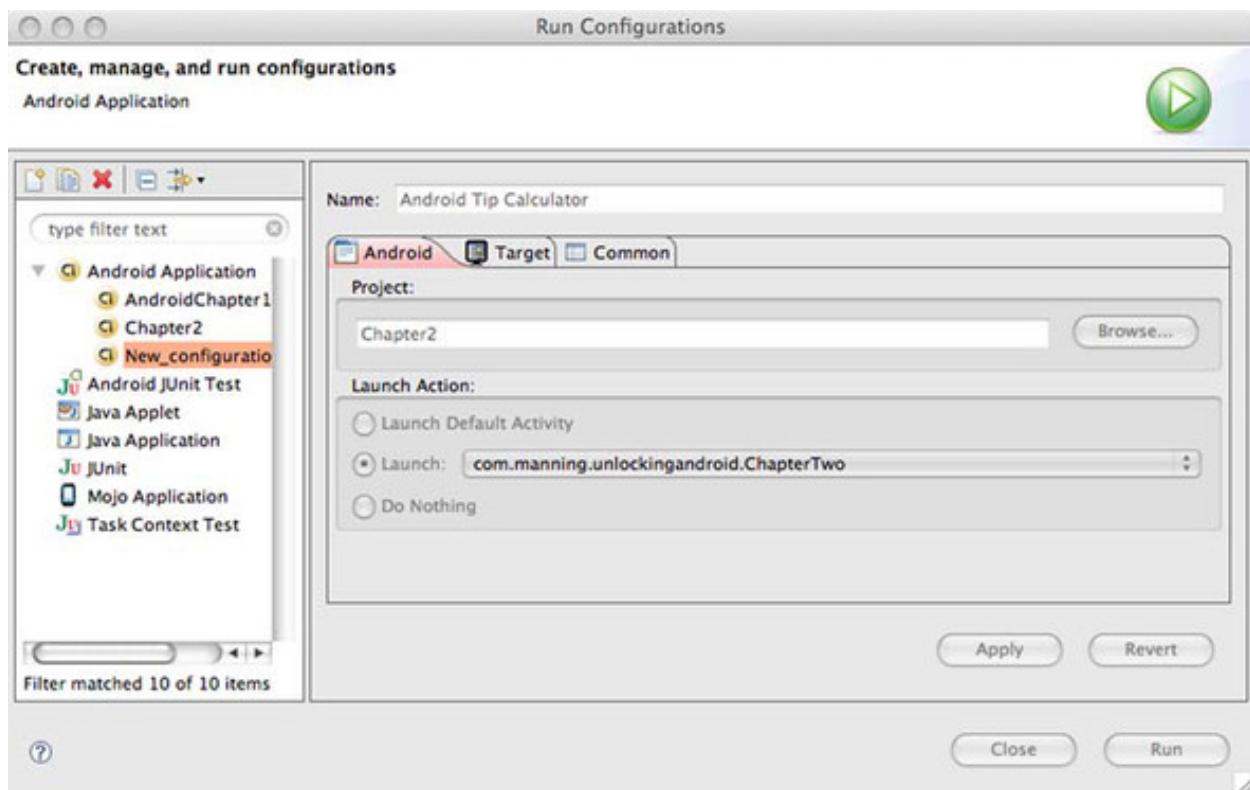
You want to create a new launch configuration, as shown in [figure 2.18](#). To begin this process, highlight the Android Application entry in the list to the left, and click the New Launch Configuration button, circled in [figure 2.18](#).

Figure 2.18. Create a new run configuration based on the Android template.



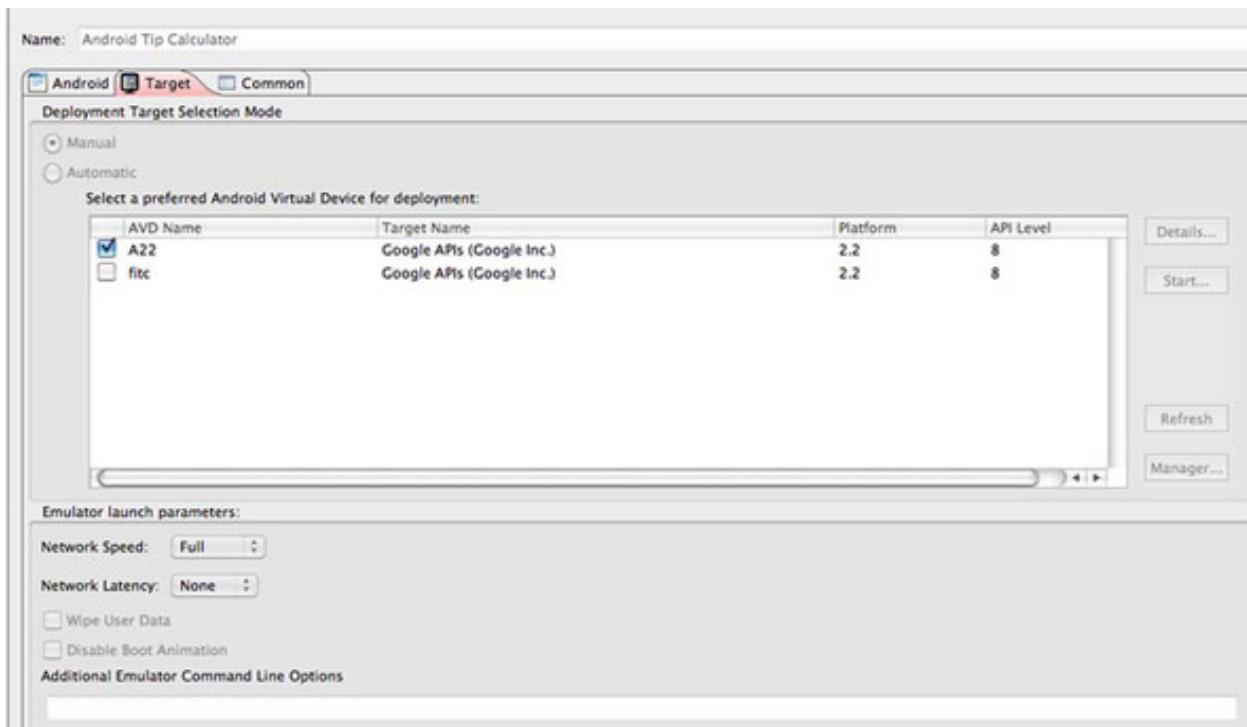
Now, give your launch configuration a name that you can readily recognize. You're going to have quite a few of these launch configurations on the menu, so make the name something unique and easy to identify. The sample is titled Android Tip Calculator, as shown in [figure 2.19](#). The three tabs have options that you can configure. The Android tab lets you select the project and the first Activity in the project to launch.

Figure 2.19. Setting up the Android emulator launch configuration



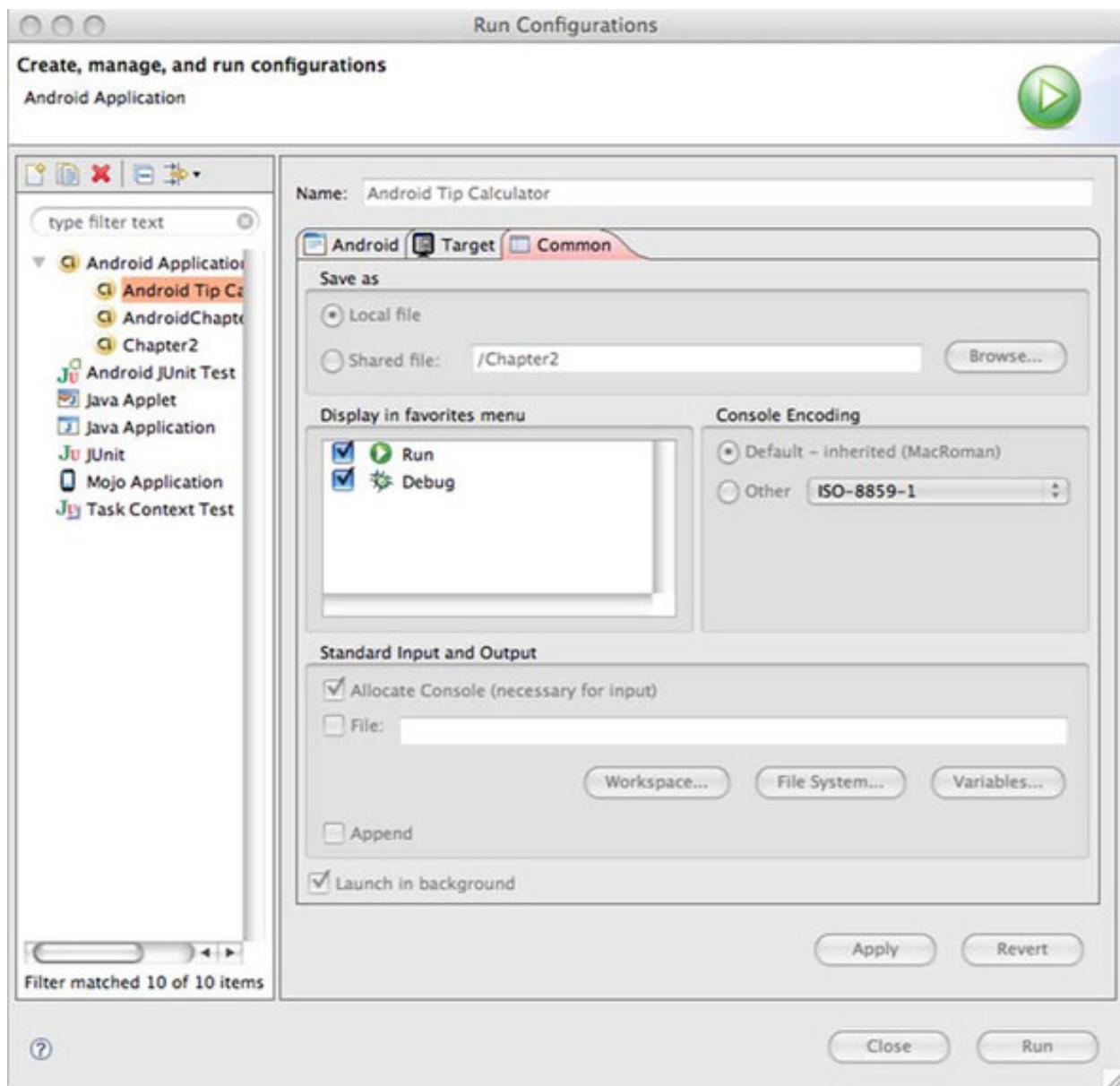
Use the next tab to select the AVD and network characteristics that you want, as shown in [figure 2.20](#). Additionally, command-line parameters might be passed to the emulator to customize its behavior. For example, you might want to add the parameter `wipe-data` to erase the device's persistent storage prior to running your application each time the emulator is launched. To see the available command-line options available, run the Android emulator from a command or terminal window with the option `emulator -help`.

Figure 2.20. Selecting the AVD to host the application and specify launch parameters



Use the third tab to put this configuration on the Favorites menu in the Eclipse IDE for easy access, as shown in [figure 2.21](#). You can select Run, Debug, or both. Let's choose both for this example, because it makes for easier launching when you want to test or debug the application.

Figure 2.21. Adding the run configuration to the toolbar menu

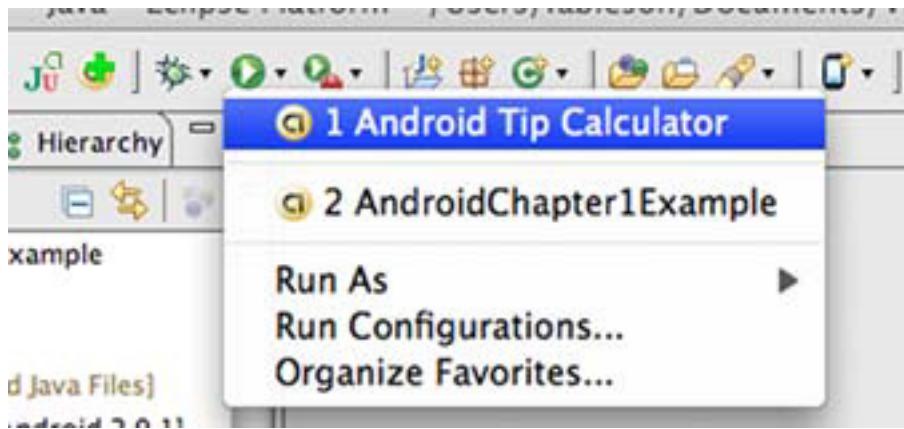


Now that you've defined your AVD and created a run configuration in Eclipse, you can test your application in the Android emulator environment.

2.4.2. Testing your application in the emulator

You're finally ready to start the Android emulator to test your tip calculator application. Select the new launch configuration from the Favorites menu, as shown in [figure 2.22](#).

Figure 2.22. Starting this chapter's sample application, an Android tip calculator



If the AVD that you choose is already running, the ADT attempts to install the application directly; otherwise, the ADT must first start the AVD and then install the application. If the application was already running, it's terminated and the new version replaces the existing copy within the Android storage system.

At this point, the Android tip calculator should be running in the Android emulator! Go ahead; test it! But wait, what if there's a problem with the code but you're not sure where? It's time to briefly look at debugging an Android application.

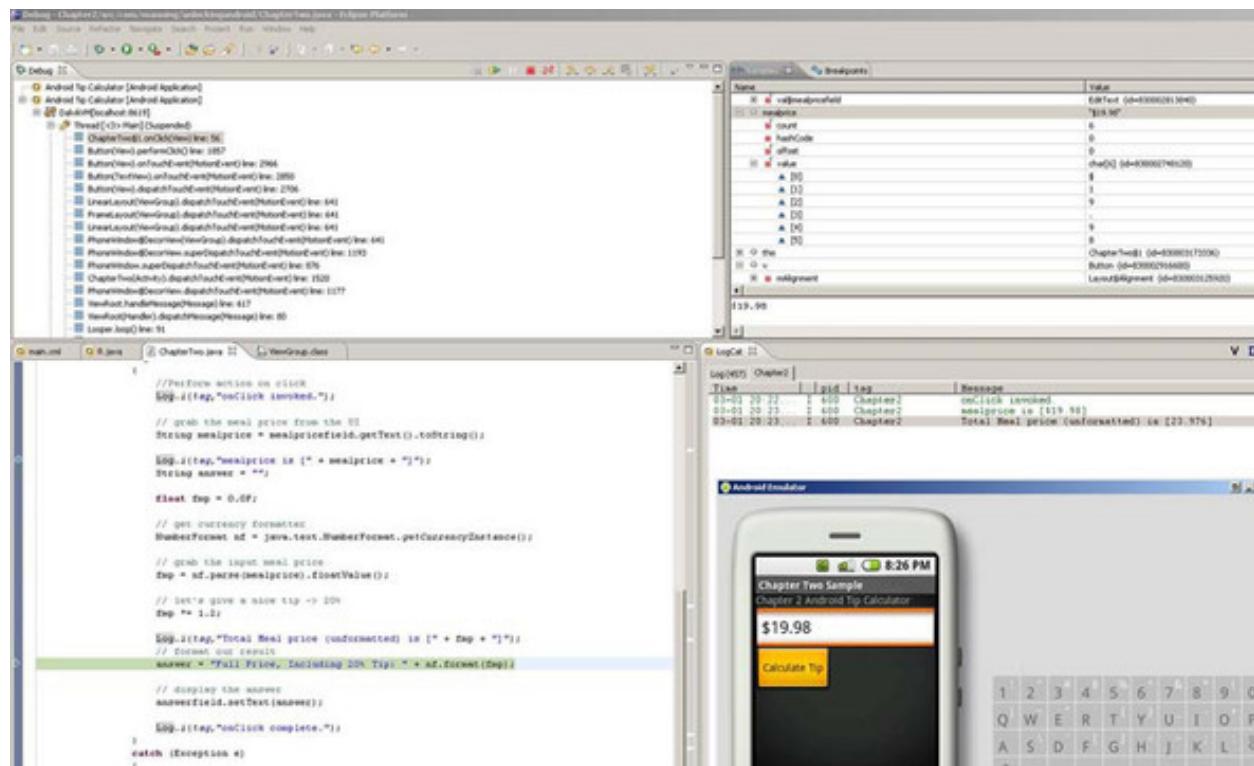
2.5. DEBUGGING YOUR APPLICATION

Debugging an application is a skill no programmer can survive without. Fortunately, debugging an Android application is straightforward under Eclipse. The first step to take is to switch to the Debug perspective in the Eclipse IDE. Remember, you switch from one perspective to another by using the Open Perspective submenu found under the Window menu.

Starting an Android application for debugging is as simple as running the application. Instead of selecting the application from the Favorites Run menu, use the Favorites Debug menu instead. This menu item has a picture of an insect (that is, a bug). Remember, when you set up the launch configuration, you added this configuration to both the Run and the Favorites Debug menus.

The Debug perspective gives you debugging capabilities similar to other development environments, including the ability to single-step into, or over, method calls, and to peer into variables to examine their value. You can set breakpoints by double-clicking in the left margin on the line of interest. [Figure 2.23](#) shows how to step through the Android tip calculator project. The figure also shows the resulting values displayed in the LogCat view. Note that the full meal price, including tip, isn't displayed on the Android emulator yet, because that line hasn't yet been reached.

Figure 2.23. The Debug perspective permits you to step line-by-line through an Android application.



Now that we've gone through the complete cycle of building an Android application and you have a good foundational understanding of using the Android ADT, you're ready to move on to digging in and unlocking Android application development by learning about each of the fundamental aspects of building Android applications.

2.6. SUMMARY

This chapter introduced the Android SDK and offered a glance at the Android SDK's Java packages to get you familiar with the contents of the SDK from a class library perspective. We introduced the key development tools for Android application development, including the Eclipse IDE and the ADT plug-in, as well as some of the behind-the-scenes tools available in the SDK.

While you were building the Android tip calculator, this chapter's sample application, you had the opportunity to navigate between the relevant perspectives in the Eclipse IDE. You used the Java perspective to develop your application, and both the DDMS perspective and the Debug perspective to interact with the Android emulator while your application was running. A working knowledge of the Eclipse IDE's perspectives will be helpful as you progress to build the sample applications and study the development topics in the remainder of this book.

We discussed the Android emulator and some of its fundamental permutations and characteristics. Employing the Android emulator is a good practice because of the benefits of using emulation for testing and validating mobile software applications in a consistent and cost-effective manner.

From here, the book moves on to dive deeper into the core elements of the Android SDK and Android application development. The next chapter continues this journey with a discussion of the fundamentals of the Android UI.

Chapter 3. User interfaces

This chapter covers

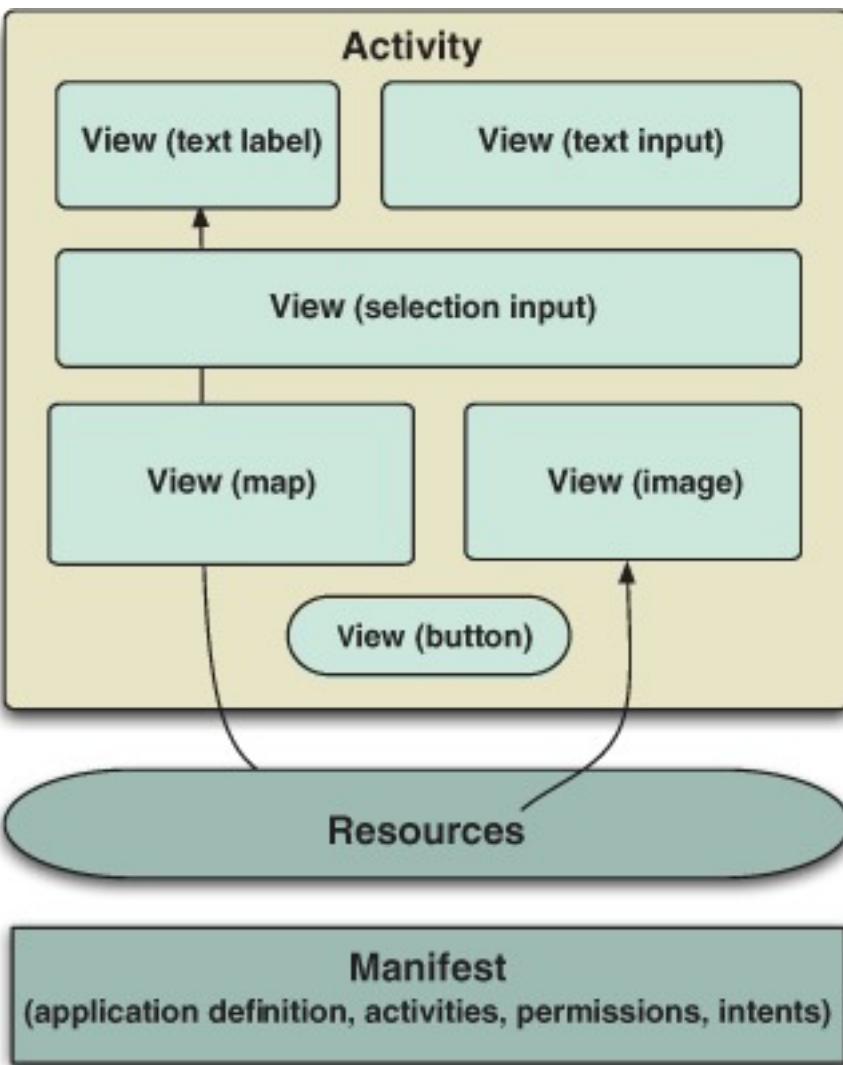
- Understanding activities and views
- Exploring the `Activity` lifecycle
- Working with resources
- Exploring the `AndroidManifest.xml` file

With our introductory tour of the main components of the Android platform and development environment complete, it's time to look more closely at the fundamental Android concepts surrounding activities, views, and resources. Activities provide screens for your application and play a key role in the Android application lifecycle. The Android framework manages the life span of visible screens, and you'll learn how to respond to the various lifecycle points you encounter.

The visible part of an `Activity` consists of subcomponents called *views*. Views are what your users see and interact with. Views handle layout, provide text elements for labels and feedback, provide buttons and forms for user input, and draw images to the device screen. You can also associate views with interface event listeners, such as those for touch-screen controls. A hierarchical collection of views is used to compose an `Activity`.

Views use strings, colors, styles, and graphic resources, which Android compiles into a binary form and makes available to applications as resources. The automatically generated `R.java` class, which we introduced in [chapter 1](#), provides a reference to individual resources and is the bridge between binary references and the source code of an Android application. You use the `R` class, for example, to grab a string of text or a color and add it to a view. The relationship between activities, views, and resources is depicted in [figure 3.1](#).

Figure 3.1. High-level diagram of `Activity`, view, resource, and manifest relationships, showing that activities are made up of views, and views use resources



Along with the components you use to build an application—views, resources, and activities—Android includes the manifest file we introduced in [chapter 1](#): `AndroidManifest.xml`. This XML file provides entrance points into your app, as well as describes what permissions it has and what components it includes. Because every Android application requires this file, we'll address it in more detail in this chapter, and we'll come back to it frequently in later parts of the book. The manifest file is the one-stop shop for the platform to start and manage your application.

If you've done any development involving UIs on any platform, the concepts of activities, views, and resources should seem familiar. Android approaches UI in a slightly different way, and this chapter will help address common points of confusion.

First, we'll introduce the sample application that we use to walk through these concepts, moving beyond theory and into the code to construct an `Activity`. You can download the complete source code for this sample from this book's website. This chapter will

include the portions that focus on the user interface, [chapter 4](#) adds the sections that integrate with other Android apps, and the online portions include the remaining components such as networking and parsing.

3.1. CREATING THE ACTIVITY

Over the course of this chapter and the next, you'll build a sample application that allows users to search for restaurant reviews based on location and cuisine. This application, RestaurantFinder, will also allow the user to call the restaurant, visit its website, or look up map directions. We chose this application as a starting point because it has a clear and simple use case, and because it involves many different parts of the Android platform. Making a sample application will let us cover a lot of ground quickly, with the additional benefit of providing a useful app on your Android phone.

To create this application, you'll need three basic screens to begin with:

- A criteria screen where the user enters parameters to search for restaurant reviews
- A list-of-reviews screen that shows pages of results matching the specified criteria
- A review-detail page that shows the details for a selected review item

Recall from [chapter 1](#) that a screen is roughly analogous to an `Activity`, which means you'll need three `Activity` classes, one for each screen. When complete, the three screens for the RestaurantFinder application will look like what's shown in [figure 3.2](#).

Figure 3.2. RestaurantFinder application screenshots, showing three activities: `ReviewCriteria`, `ReviewList`, and `ReviewDetail`



Our first step in exploring activities and views will be to build the `RestaurantFinder ReviewCriteria` screen. From there, we'll move on to the others. Along the way, we'll highlight many aspects of designing and implementing your Android UI.

3.1.1. Creating an Activity class

To create a screen, extend the `android.app.Activity` base class (as you did in [chapter 1](#)) and override the key methods it defines. The following listing shows the first portion of the `RestaurantFinder`'s `ReviewCriteria` class.

Listing 3.1. First half of the `ReviewCriteria` Activity class

```

public class ReviewCriteria extends Activity {
    private static final int MENU_GET_REVIEWS = Menu.FIRST;
    private Spinner cuisine;
    private Button grabReviews;
    private EditText location;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.review_criteria);
        location = (EditText)
            findViewById(R.id.location);
        cuisine = (Spinner)
            findViewById(R.id.cuisine);
        grabReviews = (Button)
            findViewById(R.id.get_reviews_button);
        ArrayAdapter<String> cuisines =
            new ArrayAdapter<String>(this, R.layout.spinner_view,
                getResources().
                    getStringArray(R.array.cuisines));
        cuisines.setDropDownViewResource(
            R.layout.spinner_view_dropdown);
        cuisine.setAdapter(cuisines);
        grabReviews.setOnClickListener(
            new OnClickListener() {
                public void onClick(View v) {
                    handleGetReviews();
                }
            });
    }
}

```

The diagram illustrates five steps in the code:

- 1 Override onCreate()**: Points to the `@Override` annotation and the `onCreate` method definition.
- 2 Define layout with setContentView**: Points to the `setContentView` call.
- 3 Inflate views from XML**: Points to the `findViewById` calls for `cuisine`, `grabReviews`, and `location`.
- 4 Define ArrayAdapter instance**: Points to the creation of the `ArrayAdapter` instance `cuisines`.
- 5 Set view for drop-down**: Points to the `setDropDownViewResource` call on the `cuisines` adapter.

The `ReviewCriteria` class extends `android.app.Activity`, which does a number of important things. It gives your application a context, because `Activity` itself indirectly extends the `android.content.Context` class; `Context` provides access to many important Android operations, as you'll see later. Extending `Activity` also causes you to inherit the Android lifecycle methods, which give the framework a hook to start and run your application. Finally, the `Activity` provides a container into which `View` elements can be placed.

Because an `Activity` represents an interaction with the user, it needs to provide visible components on the screen. In the `ReviewCriteria` class, you reference three views in the code: `cuisine`, `grabReviews`, and `location`. `cuisine` is a `Spinner`, a special Android single-selection list component. `grabReviews` is a `Button`. `location` is a type of `View` called `EditText`, a basic text-entry component.

You place `View` elements like these within an `Activity` using a *layout* to define the elements of a screen. You can define layouts and views directly in code or in a layout XML resource file. You'll learn more about views and layouts as we progress through this section.

After an `Activity` is started, the Android application lifecycle rules take over and ① the `onCreate()` method is invoked. This method is one of a series of important lifecycle methods the `Activity` class provides. Every `Activity` overrides `onCreate()`, where component initialization steps are invoked.

Inside the `onCreate()` method, you'll typically invoke `setContentView()` to display the ② content from an XML layout file. An XML layout file defines `view` objects, organized into a hierarchical tree structure. After they're defined in relation to the parent layout, each view can then be inflated at runtime.

3.1.2. XML vs. programmatic layouts

Android provides APIs that allow you to manage your layout through Java code instead of XML. Although this approach may be more familiar and comfortable for developers from other mobile platforms, you should generally avoid it. XML layouts tend to be much easier to read, understand, and maintain, and they nicely enforce separation of your app's UI from its logic.

Views that need some runtime manipulation, such as binding to data, can then be referenced in code and cast to their respective subtypes ③. Views that are static—those you don't need to interact with or update at runtime, such as labels—don't need to be referenced in code at all. These views automatically show up on the screen because they're part of the `layout` as defined in the XML. For example, the screenshots in [figure 3.1](#) show two labels in the `ReviewCriteria` screen as well as the three inputs we've already discussed. These labels aren't present in the code; they're defined in the `review_criteria.xml` file that's associated with this `Activity`. You'll see this layout file when we discuss XML-defined resources.

The next area of interest in `ReviewCriteria` `Activity` is binding data to the select list views, the `Spinner` objects. Android provides an adapter concept used to link views with an underlying data source. An adapter is a collection handler that returns each item in the collection as a `View`. Android provides many basic adapters: `ListAdapter`, `ArrayAdapter`, `GalleryAdapter`, `CursorAdapter`, and more. You can also easily create your own adapter, a technique you'll use when we discuss creating custom views in [section 3.2](#). Here, we're using an `ArrayAdapter` that's populated with `Context(this)`, a `View` element defined in an XML resource file, and an array representing the data. Note that the underlying data source for the array is also defined as a resource in XML ④—which you'll learn more about in [section 3.3](#). When we create the `ArrayAdapter`, we define the `View` to be used for the `element` shown in the `Spinner` before it's selected by the user. After it's selected, it must provide a different

visual interface—this is the view defined in the drop-down **5**. After we define the adapter and its view elements, we set it in the `Spinner` object.

The last thing this initial `Activity` demonstrates is our first explicit use of event handling. UI elements support many types of events, many of which you'll learn about in [section 3.2.7](#). In this instance, we're using an `OnClickListener` with our `Button` in order to respond to button clicks.

After the `onCreate()` method finishes and our data is bound to our `Spinner` views, we have menu items and their associated action handlers. The next listing shows how these are implemented in the last part of `ReviewCriteria`.

Listing 3.2. Second half of the `ReviewCriteria` Activity class

```
...
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    menu.add(0, ReviewCriteria.MENU_GET_REVIEWS, 0,
        R.string.menu_get_reviews).setIcon(
        android.R.drawable.ic_menu_more);
    return true;
}
@Override
public boolean onOptionsItemSelected(int featureId, MenuItem item) {
    switch (item.getItemId()) {
        case MENU_GET_REVIEWS:
            handleGetReviews();           ← 1 Respond when
            return true;                  menu item selected
    }
    return super.onOptionsItemSelected(featureId, item);
}
private void handleGetReviews() {
    if ((location.getText() == null) ||
        location.getText().toString().equals("")) {
        new AlertDialog.Builder(this).setTitle(R.string.alert_label).
            setMessage(R.string.location_not_supplied_message).
            setPositiveButton("Continue",
                new android.content.DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int arg1) {
                        // Just close alert.
                    }
                });
    }
}
```

← 2 Define method to process reviews

```

        }
        }).show();
    return;
}
RestaurantFinderApplication application =
(RestaurantFinderApplication)
getApplication();
application.setReviewCriteriaCuisine(
cuisine.getSelectedItem().toString());
application.setReviewCriteriaLocation(
location.getText().toString());
Intent intent =
new Intent(Constants.INTENT_ACTION_VIEW_LIST);
startActivity(intent);
}
}

```



3 Use Application object for state



4 Create Intent

The menu items at the bottom of the `Activity` screens in [figure 3.2](#) were all created using the `onCreateOptionsMenu()` method. Here, we use the `Menu` class's `add()` method to create a single `MenuItem` element. We pass a group ID, an ID, a sequence/order, and a text resource reference as parameters to create the menu item. We also assign an icon to the menu item with the `setIcon` method. The text and the image are externalized from the code, using Android's programmer-defined resources. The `MenuItem` we've added duplicates the functionality of the onscreen `Button`, so we use the same text value for the label: `Get reviews`.

In addition to creating the menu item, we need to perform an action when

the `MenuItem` is selected. We do this in the `onMenuItemSelected()` event method **1**, where we parse the ID of the multiple possible menu items with a `switch` statement. When the `MENU_GET_REVIEWS` item is selected, we invoke

the `handleGetReviews()` method **2**.

Using the Menu vs. Onscreen Buttons

We've chosen to use the `Menu` here, in addition to the onscreen buttons. When deciding whether to use buttons, a menu, or both, you need to consider whether the `Menu`, which is invoked by pressing the `Menu` button on the device and tapping a selection (button and a tap), is appropriate for what you're doing, or whether an onscreen button (single tap) is more appropriate. Generally, onscreen buttons should be tied to UI elements, such as a button that clears search form input, and menu items should be used for broader actions such as creating, saving, or deleting.

We check for valid input and use an `AlertDialog` to warn users about problems with the location they entered. Along with generally demonstrating the use of `AlertDialog`, this demonstrates how a button can be made to respond to a click event with

an `OnItemClickListener()`. Here, the Android framework automatically dismisses the pop-up, so no extra code is required in the listener.

The Builder Pattern

You might have noticed the use of the Builder pattern, where we add parameters to the `AlertDialog` we created. In this approach, each of the methods invoked, such as `AlertDialog.setMessage()` and `AlertDialog.setTitle()`, returns a reference to itself (`this`), which means we can continue chaining method calls. This approach avoids either using an extra-long constructor with many parameters or repeating the class reference throughout the code. Intents also use this handy pattern; it's something you'll see frequently in Android.

After passing validation, this method stores the user's selection state in  the `ApplicationObject`  and prepares to call the next screen. We've moved this logic into its own method because we're using it from multiple places—both from our `onScreenButton` and our `MenuItem`.

The `Application` object is used internally by Android for many purposes, and it can be extended, as we've done with `RestaurantFinderApplication`. You can find the source of this class online. To store global state information, `RestaurantFinderApplication` defines a few member variables in JavaBean style. We reference this object from other activities to retrieve the information we're storing here. Objects can be passed back and forth between activities in several ways; using `Application` is just one of them.

After we store the criteria state, we fire off an action in the form of an  `Android Intent`  . We touched on `Intents` in [chapter 1](#), and we'll delve into them further in the next chapter; here, we ask another `Activity` to respond to the user's selection of a menu item by calling `startActivity(intent)`.

With that, we've covered a good deal of material and you've completed `ReviewCriteria`, your first `Activity`. Now that this class is fully implemented, we'll take a closer look at the Android `Activity` lifecycle and how it relates to processes on the platform.

3.1.3. Exploring the Activity lifecycle

Every process running on the Android platform is placed on a *stack*. When you use an `Activity` in the foreground, the system process that hosts that `Activity` is placed at the top of the stack, and the previous process (the one hosting whatever `Activity` was previously in the foreground) is moved down one notch. This concept is a key point to understand. Android tries to keep processes running as long as it can, but it can't keep

every process running forever because system resources are finite. What happens when memory starts to run low or the CPU gets too busy?

How Processes and Activities Relate

When the Android platform decides it needs to reclaim resources, it goes through a series of steps to prune processes and the activities they host. It decides which ones to get rid of based on a simple set of priorities:

- . The process hosting the foreground `Activity` is the most important.
- . Any process hosting a visible-but-not-foreground `Activity` comes next in terms of importance (for example, a full-screen app that's visible behind an app running in a pop-up window).
- . After that comes any process hosting a background `Activity`.
- . Any process not hosting any `Activity` (or `Service` or `BroadcastReceiver`) is known as an *empty process* and is thus first in line to be killed.

A useful tool for development and debugging, especially in the context of process priority, is the `adb` tool, which you first met in [chapter 2](#). You can see the state of all the running processes in an Android device or emulator by issuing the following command:

```
adb shell dumpsys activity
```

This command outputs a lot of information about all the running processes, including the package name, PID, foreground or background status, current priority, and more.

All `Activity` classes must be able to handle being stopped and shut down at any time. Remember, a user can and will change directions at any time. They might receive a phone call or an incoming SMS message, causing them to bounce around from one application to the next. If the process your `Activity` is in falls out of the foreground, it's eligible to be killed without your consent; it's up to the platform's algorithm, based on available resources and relative priorities.

To manage this environment, Android applications (and the `Activity` classes they host) use a different design from what you might be used to in other environments. Using a series of event-related callback methods defined in the `Activity` class, you can set up and tear down the `Activity` state gracefully. The `Activity` subclasses that you implement override a set of lifecycle methods to make this happen. As we discussed in [section 3.1.1](#), every `Activity` must implement the `onCreate()` method. This method is the starting point of the lifecycle. In addition to `onCreate()`, most activities will want to implement the `onPause()` method, where data and state can be persisted before the hosting process potentially falls out of scope.

3.1.4. The server connection

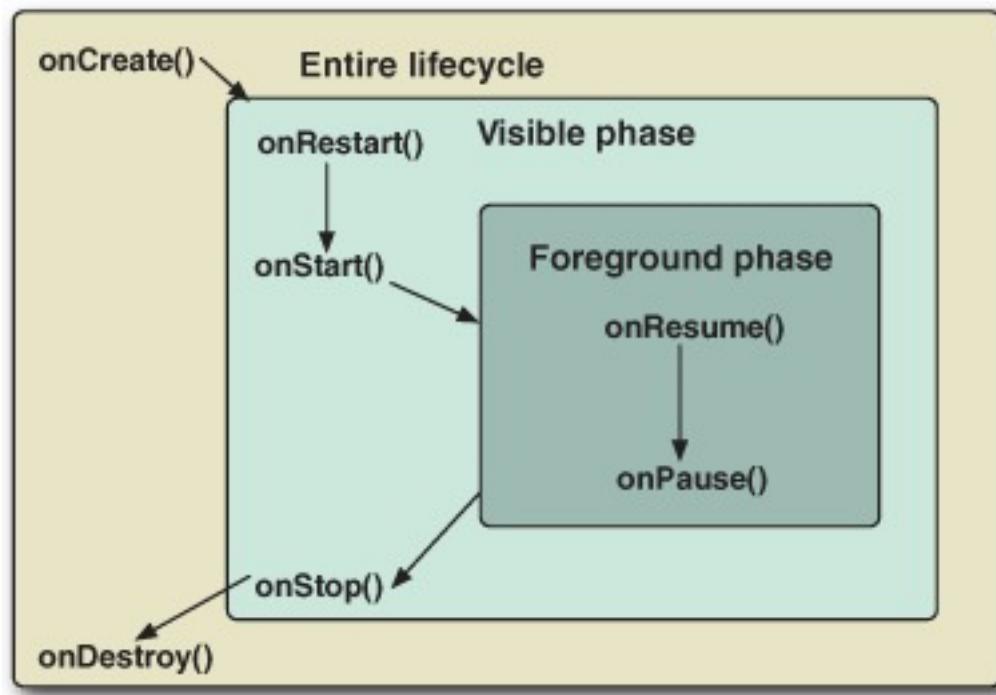
If you've worked in managed environments such as Java EE servlet containers, you should already be familiar with the concept of lifecycles. Your app responds to invocations by a framework, instead of driving its own lifespan. The critical difference for Android is that your app is much more likely to be shut down entirely, and you'll need to handle any necessary cleanup.

The lifecycle methods provided by the `Activity` class are called in a specific order by the platform as it decides to create and kill processes. Because you, as an application developer, can't control the processes, you need to rely on the callback lifecycle methods to control state in your `Activity` classes as they come into the foreground, move into the background, and fall away altogether. As the user makes choices, activities are created and paused in a defined order by the system as it starts and stops processes.

Activity Lifecycle

Beyond `onCreate()` and `onPause()`, Android provides other distinct stages, each of which is a part of a particular phase of the life of an `Activity` class. The methods that you'll encounter most and the phases for each part of the lifecycle are shown in [figure 3.3](#).

Figure 3.3. Android `Activity` lifecycle diagram, showing the methods involved in the foreground and visible phases



Each of the Android lifecycle methods has a distinct purpose, and each happens during one of the following phases:

- In the *foreground phase*, the `Activity` is viewable on the screen and is on top of everything else (when the user is interacting with the `Activity` to perform a task).
- In the *visible phase*, the `Activity` is on the screen, but it might not be on top and interacting with the user (when a dialog or floating window is on top of the `Activity`, for example).
- The *entire lifecycle phase* refers to the methods that might be called when the application isn't on the screen, before it's created, and after it's gone (prior to being shut down).

[Table 3.1](#) provides more information about the lifecycle phases and outlines the main high-level methods on the `Activity` class.

Table 3.1. Android `Activity` main lifecycle methods and their purposes

Method	Purpose
<code>onCreate()</code>	Called when the Activity is created. Setup is done here. Also provides access to any previously stored state in the form of a <code>Bundle</code> , which can be used to restore what the user was doing before this Activity was destroyed.
<code>onRestart()</code>	Called if the Activity is being restarted, if it's still in the stack, rather than starting new.
<code>onStart()</code>	Called when the Activity is becoming visible on the screen to the user.
<code>onResume()</code>	Called when the Activity starts interacting with the user. (This method is always called, whether starting or restarting.)
<code>onPause()</code>	Called when the Activity is pausing or reclaiming CPU and other resources. This method is where you should save state information so that when an Activity is restarted, it can start from the same state it was in when it quit.
<code>onStop()</code>	Called to stop the Activity and transition it to a nonvisible phase and subsequent lifecycle events.
<code>onDestroy()</code>	Called when an Activity is being completely removed from system memory. This method is called either because <code>onFinish()</code> is directly invoked or because the system decides to stop the Activity to free up resources.

Beyond the main high-level lifecycle methods outlined in [table 3.1](#), additional, finer-grained methods are available. You don't typically need methods such as `onPostCreate()` and `onPostResume()`, but be aware that they exist if you need that level of control. See the `Activity` documentation for full method details.

As for the main lifecycle methods that you'll use the majority of the time, it's important to know that `onPause()` is your last opportunity to clean up and save state information. The processes that host your `Activity` classes won't be killed by the platform until after the `onPause()` method has completed, but they might be killed there-after. The system will attempt to run through all of the lifecycle methods every time, but if resources have grown critically low, the processes that are hosting activities which are beyond

the `onPause()` method might be killed *at any point*. Any time your `Activity` is moved to the background, `onPause()` is called. Before your `Activity` is completely removed, `onDestroy()` is called, although it might not be invoked in all circumstances. You should save persistent state in `onPause()`. We'll discuss how to save data in [chapter 5](#).

Instance state

In addition to persistent state, you should be familiar with one more scenario: *instance state*. Instance state refers to the state of the UI itself. For example, instance state refers to the current selection of any buttons, lists, text boxes, and so on, whereas *persistent state* refers to data that you expect to remain after the phone reboots.

The `onSaveInstanceState()` method is called when an `Activity` might be destroyed, so that at a future time the interface state can be restored. This method is used transparently by the platform to handle the view state processing in the vast majority of cases; you don't need to concern yourself with it under most circumstances. Nevertheless, it's important to know that it's there and that the `Bundle` it saves is handed back to the `onCreate()` method when an `Activity` is restored—as `savedInstanceState` in most code examples. If you need to customize the view state, you can do so by overriding this method, but don't confuse this with the more common general lifecycle methods.

Managing activities with lifecycle events allows Android to do the heavy lifting, deciding when things come into and out of scope, relieving applications of the decision-making burden, and ensuring a level playing field for applications. This is a key aspect of the platform that varies somewhat from many other application-development environments. To build robust and responsive Android applications, you need to pay careful attention to the lifecycle.

Now that you have some background about the `Activity` lifecycle and you've created your first screen, we'll take a longer look at the various views that Android offers.

3.2. WORKING WITH VIEWS

Views are the building blocks of Android application's UI. Activities contain views, and `View` classes represent elements on the screen and are responsible for interacting with users through events.

Every Android screen contains a hierarchical tree of `View` elements. These views come in a variety of shapes and sizes. Many of the views you'll need on a day-to-day basis are provided as part of the platform—text elements, input elements, images, buttons, and the like. In addition, you can create your own composite views and custom views when the need arises. You can place views into an `Activity` (and thus on the screen) either directly in code or by using an XML resource that's later inflated at runtime.

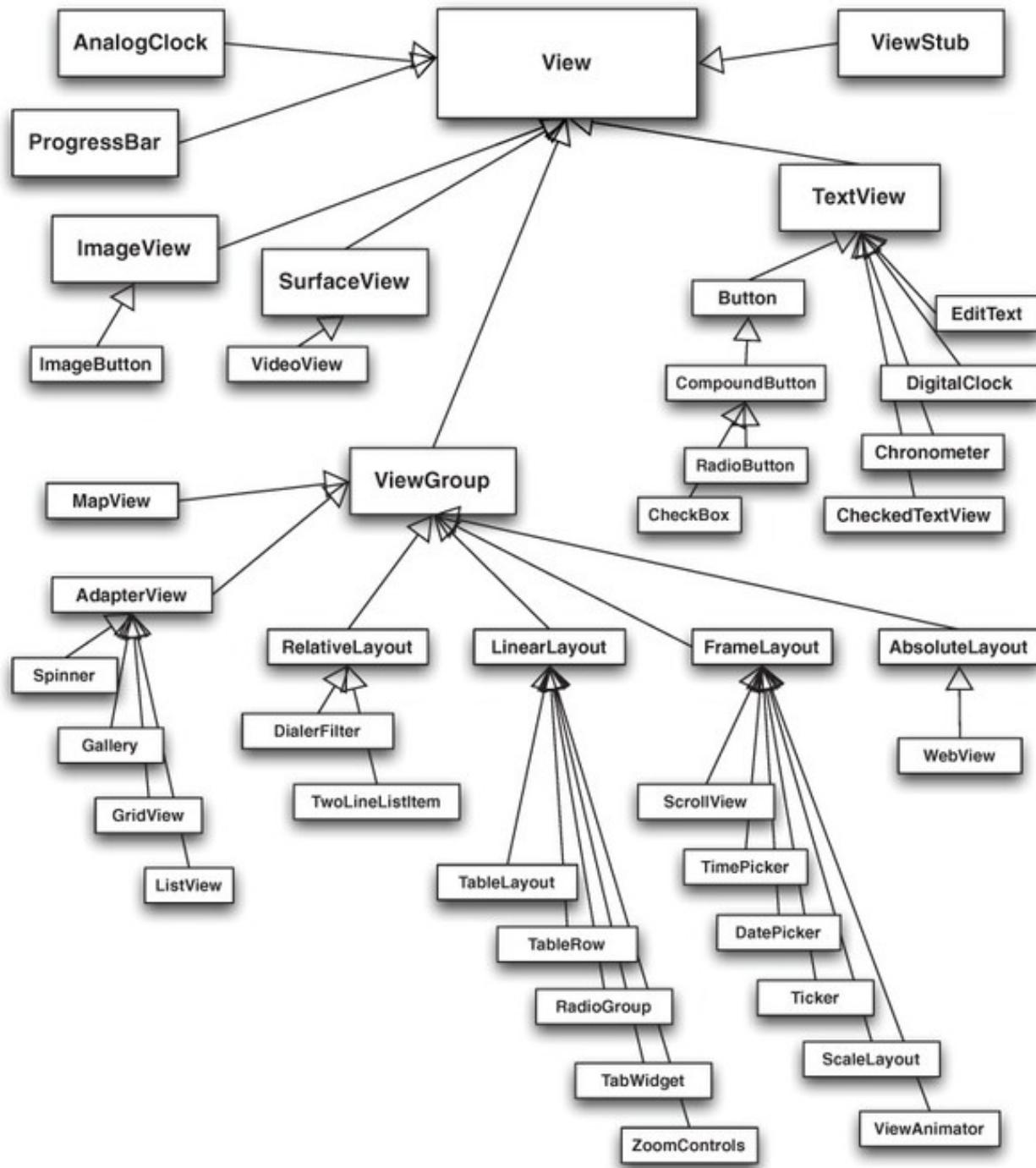
In this section, we'll discuss the fundamental aspects of views: the common views that Android provides, custom views that you can create as you need them, layout in relation to views, and event handling. Views defined in XML will be covered in [section 3.3](#) as part of a larger discussion on resources. We'll begin with the common `View` elements Android provides by taking a short tour of the API.

3.2.1. Exploring common views

Android provides a generous set of `View` classes in the `android.view` package. These classes range from familiar constructs such as the `EditText`, `Spinner`, and `TextView` that you've already seen in action, to more specialized widgets such as `AnalogClock`, `Gallery`, `DatePicker`, `TimePicker`, and `VideoView`. For a glance at some of the more eye-catching views, check out the sample page in the Android documentation: <http://mng.bz/b83c>.

The class diagram in [figure 3.4](#) provides a high-level snapshot of what the overall `View` API looks like. This diagram shows how the specializations fan out and includes many, but not all, of the `View`-derived classes.

Figure 3.4. Class diagram of the Android View API, showing the root `View` class and specializations from there. Note that `ViewGroup` classes such as layouts are also a type of `View`.



As is evident from the diagram in [figure 3.4](#), `View` is the base class for many classes. `ViewGroup` is a special subclass of `View` related to layout, as are other elements such as the commonly used `TextView`. All UI classes are derived from the `View` class, including the layout classes (which extend `ViewGroup`).

Of course, everything that extends `View` has access to the base class methods. These methods allow you to perform important UI-related operations such as setting the

background, minimum height and width, padding, layout parameters, and event-related attributes. [Table 3.2](#) lists some of the methods available in the root `View` class. Beyond the base class, each `View` subclass typically adds a host of refined methods to further manipulate its respective state, such as what's shown for `TextView` in [table 3.3](#).

Table 3.2. A subset of methods in the base Android `View` API

Method	Purpose
<code>setBackgroundColor(int color)</code>	Set the background color
<code>setBackgroundDrawable(Drawable d)</code>	Set the background drawable (such as an image or gradient)
<code>setClickable(boolean c)</code>	Set whether element is clickable
<code>setFocusable(boolean f)</code>	Set whether element is focusable
<code>setLayoutParams(ViewGroup.LayoutParams l)</code>	Set parameters for layout (position, size, and more)
<code>setMinimumHeight(int minHeight)</code>	Set the minimum height (parent can override)
<code>setMinimumWidth(int minWidth)</code>	Set the minimum width (parent can override)
<code>setOnClickListener(OnClickListener l)</code>	Set listener to fire when click event occurs
<code>setOnFocusChangeListener(OnFocusChangeListener l)</code>	Set listener to fire when focus event occurs
<code>setPadding(int left, int right, int top, int bottom)</code>	Set the padding

Table 3.3. More `View` methods for the `TextView` subclass

Method	Purpose
<code>setGravity(int gravity)</code>	Set alignment gravity: top, bottom, left, right, and more
<code>setHeight(int height)</code>	Set height dimension
<code>setText(CharSequence text)</code>	Set text to display in <code>TextView</code>
<code>setTypeface(TypeFace face)</code>	Set typeface
<code>setWidth(int width)</code>	Set width dimension

The `View` base class and the methods specific to `TextView` combine to give you extensive control over how an application can manipulate an instance of `TextView`. For example, you can set layout, padding, focus, events, gravity, height, width, colors, and so on. These methods can be invoked in code or set at design time when defining a UI layout in XML, as we'll introduce in [section 3.3](#).

Each `View` element you use has its own unique API; for details on all the methods, see the Android Javadocs at <http://mng.bz/82Qy>.

When you couple the wide array of classes with the rich set of methods available from the base `View` class on down, the Android `View` API can seem intimidating. Thankfully, despite this initial impression, many of the concepts involved quickly become evident; and their use becomes more intuitive as you move from view to view, because they're ultimately just specializations of the same base class. When you get familiar with working with `View` classes, learning to use a new view becomes intuitive and natural.

Although the RestaurantFinder application won't use many of the views listed in our whirlwind tour here, they're still useful to know about. We'll use many of them in later examples throughout the book.

The next thing we'll focus on is a bit more detail concerning one of the most common nontrivial `View` elements—the `ListView` component.

3.2.2. Using a `ListView`

On the RestaurantFinder application's `ReviewList Activity`, shown in [figure 3.2](#), you can see a view that's different from the simple user inputs and labels we've used up to this point—this screen presents a scrollable list of choices for the user to pick from.

This `Activity` uses a `ListView` component to display a list of review data that's obtained from calling a mock web service for restaurant reviews. We make an HTTP call by appending the user's criteria to the mock web service's URL. We then parse the results with the *Simple API for XML (SAX)* and create a `List` of reviews. Neither the details of XML parsing nor the use of the network itself is of much concern to us here—rather we'll focus on the views employed to represent the data returned from the web service call. The resulting `List` will be used to populate our screen's list of items to choose from.

The code in the following listing shows how to create and use a `ListView` to present to the user the `List` of reviews within an `Activity`.

Listing 3.3. First half of the `ReviewList Activity` class, showing a `ListView`

```

public class ReviewList extends ListActivity {
    private static final int MENU_CHANGE_CRITERIA = Menu.FIRST + 1;
    private static final int MENU_GET_NEXT_PAGE = Menu.FIRST;
    private static final int NUM_RESULTS_PER_PAGE = 8;
    private TextView empty;
    private ProgressDialog progressDialog;
    private ReviewAdapter reviewAdapter;
    private List<Review> reviews;
    private final Handler handler = new Handler() {
        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
            if ((reviews == null) || (reviews.size() == 0)) {
                empty.setText("No Data");
            } else {
                reviewAdapter = new ReviewAdapter(
                    ReviewList.this, reviews);
                setListAdapter(reviewAdapter);
            }
        }
    };
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.review_list);
        empty = (TextView)
            findViewById(R.id.empty);
        ListView listView = getListView();
        listView.setItemsCanFocus(false);
        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        listView.setEmptyView(empty);
    }
    @Override
    protected void onResume() {
        super.onResume();
        RestaurantFinderApplication application =
            (RestaurantFinderApplication) getApplication();
        String criteriaCuisine = application.getReviewCriteriaCuisine();
        String criteriaLocation = application.getReviewCriteriaLocation();
        int startFrom = getIntent().getIntExtra(
            Constants.STARTFROM_EXTRA, 1);
        loadReviews(criteriaLocation,
            criteriaCuisine, startFrom);
    }
    // onCreateOptionsMenu omitted for brevity
    . . .

```

1 Use ReviewAdapter

2 Apply resource-defined layout

3 Retrieve TextView

4 Access Application for global state

5 Use Intent extra

The `ReviewList` Activity extends `ListActivity`, which is used to host a `ListView`. The default layout of a `ListActivity` is a full-screen, centered list of choices for the user to select from. A `ListView` provides functionality similar to a `Spinner`; in fact, they're both

subclasses of `AdapterView`, as you saw in the class diagram in [figure 3.4](#). `ListView`, like `Spinner`, uses an adapter to bind to data. In this case, we're using a

custom `ReviewAdapter` class [1](#). You'll learn more about `ReviewAdapter` in the next section, when we discuss custom views. For now, note that we're using a custom adapter for our `ListView`, and we use a `List` of `Review` objects to populate the adapter.

Because we don't yet have the data to populate the list, which we'll get from a web service call in another thread, we need to include a handler to allow for fetching data and updating the UI to occur in separate steps. Don't worry too much about these concepts here; they'll make more sense when we look at the second half of `ReviewList` in [listing 3.4](#).

After we declare our `ListView` and its data, we move on to the typical `onCreate()` tasks you've already seen, including using a layout defined in an XML file [2](#). This is significant with respect to `ListActivity` because a `ListView` with the ID name `list` is required if you want to customize the layout, as we've done. Note that the ID is defined in the layout XML file; we'll cover that in [section 3.3.3](#). If you don't provide a layout, you can still use `ListActivity` and `ListView`, but you get the system default configuration. We also look up a UI element that's used to display the message `No Data` in the event that our `List` of reviews is empty [3](#). We set several specific properties on the `ListView`, using its customization methods: we make the list items selectable, allow a single selection at a time, and provide the view to display for an empty list.

After we set up the `View` elements that are needed for the `Activity`, we get the criteria to make our web service call from the `Review` object, which we previously placed in the Application back in the `ReviewCriteria` Activity [4](#). Here we also use an `Intent` extra to store a primitive `int` for page number [5](#). We pass all the criteria data (`criteriaLocation`, `criteriaCuisine`, and `startFrom`) into the `loadReviews()` method, which makes our web service call to populate the data list. This method, and several others that show how we deal with items in the list being clicked, are shown here in the second half of the `ReviewList` class.

Listing 3.4. Second half of the `ReviewList` Activity class

```

    ...
    @Override
    public boolean onMenuItemSelected
    (int featureId, MenuItem item) {
        Intent intent = null;
        switch (item.getItemId()) {
            case MENU_GET_NEXT_PAGE:
                intent = new Intent(Constants.INTENT_ACTION_VIEW_LIST);
                intent.putExtra(Constants.STARTFROM_EXTRA,
                               getIntent().getIntExtra(Constants.STARTFROM_EXTRA, 1)
                               + ReviewList.NUM_RESULTS_PER_PAGE);
                startActivity(intent);
                return true;
            case MENU_CHANGE_CRITERIA:
                intent = new Intent(this, ReviewCriteria.class);
                startActivity(intent);
                return true;
        }
        return super.onMenuItemSelected(featureId, item);
    }

    @Override
    protected void onListItemClick(ListView l, View v,
        int position, long id) {
        RestaurantFinderApplication application =
            (RestaurantFinderApplication) getApplication();
        application.setCurrentReview(reviews.get(position));
        Intent intent = new Intent(Constants.INTENT_ACTION_VIEW_DETAIL);
        intent.putExtra(Constants.STARTFROM_EXTRA, getIntent().getIntExtra(
            Constants.STARTFROM_EXTRA, 1));
        startActivity(intent);
    }

    private void loadReviews(String location, String cuisine,
        int startFrom) {
        final ReviewFetcher rf = new ReviewFetcher(location,
            cuisine, "ALL", startFrom,
            ReviewList.NUM_RESULTS_PER_PAGE);
        progressDialog =
            ProgressDialog.show(this, " Working...",
                " Retrieving reviews", true, false);
        new Thread() {
            public void run() {
                reviews = rf.getReviews();
                handler.sendEmptyMessage(0);
            }
        }.start();
    }
}

```

1 Increment startFrom Intent extra

2 Set state in Application

3 Create loadReviews method

4 Show ProgressDialog

5 Make web service call

This Activity has a menu item that allows the user to access the next page of results or change the list criteria. To support this, we must implement

the `onMenuItemSelected()` method. When the `MENU_GET_NEXT_PAGE` menu item is selected, we define a new `Intent` to reload the screen with an incremented `startFrom` value, with some assistance from

the `Intent` class's `getExtras()` and `putExtras()` methods ①.

After the menu-related methods comes the method `onListItemClick()`. Android invokes this method when a user clicks one of the list items in a `ListView`. We use the clicked item's ordinal position to reference the particular `Review` item the user selected, and we set this into the `Application` for later use in the `ReviewDetail Activity` (which we'll begin to implement in [section 3.3](#)) ②. After we have the data set, we then call the next `Activity`, including the `startFrom` extra.

In the `ReviewList` class, we have the `loadReviews()` method ③. This method is significant for several reasons. First, it sets up the `ReviewFetcher` class instance, which initiates a call to the mock web service over the network to retrieve a `List` of `Review` objects. Then it invokes the `ProgressDialog.show()` method to show the user we're retrieving data ④. Finally, it sets up a new thread ⑤, within which the `ReviewFetcher` is used, and the earlier handler you saw in the first half of `ReviewList` is sent an empty message. If you refer to [listing 3.3](#), which is when the handler was established, you can see where we dismiss the `ProgressDialog` when the message is received, populate the adapter our `ListView` is using, and call `setListAdapter()` to update the UI. The `setListAdapter()` method iterates the adapter and displays a returned view for every item. With the `Activity` configured and the handler ready to update the adapter with data, we now have a second screen in our application.

Next, we'll explore some details regarding handlers and multithreaded apps. These concepts aren't view-specific but are worth a small detour at this point, because you'll want to use these classes when you're trying to perform tasks related to retrieving and manipulating data that the UI needs—a common design pattern for building Android applications.

3.2.3. Multitasking with Handler and Message

`Handler` helps you manage messaging and scheduling operations for Android. This class allows you to queue tasks to be run on different threads and to schedule tasks using `Message` and `Runnable` objects.

The Android platform monitors the responsiveness of applications and kills those that are considered nonresponsive. An *Application Not Responding (ANR)* event occurs when no response is received to a user input for five seconds. When a user interacts with

your application by touching the screen or pressing a key, your application must respond. Not every operation in your code must complete within five seconds, but the main UI thread does need to respond within that time frame. To keep the main UI thread snappy, any long-running tasks, such as retrieving data over the network, reading a large amount of data from a database, or performing complicated or time-consuming calculations, should be performed in a separate thread, apart from the main UI thread.

Getting tasks into a separate thread and getting results back to the main UI thread is where `Handler` and related classes come into play. When a handler is created, it's associated with a `Looper`. A `Looper` is a class that contains a `MessageQueue` and that processes `Message` or `Runnable` objects that are sent via the handler.

When we used a handler in [listings 3.3](#) and [3.4](#), we created a handler with a no-argument constructor. With this approach, the handler is automatically associated with the `Looper` of the currently running thread, typically the main UI thread. The main UI thread, which is created by the process of the running application, is an instance of `HandlerThread`. A `HandlerThread` is an Android `Thread` specialization that provides a `Looper`. The key parts involved in this arrangement are depicted in [figure 3.5](#).

Figure 3.5. Using the `Handler` class with separate threads, and the relationship among `HandlerThread`, `Looper`, and `MessageQueue`

MainUIThread (HandlerThread)

```
Handler myHandler = new Handler() {  
    public void handleMessage (Message m) {  
        updateUIHere();  
    }  
};  
  
new Thread() {  
    public void run() {  
        doStuff();  
        Message m = myHandler.obtainMessage();  
        Bundle b = new Bundle();  
        b.putString("key", "value");  
        m.setData(b);  
        myHandler.sendMessage(m);  
    }  
}.start();
```

Looper

MessageQueue

When you're implementing a handler, you'll need to provide a `handleMessage (Message m)` method. When you create a new thread, you can then call one of several `sendMessage` methods on `Handler` from within that thread's `run` method, as our examples and [figure 3.5](#) demonstrate. Calling `sendMessage ()` puts your message on the `MessageQueue`, which the `Looper` services.

Along with sending messages into handlers, you can also send `Runnable` objects directly, and you can schedule things to be run at different times in the future. You *send* messages and you *post* runnables. Each of these concepts supports methods such as `sendEmptyMessage (int what)`, which we've already used, and its counterparts `sendEmptyMessageAtTime (int what, long time)` and `sendEmptyMessageDelayed (int what, long delay)`. After your `Message` is in

the queue, Android will deliver it either as soon as possible or according to the requested time that you indicated.

You'll see more of `Handler` and `Message` in other examples throughout the book, and we'll cover more detail in some instances, but the main point to remember when you see these classes is that they're used to communicate between threads and for scheduling.

Getting back to our `RestaurantFinder` application and more view-oriented topics, we next need to elaborate on the `ReviewAdapter` used by our `RestaurantFinder`'s `ReviewList` screen after it's populated with data from a `Message`. This adapter returns a custom `View` object for each data element it processes.

3.2.4. Creating custom views

Although the views that are provided with Android will suffice for many apps, there might be situations where you prefer a custom view to display your own object in a unique way.

In the `ReviewList` screen, we used an adapter of type `ReviewAdapter` to back our `ListView`. This custom adapter contains a custom `View` object, `ReviewListView`. A `ReviewListView` is what our `ReviewList` Activity displays for every row of data it contains. The adapter and view are shown in the following listing.

Listing 3.5. `ReviewAdapter` and inner `ReviewListView` classes

```
public class ReviewAdapter extends BaseAdapter {
    private final Context context;
    private final List<Review> reviews;
    public ReviewAdapter(Context context, List<Review> reviews) {
        this.context = context;
        this.reviews = reviews;
    }
    @Override
    public int getCount() {
        return reviews.size();
    }
    @Override
    public Object getItem(int position) {
        return reviews.get(position);
    }
    @Override
    public long getItemId(int position) {
        return position;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        Review review = reviews.get(position);
        if (convertView == null || !(convertView instanceof ReviewListView))
        {
            return new ReviewListView(context, review.name,
                review.rating);
        }
        ReviewListView view = (ReviewListView) convertView;
        view.setName(review.name);
        view.setRating(review.rating);
        return view;
    }
    private final class ReviewListView extends LinearLayout {
        private TextView name;
        private TextView rating;
        public ReviewListView(
```

The diagram illustrates the steps to implement a custom adapter. It shows three numbered callouts pointing to specific sections of the code:

- 1 Override basic adapter**: Points to the overridden methods `getCount()`, `getItem()`, and `getItemId()`.
- 2 Override adapter getView**: Points to the implementation of the `getView()` method.
- 3 Define custom inner view class**: Points to the definition of the `ReviewListView` inner class.

```

        Context context, String itemName,
        String itemRating) {
    super(context);
    setOrientation(LinearLayout.VERTICAL);
    LinearLayout.LayoutParams params =
new LinearLayout.LayoutParams(
        ViewGroup.LayoutParams.WRAP_CONTENT,
        ViewGroup.LayoutParams.WRAP_CONTENT);
    params.setMargins(5, 3, 5, 0);
    name = new TextView(context);
    name.setText(itemName);
    name.setTextSize(16f);
    name.setTextColor(Color.WHITE);
    addView(name, params);
    rating = new TextView(context);
    rating.setText(itemRating);
    rating.setTextSize(16f);
    rating.setTextColor(Color.GRAY);
    addView(rating, params);
}
public void setName(String itemName)
{
    name.setText(itemName);
}

public void setRating(String itemRating)
{
    rating.setText(itemRating);
}
}

```

4 Set layout in code

5 Add TextView to tree

The first thing to note in `ReviewAdapter` is that it extends `BaseAdapter`. `BaseAdapter` is an `Adapter` implementation that provides basic event-handling support. `Adapter` itself is an interface in the `android.widget` package and provides a way to bind data to a view with some common methods. This is often used with collections of data, as you saw with `Spinner` and `ArrayAdapter` in [listing 3.1](#). Another common use is with a `CursorAdapter`, which returns results from a database (something you'll see in [chapter 5](#)). Here we're creating our own adapter because we want it to return a custom view.

Our `ReviewAdapter` class accepts and stores two parameters in the constructor. This class goes on to implement the required `Adapter` interface methods that return a count,

an item, and an ID; we use the ordinal position in the collection as the ID 1. The next `Adapter` method to implement is the most important: `getView()`. The adapter returns any view we create for a particular item in the collection of data that it's supporting. Within this method, we get a particular `Review` object based on the position/ID. The UI framework might call `getView()` multiple times for a given item; for

example, the UI may need to make multiple layout passes in order to determine how items will fit inside. The framework might provide an old view that we may be able to recycle for this item; doing so helps avoid wasted allocations. If there isn't a valid older view, we create an instance of a custom `ReviewListView` object to return as the view 2.

`ReviewListView` itself is an inner class inside `ReviewAdapter`; we never use it except to return a view from `ReviewAdapter` 3. Within it, you see an example of setting layout and view details in code, rather than relying on their definition in XML. In this listing, we set the orientation, parameters, and margin for our layout 4. Next, we populate the simple `TextView` objects that will be children of our new view and represent data. When these are set up via code, we add them to the parent container, which is in this case our custom class `ReviewListView` 5. This is where the data binding happens—the bridge to the view from data. Another important thing to note about this is that we've created not only a custom view, but also a composite one. We're using simple existing `View` objects in a particular layout to construct a new type of reusable view, which shows the detail of a selected `Review` object on screen, as depicted in [figure 3.2](#).

Our custom `ReviewListView` object is intentionally fairly simple. In many cases, you'll be able to create custom views by combining existing views in this manner. An alternative approach is to extend the `View` class itself. If you extend `View`, you can implement core methods as desired, and you have access to the lifecycle methods of a `View`, such as `onMeasure()`, `onLayout()`, `onDraw()`, and `onVisibilityChanged()`. You should rarely need to go to these lengths; for most apps, you can achieve your desired UI by combining preexisting `View` components, as we've done here.

Now that you've seen how you get the data for your reviews and what the adapter and custom view look like, let's take a closer look at a few more aspects of views, including layout.

3.2.5. Understanding layout

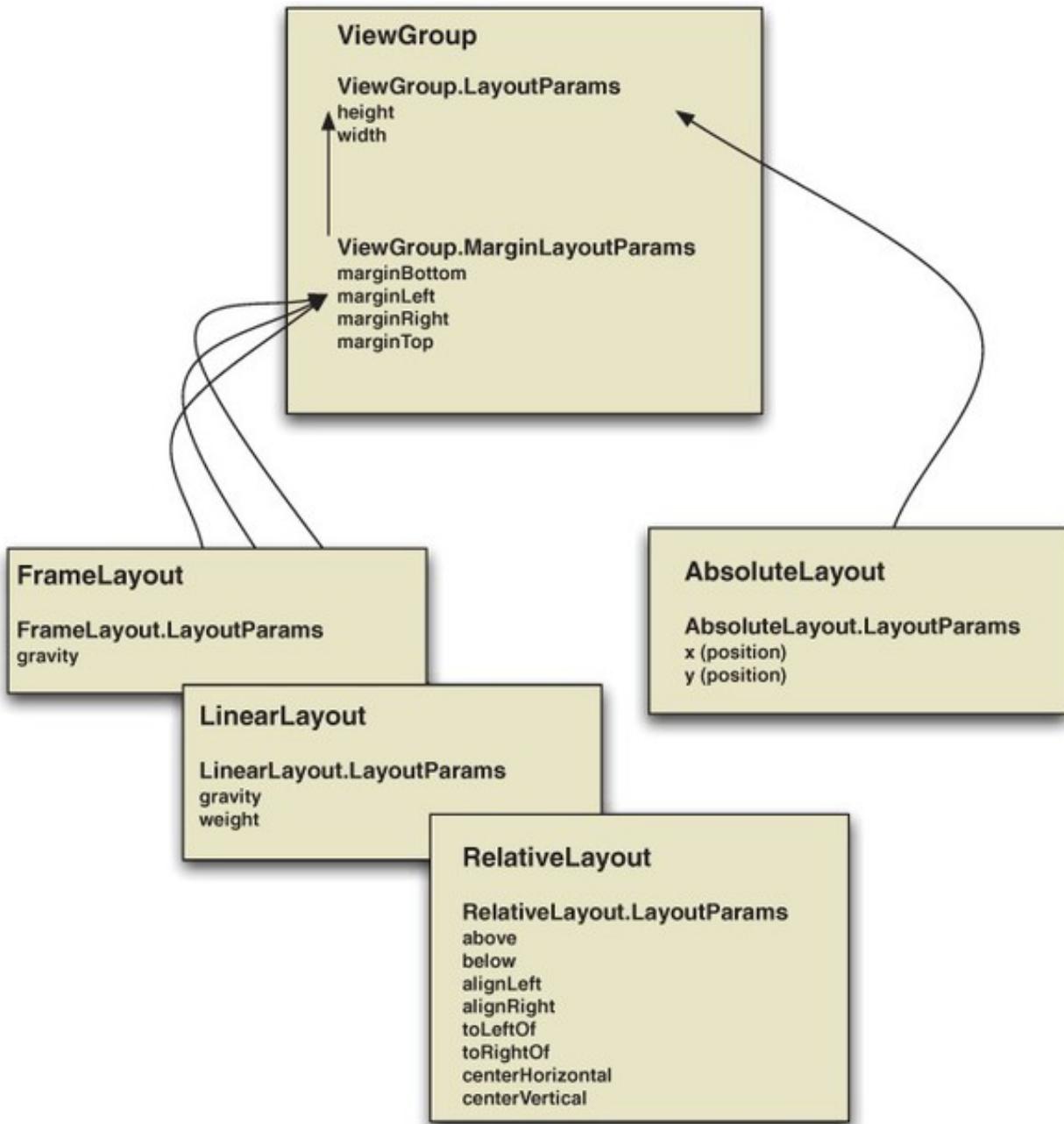
One of the most significant aspects of creating your UI and designing your screens is understanding layout. Android manages layouts through `ViewGroup` and `LayoutParams` objects. `ViewGroup` is a view that contains other views and also provides access to the layout.

On every screen, all the views are placed in a hierarchical tree; every element can have one or more children, with a `ViewGroup` at the root. All the views on the screen support a host of attributes that we addressed in [section 3.2.1](#). Dimensions—width and height—and other properties such as the margins and whether to use relative or absolute placement

are based on the `LayoutParams` a view requests and what the parent can accommodate. The final layout reflects the cumulative dimensions of the parent and its child views.

The main `ViewGroup` classes are shown in the class diagram in [figure 3.4](#). The diagram in [figure 3.6](#) expands on this class structure to show the specific `LayoutParams` inner classes of the view groups and layout properties each type provides.

Figure 3.6. Common `ViewGroup` classes with `LayoutParams` and properties provided



As figure 3.6 shows, the base `ViewGroup.LayoutParams` class supports `height` and `width`. From there, an `AbsoluteLayout` type with `AbsoluteLayout.LayoutParams` allows you to specify the exact `x` and `y` coordinates of the child `view` objects placed within. You should generally avoid the `AbsoluteLayout` because it prevents layouts from looking good on larger or smaller screen resolutions.

As an alternative to `AbsoluteLayout`, you can use the `FrameLayout`, `LinearLayout`, and `RelativeLayout` subtypes, all of which support variations of `LayoutParams` that are derived from `ViewGroup.MarginLayoutParams`. A `FrameLayout` frames one child element, such as an image. A `FrameLayout` supports multiple children, but all the items are pinned to the top left—they'll overlap each other in a stack. A `LinearLayout` aligns child elements in either a horizontal or a vertical line. Recall that we used a `LinearLayout` in our `ReviewListView` in [listing 3.5](#). There we created our view and its `LayoutParams` directly in code. Also, in our previous `Activity` examples, we used a `RelativeLayout` in our XML layout files that was inflated into our code. A `RelativeLayout` specifies child elements relative to each other: `above`, `below`, `toLeftOf`, and so on.

To summarize, the container is a `ViewGroup`, and a `ViewGroup` supports a particular type of `LayoutParams`. Child `View` elements are then added to the container and must fit into the layout specified by their parents. Even though a child view needs to lay itself out based on its parents' `LayoutParams`, it can also specify a different layout for its own children. This flexibility allows you to construct just about any type of screen you want.

The dimensions for a given view are dictated by the `LayoutParams` of its parent—so for each dimension of the layout of a view, you must define one of the following three values:

- An exact number (unit required)
- `FILL_PARENT`
- `WRAP_CONTENT`

The `FILL_PARENT` constant means “take up as much space in that dimension as the parent does (subtracting padding).” `WRAP_CONTENT` means “take up only as much space as is needed for the provided content (adding padding).” A child view requests a size, and the parent makes a decision on how to position the child view on the screen. The child makes a request, and the parent makes the decision.

Child elements do keep track of what size they're initially asked to be, in case layout is recalculated when things are added or removed, but they can't force a particular size. Because of this, `View` elements have two sets of dimensions: the size and width they want to take up (`getMeasuredWidth()` and `getMeasuredHeight()`) and the actual size they end up after a parent's decision (`getWidth()` and `getHeight()`). Layout is a two-step process: first, measurements are taken during the *measure pass*, and subsequently, the items are placed to the screen during the *layout pass*, using the associated `LayoutParams`.

Components are drawn to the screen in the order in which they're found in the layout tree: parents first, then children. Note that parent views end up behind children if they overlap in positioning.

Layout is a big part of understanding screen design with Android. Along with placing your `View` elements on the screen, you need to have a good grasp of focus and event handling in order to build effective applications.

Fragmentation

Android 3.0 has introduced a new concept, the *fragment*, which lies somewhere between a view and an `Activity`. A fragment defines a reusable user interface chunk with its own lifecycle. Fragments are most useful if you wish to present multiple “screens” at once on a larger device such as a tablet. For example, in the `RestaurantFinder`, you could represent the `ReviewCriteria` in one fragment, the `ReviewList` in another, and the `ReviewDetail` in a third. A smartphone would display one fragment at a time, but on a tablet, you could show the list of reviews in one pane and the selected review’s detail in another pane. Fragments are more complicated than standard views, but in the long run they can reduce overall maintenance in your code by letting you keep a single codebase that supports significantly different user interfaces.

3.2.6. Handling focus

Focus is like a game of tag; one and only one component on the screen is “it” at any given time. Although a particular screen can have many different windows and widgets, only one has the current focus and can respond to user input. An event, such as movement of a stylus or finger, a tap, or a keyboard press, might trigger the focus to shift to another component.

In Android, focus is handled for you by the platform a majority of the time. When a user selects an `Activity`, it’s invoked and the focus is set to the foreground `View`. Internal Android algorithms then determine where the focus should go next based on events taking place in the applications. Events might include buttons being clicked, menus being selected, or services returning callbacks. You can override the default behavior and provide hints about where specifically you want the focus to go using the following `View` class methods or their counterparts in XML:

- `nextFocusDown()`
- `nextFocusLeft()`
- `nextFocusRight()`
- `nextFocusUp()`

Views can also indicate a particular focus type, `DEFAULT_FOCUS` or `WEAK_FOCUS`, to set the priority of focus to either themselves (default) or their descendants (weak). In addition

to hints, such as `UP`, `DOWN`, and `WEAK`, you can use the `View.requestFocus()` method directly, if you need to, to indicate that focus should be set to a particular view at a given time. Manipulating the focus manually should be the exception rather than the rule—the platform logic generally does what you’d expect (and more important, what the user expects). Your application’s behavior should be mindful of how other Android applications behave and should act accordingly.

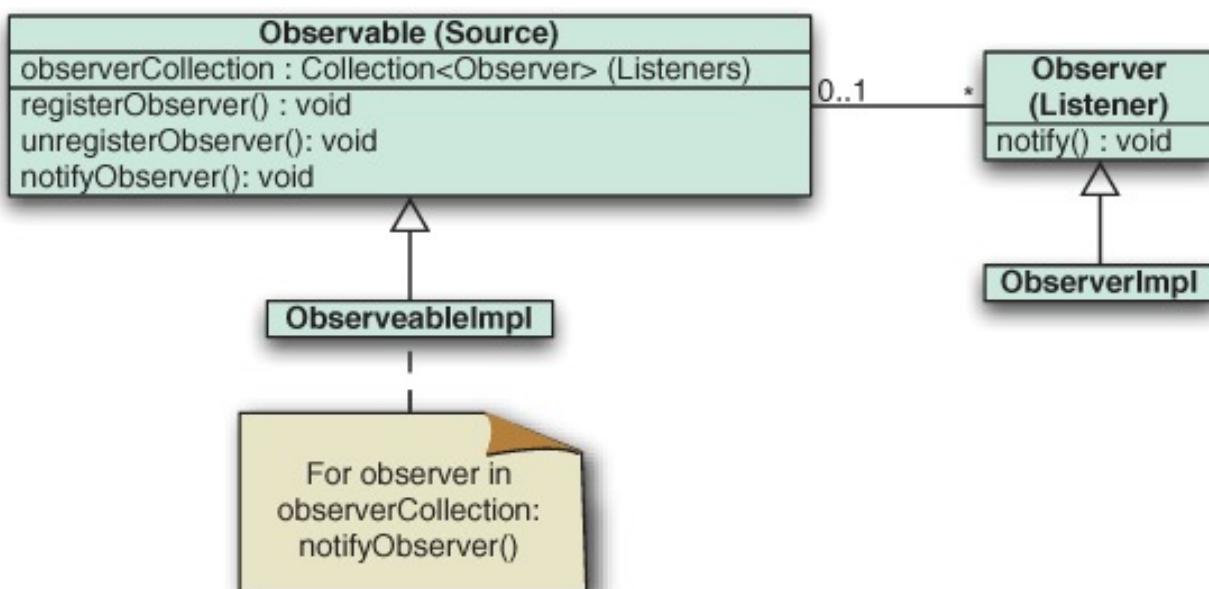
Focus changes based on event-handling logic using the `OnFocusChangeListener` object and related `setOnFocusChangedListener()` method. This brings us to the topic of event handling.

3.2.7. Grasping events

Events are used to change the focus and for many other actions. We’ve already implemented several `onClickListener()` methods for buttons in [listing 3.2](#). Those `OnClickListener` instances were connected to button presses. They indicated events that said, “Hey, somebody pressed me.” Focus events go through this same process when announcing or responding to `OnFocusChange` events.

Events have two halves: the component raising the event and the component (or components) that respond to the event. These two halves are variously known as *Observable* and *Observer* in design-pattern terms, or sometimes *subject* and *observer*. [Figure 3.7](#) is a class diagram of the relationships in this pattern.

[Figure 3.7](#). A class diagram depicting the Observer design pattern. Each `Observable` component has zero to many `Observers`, which can be notified of changes when necessary.



An `Observable` component provides a way for `Observer` instances to register. When an event occurs, the `Observable` notifies all the `Observers` that something has taken place. The `Observers` can then respond to that notification however they see fit. Interfaces are typically used for the various types of events in a particular API. An `Android Button` represents this as follows:

- `Observable—Button.setOnClickListener(OnClickListener listener)`
- `Observer—listener.onClick(View v)`

This pattern affects `Android View` items, because many things are `Observable` and allow other components to attach and listen for events. For example, most of the `View` class methods that begin with `on` are related to events: `onFocusChanged()`, `onSizeChanged()`, `onLayout()`, `onTouchEvent()`, and the like.

Events occur both within the UI and all over the platform. For example, when an incoming phone call occurs or a GPS-based location changes based on physical movement, many different reactions can occur. More than one component might want to be notified when the phone rings or when the location changes—not just the one you’re working on—and this list of `Observers` isn’t necessarily limited to UI-oriented objects.

Views support events on many levels. When an interface event occurs, such as a user pressing a button, scrolling, or selecting a portion of a window, the event is dispatched to the appropriate view. Click events, keyboard events, touch events, and focus events represent the kinds of events you’ll primarily deal with in the UI.

Remember that Android’s user interface is single-threaded. If you call a method on a view, you need to be on the UI thread. Recall that this is why we used a handler in [listing 3.3](#)—to get data outside the UI thread and to notify the UI thread to update the view after the data was retrieved. The data was sent back to the handler as a `Message` via the `setMessage()` event.

Our coverage of events in general and how they relate to layout rounds out the majority of our discussion of views, but we still have one notable related concept to discuss—resources. In the next section, we’ll address all the aspects of resources, including XML-defined views.

3.3. USING RESOURCES

You’ve already seen several examples of resources throughout the book. We’ll now explore them in detail and implement the third and final `Activity` in `RestaurantFinder`—the `ReviewDetail` screen.

When you begin working with Android, you'll quickly notice many references to a class named `R`. This class was introduced in [chapter 1](#), and we've used it in our previous `Activity` examples in this chapter. Android automatically generates this class for each of your projects to provide access to resources. Resources are noncode items that the platform automatically includes in your project.

To begin looking at resources, we'll first explore the various available types, and then we'll demonstrate examples of each type of resource.

3.3.1. Supported resource types

Each Android project's resources are located in the `res` directory. Not every project will use every type, but any resource must fit one of the available types:

- `res/anim`—XML representations of frame-by-frame animations
- `res/drawable`—Graphics such as PNG and JPG images, stretchable nine-patch images, and gradients
- `res/layout`—XML representations of `View` object hierarchies
- `res/values`—XML representations of strings, colors, styles, dimensions, and arrays
- `res/xml`—User-defined XML files that are compiled into a compact binary representation
- `res/raw`—Arbitrary and uncompiled files

Resources are treated specially in Android because they're typically compiled into an efficient binary type, with the noted exceptions of items that are already binary and the raw type, which isn't compiled. Animations, layouts and views, string and color values, and arrays can all be defined in an XML format on the platform. These XML resources are then processed by the `aapt` tool, which you saw in [chapter 2](#), and compiled. After resources have been compiled, they're accessible in Java through the automatically generated `R` class.

3.3.2. Referencing resources in Java

The first portion of the `ReviewDetail` `Activity`, shown in the following listing, reuses many of the `Activity` tenets you've already learned and uses several subcomponents that come from `R.java`.

Listing 3.6. First portion of `ReviewDetail` showing multiple uses of the `R` class

```
public class ReviewDetail extends Activity {  
    private static final int MENU_CALL REVIEW = Menu.FIRST + 2;  
    private static final int MENU_MAP REVIEW = Menu.FIRST + 1;  
    private static final int MENU_WEB REVIEW = Menu.FIRST;  
    private String imageLink;  
    private String link;  
    private TextView location;  
    private TextView name;  
    private TextView phone;  
    private TextView rating;  
    private TextView review;  
    private ImageView reviewImage;  
    private Handler handler = new Handler() {  
        public void handleMessage(Message msg) {  
            if ((imageLink != null) && !imageLink.equals("")) {  
                try {  
                    URL url = new URL(imageLink);  
                    URLConnection conn = url.openConnection();  
                    conn.connect();  
                    BufferedInputStream bis = new  
BufferedInputStream(conn.getInputStream());  
                    Bitmap bm = BitmapFactory.decodeStream(bis);  
                    bis.close();  
                    reviewImage.setImageBitmap(bm);  
                } catch (IOException e) {  
                    // log and or handle here  
                }  
            } else {  
                reviewImage.setImageResource(R.drawable.no_review_image);  
            }  
        }  
    };  
};
```

1 Define inflateable
View items

```
@Override  
public void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.review_detail);  
    name =  
        (TextView) findViewById(R.id.name_detail);  
    rating =  
        (TextView) findViewById(R.id.rating_detail);  
    location =  
        (TextView) findViewById(R.id.location_detail);  
    phone =  
        (TextView) findViewById(R.id.phone_detail);  
    review =  
        (TextView) findViewById(R.id.review_detail);  
    reviewImage =  
        (ImageView) findViewById(R.id.review_image);  
    RestaurantFinderApplication application =  
        (RestaurantFinderApplication) getApplication();  
    Review currentReview = application.getCurrentReview();  
    link = currentReview.link;  
    imageLink = currentReview.imageLink;  
    name.setText(currentReview.name);  
    rating.setText(currentReview.rating);  
    location.setText(currentReview.location);  
    review.setText(currentReview.content);  
    if ((currentReview.phone != null) &&  
        !currentReview.phone.equals("")) {  
        phone.setText(currentReview.phone);  
    } else {  
        phone.setText("NA");  
    }  
}  
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    super.onCreateOptionsMenu(menu);  
    menu.add(0, ReviewDetail.MENU_WEB REVIEW, 0,  
R.string.menu_web_review).setIcon(  
        android.R.drawable.ic_menu_info_details);  
    menu.add(0, ReviewDetail.MENU_MAP REVIEW, 1,  
R.string.menu_map_review).setIcon(  
        android.R.drawable.ic_menu_mapmode);  
    #3  
    menu.add(0, ReviewDetail.MENU_CALL REVIEW, 2,  
R.string.menu_call_review).setIcon(  
        android.R.drawable.ic_menu_call);  
  
    return true;  
}  
...  
}
```

2 Set layout using
setContentView()

3 Use String
and Drawable
resources

In the `ReviewDetail` class, we first define `View` components that we'll later reference from resources   . Next, you see a handler that's used to perform a network call to populate an `ImageView` based on a URL. (Don't worry too much about the details of the network calls here; these will be addressed in the networking sections in [chapter 5](#).) After the handler, we set the layout and view tree using `setContentView`  (`R.layout.review_detail`)  . This maps to an XML layout file at [src/res/layout/review_detail.xml](#). Next, we reference some of the `View` objects in the layout file directly through resources and corresponding IDs.

Views defined in XML are inflated by parsing the layout XML and injecting the corresponding code to create the objects for you. This process is handled automatically by the platform. All the `View` and `LayoutParams` methods we've discussed have counterpart attributes in the XML format. This inflation approach is one of the most important aspects of view-related resources, and it makes them convenient to use and reuse. We'll examine the layout file we're referring to here and the specific views it contains more closely in the next section.

You reference resources in code, as we've been doing here, using the automatically generated `R` class. The `R` class is made up of static inner classes (one for each resource type) that hold references to all of your resources in the form of an `int` value. This value is a constant pointer to an object file, by way of a resource table that's contained in a special file which is created by the `aapt` tool and used by the `R.java` file.

The last reference to resources in [listing 3.6](#) shows the creation of our menu items  . For each of these, we reference a `String` for text from our own local resources, and we also assign an icon from the `android.R.drawable` resources namespace. You can qualify resources in this way and reuse the platform drawables, which provides stock icons, images, borders, backgrounds, and so on. You'll likely want to customize much of your own applications and provide your own drawable resources. Note that the platform provides resources if you need them, and they're arguably the better choice in terms of consistency for the user, particularly if you're calling out to well-defined actions as we are here: map, phone call, and web page.

We'll cover how all the different resource types are handled in the next several sections. The first types of resources we'll look at more closely are layouts and views.

3.3.3. Defining views and layouts through XML resources

As we've noted in several earlier sections, views and layouts are often defined in XML rather than in Java code. Defining views and layouts as resources in this way makes them easier to work with, because they're decoupled from the code and in some cases reusable in across different screens.

View resource files are placed in the res/layout source directory. The root of these XML files is usually one of the `viewGroup` layout subclasses we've already discussed: `RelativeLayout`, `LinearLayout`, `FrameLayout`, and so on. Within these root elements are child XML elements that form the view/layout tree.

Resources in the res/layout directory don't have to be complete layouts. For example, you can define a single `TextView` in a layout file the same way you might define an entire tree starting from an `AbsoluteLayout`. More often, you might create a composite view that contains several interior `View` components. You might use this approach when a particularly configured view is used in multiple areas of your application. By defining it as a *standalone* resource, you can maintain it more readily over the lifetime of your project.

You can have as many XML layout/view files as you need, all defined in the res/layout directory. Each view is then referenced in code, based on the type and ID. Our layout file for the `ReviewDetail` screen—`review_detail.xml`, shown in the following listing—is referenced in the Activity code as `R.layout.review_detail`, which is a pointer to the `RelativeLayout` parent `View` object in the file.

Listing 3.7. XML layout resource file for review_detail.xml

```
<?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"
    android:gravity="center_horizontal"
    android:padding="10px"
    android.setVerticalScrollBarEnabled="true"
    >
    <ImageView android:id="@+id/review_image"
        android:layout_width="100px"
        android:layout_height="100px"
        android:layout_marginLeft="10px"
        android:layout_marginBottom="5px" />
    <TextView android:id="@+id/name_detail"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_below="@id/review_image"
        android:layout_marginLeft="10px"
        android:layout_marginBottom="5px"
        style="@style/intro_blurb" />
    <TextView android:id="@+id/rating_label_detail"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@id/name_detail"
        android:layout_marginLeft="10px"
        android:layout_marginBottom="5px"
        style="@style/label"
        android:text="@string/rating_label" />
    .
    .
    .
</RelativeLayout>
```

1 Include child element with ID

2 Reference another resource

This file uses a `RelativeLayout` as the root of the view tree. The XML also defines LayoutParams using the `android:layout_[attribute]` convention, where `[attribute]` refers to a layout attribute such as width or height. Along with layout, you can also define other view-related attributes in XML, such as `android:padding`, which is analogous to the `setPadding()` method.

After we've defined the `RelativeLayout` parent itself, we add the child `View` elements. Here we're using an `ImageView` and multiple `TextView` components. Each of the

components is given an ID using the form `android:id="@+id/[name]"` 1. When you define an ID like this, Android generates an `int` reference in the resource table and gives it your specified name. Other components can reference the ID using the friendly textual name. Never use the integer value directly, because it will change over time as your view changes. Always use the constant value defined in the `R` class.

After you've defined your views in a layout resource file and set the content view in your `Activity`, you can use the `Activity` method `findViewById()` to obtain a reference to a particular view. You can then manipulate that view in code. For example, in [listing 3.6](#) we retrieved the rating `TextView` as follows:

```
rating = (TextView) findViewById(R.id.rating_detail)
```

This provides access to the `rating_detail` element.

XML can define all the properties for a view, including the layout. Because we're using a `RelativeLayout`, we use attributes that place one view relative to another, such as `below` or `toRightOf`. To accomplish relative placement, we use

the `android:layout_below="@id/[name]"` syntax  . The `@id` syntax lets you reference other resource items from within a current resource file. Using this approach, you can reference other elements defined in the file you're currently working on or other elements defined in other resource files.

Some of our views represent labels that are shown on the screen as-is and aren't manipulated in code, such as `rating_label_detail`. Others we'll populate at runtime, such as `name_detail`; these views don't have a text value set. We do know the text for labels, which we'll apply with references to externalized strings.

You use the same syntax for styles, using the form `style="@style/[stylename]"`. Strings, styles, and colors are themselves defined as resources in another type of resource file.

3.3.4. Externalizing values

It's common practice in the programming world to externalize string literals from code. In Java, you usually use a `ResourceBundle` or a properties file to externalize values. Externalizing references to strings in this way allows the value of a component to be stored and updated separately from the component itself, away from code.

Android includes support for values resources that are subdivided into several groups: animations, arrays, styles, strings, dimensions, and colors. Each of these items is defined in a specific XML format and made available in code as references from the `R` class, just like layouts, views, and drawables. We use externalized strings in the RestaurantFinder application, as shown in the following listing for `strings.xml`.

Listing 3.8. Externalized strings for the RestaurantFinder application, strings.xml

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

```
<resources>
```

```

<string name="app_name_criteria">RestaurantFinder - Criteria</string>
<string name="app_name_reviews">RestaurantFinder - Reviews</string>
<string name="app_name_review">RestaurantFinder - Review</string>
<string name="app_short_name">Restaurants</string>
<string name="menu_get_reviews">Get reviews</string>
<string name="menu_web_review">Get full review</string>
<string name="menu_map_review">Map location</string>
<string name="menu_call_review">Call restaurant</string>
<string name="menu_change_criteria">Change review criteria</string>
<string name="menu_get_next_page">Get next page of results</string>
<string name="intro_blurb_criteria">Enter review criteria</string>
<string name="intro_blurb_detail">Review details</string>
. . .
</resources>
```

This file uses a `<string>` element with a `name` attribute for each string value we define. We used this file for the application name, menu buttons, labels, and alert validation messages. This format is known as *simple value* in Android terminology. This file is placed in source at the `res/values/strings.xml` location. In addition to strings, you can define colors and dimensions the same way.

Dimensions are placed in `dimens.xml` and defined with the `<dimen>` element: `<dimen name=dimen_name>dimen_value</dimen>`. Dimensions can be expressed in any of the following units:

- *Pixels (px)* indicate the actual number of pixels on a screen. You should generally avoid using this unit, because it might make your UI look tiny on a high-resolution screen or huge on a low-resolution screen.
- *Inches (in)* determine the physical amount of space the item will occupy. Again, use caution; one inch looks big on a handset but tiny on a tablet.
- *Millimeters (mm)* are the metric counterpart to inches.
- *Density-independent pixels (dp)* will scale automatically based on the pixel density (dots per inch, or dpi) of the screen; you should try to use this unit for most items.
- *Scaled pixels (sp)* are similar to dp but also take into account the user's preferred text size. Developers should try to use sp to describe text sizes.

Colors are defined in colors.xml and are declared with the `<color>` element: `<color name=color_name>#color_value</color>`. Color values are expressed using Red Green Blue triplet values in hexadecimal format, as in HTML. For example, solid blue is `#0000ff`. Color and dimension files are also placed in the res/values source location.

Although we haven't defined separate colors and dimensions for the RestaurantFinder application, we're using several styles, which we referenced in [listing 3.7](#). The style definitions are shown in the following listing. Unlike the string, dimension, and color resource files, which use a simplistic value structure, the style resource file has a more complex structure, including specific attributes from the `android:` namespace.

Listing 3.9. Values resource defining reusable styles, styles.xml

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

<resources>

    <style name="intro_blurb">
        <item name="android:textSize">22sp</item>
        <item name="android:textColor">#ee7620</item>
        <item name="android:textStyle">bold</item>
    </style>

    <style name="label">
        <item name="android:textSize">18sp</item>
        <item name="android:textColor">#ffffff</item>
    </style>

    <style name="edit_text">
        <item name="android:textSize">16sp</item>
        <item name="android:textColor">#000000</item>
    </style>

    . . .
</resources>
```

The Android styles approach is similar in concept to using *Cascading Style Sheets* (CSS) with HTML. You define styles in styles.xml and then reference them from other resources or code. Each `<style>` element has one or more `<item>` children that define a single setting. Styles consist of the various view settings: dimensions, colors, margins, and such. They're helpful because they facilitate easy reuse and the ability to make

changes in one place that are applied throughout your app. Styles are applied in layout XML files by associating a style name with a particular `View` component, such as `style="@style/intro_blurb"`. Note that in this case, `style` isn't prefixed with the `android:` namespace; it's a custom local style, not one provided by the platform.

Styles can be taken one step further and used as *themes*. Whereas a style refers to a set of attributes applied to a single `View` element, themes refer to a set of attributes being applied to an entire screen. Themes can be defined in the same `<style>` and `<item>` structure as styles are. To apply a theme, you associate a style with an entire `Activity`, such as `android:theme="@android:style/[styleName]"`.

Along with styles and themes, Android supports a specific XML structure for defining arrays as a resource. You can place arrays in `res/values/arrays.xml` and use them to define collections of constant values, such as the `cuisines` we used to pass to our `ArrayAdapter` back in [listing 3.1](#). The following listing shows an example of defining these arrays in XML.

Listing 3.10. Arrays.xml used for defining cuisines and ratings

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

<resources>

    <array name="cuisines">

        <item>ANY</item>
        <item>American</item>
        <item>Barbeque</item>
        <item>Chinese</item>
        <item>French</item>
        <item>German</item>
        <item>Indian</item>
        <item>Italian</item>
        <item>Mexican</item>
        <item>Thai</item>
        <item>Vegetarian</item>
        <item>Kosher</item>
    </array>
</resources>
```

Array resources use an `<array>` element with a `name` attribute and include any number of `<item>` child elements to define each array member. You can access arrays in code using the syntax shown in [listing 3.1](#):

```
String[] ratings =  
getResources().getStringArray(R.array.ratings).
```

Android resources can also support raw files and XML. Using the `res/raw` and `res/xml` directories, respectively, you can package these file types with your application and access them through either `Resources.openRawResource(int id)` or `Resources.getXml(int id)`.

The last type of resource to examine is the most complex one: the animation resource.

3.3.5. Providing animations

Animations are more complicated than other Android resources, but they're also the most visually impressive. Android allows you to define animations that can rotate, fade, move, or stretch graphics or text. Although you don't want to go overboard with a constantly blinking animated shovel, an initial splash or occasional subtle animated effect can enhance your UI.

Animation XML files go into the `res/anim` source directory. As with layouts, you reference the respective animation you want by name/ID. Android supports four types of animations:

- `<alpha>`—Defines fading, from 0.0 to 1.0 (0.0 being transparent)
- `<scale>`—Defines sizing, x and y (1.0 being no change)
- `<translate>`—Defines motion, x and y (percentage or absolute)
- `<rotate>`—Defines rotation, pivot from x and y (degrees)

In addition, Android provides several attributes that can be used with any animation type:

- `duration`—Time for the animation to complete, in milliseconds
- `startOffset`—Offset start time, in milliseconds
- `interpolator`—Used to define a velocity curve for speed of animation

The following listing shows a simple animation that you can use to scale a view.

Listing 3.11. Example of an animation defined in an XML resource, scaler.xml

```
<?xml version="1.0" encoding="utf-8"?>  
  
<scale xmlns:android="http://schemas.android.com/apk/res/android"  
       android:fromXScale="0.5"
```

```
    android:toXScale="2.0"  
  
    android:fromYScale="0.5"  
  
    android:toYScale="2.0"  
  
    android:pivotX="50%"  
  
    android:pivotY="50%"  
  
    android:startOffset="700"  
  
    android:duration="400"  
  
    android:fillBefore="false" />
```

In code, you can reference and use this animation with any `View` object (`TextView`, for example) as follows:

```
view.startAnimation(AnimationUtils.loadAnimation(this, R.anim.scaler));
```

This will scale the `View` element up in size on both the x and y axes. Although we don't have any animations in the `RestaurantFinder` sample application by default, to see this animation work, you can add the `startAnimation()` method to any `View` element in the code and reload the application.

Animations can come in handy, so you should be aware of them. We'll cover animations and other graphics topics in greater detail in [chapter 9](#).

With our journey through Android resources now complete, we're going to address the final aspect of `RestaurantFinder` that we need to cover: the `AndroidManifest.xml` manifest file, which is required for every Android application.

3.4. EXPLORING THE ANDROIDMANIFEST FILE

As you learned in [chapter 1](#), Android requires a manifest file for every application—`AndroidManifest.xml`. This file, located in the root directory of the project source, describes the application context and any supported activities, services, broadcast receivers, or content providers, as well as the requested permissions for the application. You'll learn more about services, `Intents`, and `BroadcastReceivers` in [chapter 4](#) and about content providers in [chapter 5](#). For now, the manifest file for our `RestaurantFinder` sample application, as shown in the following listing, contains only the `<application>` itself, an `<activity>` element for each screen, and several `<usespermission>` elements.

Listing 3.12. RestaurantFinder `AndroidManifest.xml` file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application android:icon="@drawable/restaurant_icon_trans"
        android:label="@string/app_short_name"
        android:name="RestaurantFinderApplication"
        android:allowClearUserData="true"
        android:theme="@android:style/Theme.Black">
        <activity android:name="ReviewCriteria"
            android:label="@string/app_short_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name="ReviewList"
            android:label="@string/app_name_reviews">
            <intent-filter>
                <category
                    android:name="android.intent.category.DEFAULT" />
                <action
                    android:name="

"com.msi.manning.restaurant.VIEW_LIST" />
            </intent-filter>
        </activity>
        <activity android:name="ReviewDetail"
            android:label="@string/app_name_review">
            <intent-filter>
                <category
                    android:name="android.intent.category.DEFAULT" />
                <action
                    android:name="

"com.msi.manning.restaurant.VIEW_DETAIL" />
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-sdk android:minSdkVersion="3"
        android:targetSdkVersion="9"></uses-sdk>
    <supports-screens largeScreens="false" xlargeScreens="false"
        anyDensity="false" />
</manifest>
```

1 Define ReviewCriteria Activity

2 Define MAIN LAUNCHER Intent filter

In the RestaurantFinder descriptor file, you first see the root `<manifest>` element declaration, which includes the application's package declaration and the Android namespace. Then you see the `<application>` element with both the `name` and `icon` attributes defined. You don't need to include the `name` attribute here unless you want to extend the default Android `Application` object to provide some

global state to your application. We took this approach so we could access the `Application` instance to store the current `Review` object. The icon is also optional; if you don't specify one, a system default will represent your application on the main menu. We highly recommended that you provide an attractive icon for your application to make it stand out.

After the application is defined, you see the child `<activity>` elements within. These elements define each `Activity` the application supports  1. As we noted when we discussed activities in general, one `Activity` in every application is defined as the entry point for the application; this `Activity` has the `<intent-filter>` action `MAIN` and category `LAUNCHER` designation  2. This tells the Android platform how to start an application from the `Launcher`, meaning this `Activity` will be placed in the main menu on the device.

After the `ReviewCriteria` `Activity`, you see another `<activity>` designation for `ReviewList`. This `Activity` also includes an `<intent-filter>`, but for our own action, `com.msi.manning.restaurant.VIEW_LIST`. This tells the platform that this `Activity` should be invoked for that `Intent`. Next, the `<uses-permission>` elements tell the platform that this application needs the `CALL_PHONE` and `INTERNET` permissions.

The `<uses-sdk>` element has grown increasingly important as Android has evolved. This element lets Android's build tools recognize which version of the SDK you intend to build with. If you want to access advanced features, either in your code or in the manifest itself, you must set a `targetSdkVersion` that supports those features. Using `targetSdkVersion` will usually restrict your app to only run on devices with that version or higher; if you want to allow running on earlier devices, you can set a lower `minSdkVersion`. When setting `minSdkVersion`, make sure that you test on that version of device, and in particular verify that you don't call any APIs that weren't present in that SDK—doing so will crash your app. In this example, we're setting `targetSdkVersion` to 9 so we can access the `xlargeScreens` property that was added in that SDK revision; because we don't call any APIs that were defined after Android 1.5, we can safely leave `minSdkVersion` at 3.

Finally, `<supports-screens>` provides some instructions to Android that tell it how to display our UI. By default, Android won't try to stretch your app to fit very large screens; as a result, apps that looked good when running on smartphones might look tiny when running on a tablet. By setting `xlargeScreens="false"`, we're telling Android that we don't offer any custom support for larger screens. This will cause Android to run our app in screen-compatibility mode, automatically scaling up the size of our screens to fill a tablet or other large device.

The RestaurantFinder sample application uses a fairly basic manifest file with three activities and a series of `Intents`. Wrapping up the description of the manifest file

completes our discussion of views, activities, resources, and working with UIs in Android.

3.5. SUMMARY

A big part of the Android platform revolves around the UI and the concepts of activities and views. In this chapter, we explored these concepts in detail and worked on a sample application to demonstrate them. In relation to activities, we addressed the concepts and methods involved, and we covered the all-important lifecycle events the platform uses to manage them. Moving on to views, we looked at common and custom types, attributes that define layout and appearance, and focus and events.

In addition, we examined how Android handles various types of resources, from simple strings to more involved animations. We also explored the `AndroidManifest.xml` application descriptor and saw how it brings all these components together to define an Android application.

This chapter has given you a good foundation for general Android UI development. Now we need to go deeper into the `Intent` and `BroadcastReceiver` classes, which form the communication layer that Android activities and other components rely on. We'll cover these items, along with longer-running `Service` processes and the Android interprocess communication (IPC) system involving the `Binder`, in [chapter 4](#), where you'll also complete the `RestaurantFinder` application.

Chapter 4. Intents and Services

This chapter covers

- Asking other programs to do work for you with `Intents`
- Advertising your capabilities with intent filters
- Eavesdropping on other apps with broadcast receivers
- Building `Services` to provide long-lived background processing
- Offering APIs to external applications through AIDL

You've already created some interesting applications that didn't require much effort to build. In this chapter, we'll dig deeper into the use of `Intent` objects and related classes to accomplish tasks. We'll expand the RestaurantFinder application from [chapter 3](#), and show you how an `Intent` can carry you from one `Activity` to another and easily link into outside applications. Next, you'll create a new weather-reporting application to demonstrate how Android handles background processes through a `Service`. We'll wrap up the chapter with an example of using the *Android Interface Definition Language* (AIDL) to make different applications communicate with one another.

We introduced the `Intent` in [chapter 1](#). An `Intent` describes something you want to do, which might be as vague as "Do whatever is appropriate for this URL" or as specific as "Purchase a flight from San Jose to Chicago for \$400." You saw several examples of working with `Intent` objects in [chapter 3](#). In this chapter, we'll look more closely at the contents of an `Intent` and how it matches with an `IntentFilter`. The RestaurantFinder app will use these concepts to display a variety of screens.

After you complete the RestaurantFinder application, we'll move on to WeatherReporter. WeatherReporter will use the Yahoo! Weather API to retrieve weather data and alerts and show them to the user. Along the way, you'll see how an `Intent` can request work outside your UI by using a `BroadcastReceiver` and a `Service`. A `BroadcastReceiver` catches broadcasts sent to any number of interested receivers. `Services` also begin with an `Intent` but work in background processes rather than UI screens.

Finally, we'll examine the mechanism for making interprocess communication (IPC) possible using `Binder` objects and AIDL. Android provides a high-performance way for different processes to pass messages among themselves.

All these mechanisms require the use of `Intent` objects, so we'll begin by looking at the details of this class.

4.1. SERVING UP RESTAURANTFINDER WITH INTENT

The mobile Android architecture looks a lot like the service-oriented architecture (SOA) that's common in server development. Each `Activity` can make an `Intent` call to get something done without knowing exactly who'll receive that `Intent`. Developers usually don't care how a particular task gets performed, only that it's completed to their requirements. As you complete the RestaurantFinder application, you'll see that you can request sophisticated tasks while remaining vague about how those tasks should get done.

`Intent` requests are *late binding*; they're mapped and routed to a component that can handle a specified task at runtime rather than at build or compile time.

One `Activity` tells the platform, "I need a map of Langtry, TX, US," and another component returns the result. With this approach, individual components are decoupled and can be modified, enhanced, and maintained without requiring changes to a larger application or system.

Let's look at how to define an `Intent` in code, how to invoke an `Intent` within an `Activity`, and how Android resolves `Intent` routing with `IntentFilter` classes. Then we'll talk about `Intents` that anyone can use because they're built into the platform.

4.1.1. Defining Intents

Suppose that you want to call a restaurant to make a reservation. When you're crafting an `Intent` for this, you need to include two critical pieces of information. An *action* is a verb describing what you want to do—in this case, make a phone call. *Data* is a noun describing the particular thing to request—in this case, the phone number. You describe the data with a `Uri` object, which we'll describe more thoroughly in the next section. You can also optionally populate the `Intent` with other information that further describes how to handle the request. [Table 4.1](#) lists all the components of an `Intent` object.

Table 4.1. `Intent` data and descriptions

Intent attribute	Description
Action	Fully qualified String indicating the action (for example, <code>android.intent.action.DIAL</code>)

Intent attribute	Description
Category	Describes where and how the Intent can be used, such as from the main Android menu or from the browser
Component	Specifies an explicit package and class to use for the Intent, instead of inferring from action, type, and categories
Data	Data to work with, expressed as a URI (for example, content://contacts/1)
Extras	Extra data to pass to the Intent in the form of a Bundle
Type	Specifies an explicit MIME type, such as text/plain or vnd.android.cursor.item/email_v2

`Intent` definitions typically express a combination of action, data, and other attributes, such as category. You combine enough information to describe the task you want done. Android uses the information you provide to resolve which class should fulfill the request.

4.1.2. Implicit and explicit invocation

Android's loose coupling allows you to write applications that make vague requests. An implicit `Intent` invocation happens when the platform determines which component should run the `Intent`. In our example of making a phone call, we don't care whether the user has the native Android dialer or has installed a third-party dialing app; we only care that the call gets made. We'll let Android resolve the `Intent` using the action, data, and category we defined. We'll explore this resolution process in detail in the next section.

Other times, you want to use an `Intent` to accomplish some work, but you want to make sure that you handle it yourself. When you open a review in `RestaurantFinder`, you don't want a third party to intercept that request and show its own review instead. In an explicit `Intent` invocation, your code directly specifies which component should handle the `Intent`. You perform an explicit invocation by specifying either the receiver's `Class` or its `ComponentName`. The `ComponentName` provides the fully qualified class name, consisting of a `String` for the package and a `String` for the class.

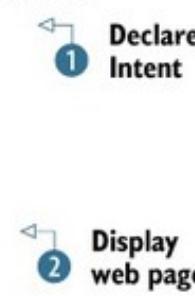
To explicitly invoke an `Intent`, you can use the following form: `Intent(Context ctx, Class cls)`. With this approach, you can short-circuit all the Android `Intent`-resolution wiring and directly pass in an `Activity` class reference to handle the `Intent`. Although this approach is convenient and fast, it also introduces tight coupling that might be a disadvantage later if you want to start using a different `Activity`.

4.1.3. Adding external links to `RestaurantFinder`

When we started the RestaurantFinder in [listing 3.6](#), we used `Intent` objects to move between screens in our application. In the following listing, we finish the `ReviewDetailActivity` by using a new set of implicit `Intent` objects to link the user to other applications on the phone.

Listing 4.1. Second section of `ReviewDetail`, demonstrating `Intent` invocation

```
@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {
    Intent intent = null;
    switch (item.getItemId()) {
        case MENU_WEB REVIEW:
            if ((link != null) && !link.equals("")) {
                intent = new Intent(Intent.ACTION_VIEW,
                    Uri.parse(link));
                startActivity(intent);
            } else {
                new AlertDialog.Builder(this)
                    .setTitle(getResources()
                        .getString(R.string.alert_label))
                    .setMessage(R.string.no_link_message)
                    .setPositiveButton("Continue",
                        new OnClickListener() {
                            public void onClick(DialogInterface dialog,
                                int arg1) {
                            }
                        }).show();
            }
            return true;
        case MENU_MAP REVIEW:
            if ((location.getText() != null)
                && !location.getText().equals("")) {
                intent = new Intent(Intent.ACTION_VIEW,
                    Uri.parse("geo:0,0?q=" +
```



```

        location.getText().toString()));
        startActivityForResult(intent);
    } else {
        new AlertDialog.Builder(this)
            .setTitle(getResources()
                .getString(R.string.alert_label))
            .setMessage(R.string.no_location_message)
            .setPositiveButton("Continue", new OnClickListener() {
                public void onClick(DialogInterface dialog,
                    int arg1) {
                }
            }).show();
    }
    return true;
case MENU_CALL_REVIEW:
    if ((phone.getText() != null)
        && !phone.getText().equals(""))
        && !phone.getText().equals("NA")) {
        String phoneString =
            parsePhone(phone.getText().toString());
        intent = new Intent(Intent.ACTION_CALL,
            Uri.parse("tel:" + phoneString));
        startActivityForResult(intent);
    } else {
        new AlertDialog.Builder(this)
            .setTitle(getResources()
                .getString(R.string.alert_label))
            .setMessage(R.string.no_phone_message)
            .setPositiveButton("Continue", new OnClickListener() {
                public void onClick(DialogInterface dialog,
                    int arg1) {
                }
            }).show();
    }
    return true;
}
return super.onOptionsItemSelected(featureId, item);
}
private String parsePhone(final String phone) {
    String parsed = phone;
    parsed = parsed.replaceAll("\\D", " ");
    parsed = parsed.replaceAll("\\s", " ");
    return parsed.trim();
}

```

3 Set Intent for map menu item

4 Set Intent for call menu item

The `Review` model object contains the address and phone number for a restaurant and a link to a full online review. Using `ReviewDetail` Activity, the user can open the menu and choose to display a map with directions to the restaurant, call the restaurant, or view the full review in a web browser. To allow all of these actions to take

place, `ReviewDetail` launches built-in Android applications through implicit `Intent` calls.

In our new code, we initialize an `Intent` class instance ① so it can be used later by the menu cases. If the user selects the `MENU_WEB REVIEW` menu button, we create a new instance of the `Intent` variable by passing in an action and data. For the action, we use the String constant `Intent.ACTION_VIEW`, which has the value `android.app.action.VIEW`. You can use either the constant or the value, but sticking to constants helps ensure that you don't mistype the name. Other common actions are `Intent.ACTION_EDIT`, `Intent.ACTION_INSERT`, and `Intent.ACTION_DELETE`.

For the data component of the `Intent`, we use `Uri.parse(link)` to create a `URI`. We'll look at `Uri` in more detail in the next section; for now, just know that this allows the correct component to answer the `startActivity(Intent i)` request ② and render the resource identified by the `URI`. We don't directly declare any particular `Activity` or `Service` for the `Intent`; we simply say we want to view <http://somehost/somepath>. Android's late-binding mechanism will interpret this request at runtime, most likely by launching the device's built-in browser.

`ReviewDetail` also handles the `MENU_MAP REVIEW` menu item. We initialize the `Intent` to use `Intent.ACTION_VIEW` again, but this time with a different type of `URI`: "geo:0,0?q=" ④ + `street_address` ④. This combination of `VIEW` and `geo` invokes a different `Intent`, probably the built-in maps application. Finally, when handling `MENU_MAP_CALL`, we request a phone call using the `Intent.ACTION_CALL` action and the `tel:Uri` scheme ④.

Through these simple requests, our `RestaurantFinder` application uses implicit `Intent` invocation to allow the user to phone or map the selected restaurant or to view the full review web page. These menu buttons are shown in [figure 4.1](#).

Figure 4.1. Menu buttons on the `RestaurantFinder` sample application that invoke external applications



The RestaurantFinder application is now complete. Users can search for reviews, select a particular review from a list, display a detailed review, and use additional built-in applications to find out more about a selected restaurant.

You'll learn more about all the built-in apps and action-data pairs in [section 4.1.5](#). Right now, we're going to focus on the Intent-resolution process and how it routes requests.

4.1.4. Finding your way with Intent

RestaurantFinder makes requests to other applications by using Intent invocations, and guides its internal movement by listening for Intent requests. Three types of Android components can register to handle Intent requests: Activity, BroadcastReceiver, and Service. They advertise their capabilities through the <intent-filter> element in the `AndroidManifest.xml` file.

Android parses each `<intent-filter>` element into an `IntentFilter` object. After Android installs an .apk file, it registers the application's components, including the Intent filters. When the platform has a registry of Intent filters, it can map any Intent requests to the correct installed Activity, BroadcastReceiver, or Service.

To find the appropriate handler for an Intent, Android inspects the action, data, and categories of the Intent. An `<intent-filter>` must meet the following conditions to be considered:

- The action and category must match.
- If specified, the data type must match, or the combination of data scheme and authority and path must match.

Let's look at these components in more detail.

Actions and Categories

Each individual `IntentFilter` can specify zero or more actions and zero or more categories. If no action is specified in the `IntentFilter`, it'll match any Intent; otherwise, it'll match only if the Intent has the same action.

An `IntentFilter` with no categories will match *only* an Intent with no categories; otherwise, an `IntentFilter` must have at least what the Intent specifies. For example, if an `IntentFilter` supports both the HOME and the ALTERNATIVE categories, it'll match an Intent for either HOME or CATEGORY. But if the `IntentFilter` doesn't provide any categories, it won't match HOME or CATEGORY.

You can work with actions and categories without specifying any data. We used this technique in the `ReviewList` Activity we built in [chapter 3](#). In that example, we defined the `IntentFilter` in the manifest XML, as shown in the following listing.

Listing 4.2. Manifest declaration of `ReviewList` Activity with `<intent-filter>`

```
<activity android:name="ReviewList" android:label="@string/app_name">

    <intent-filter>

        <category android:name="android.intent.category.DEFAULT" />

        <action android:name="com.msi.manning.restaurant.VIEW_LIST" />

    </intent-filter>

</activity>
```

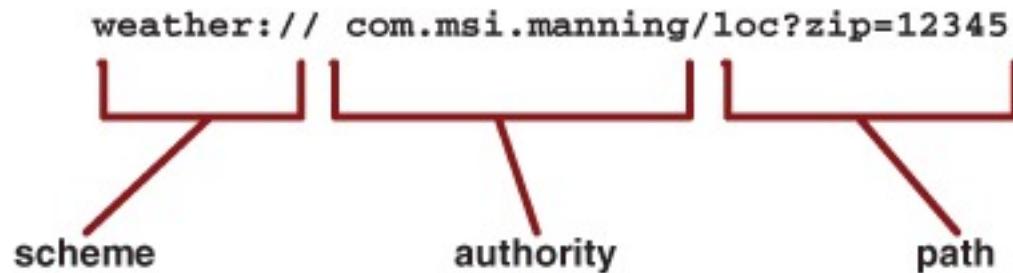
To match the filter declared in this listing, we used the following `Intent` in code, where `Constants.INTENT_ACTION_VIEW_LIST` is the String "`com.msi.manning.restaurant.VIEW_LIST`":

```
Intent intent = new Intent(Constants.INTENT_ACTION_VIEW_LIST);  
startActivity(intent);
```

Data

After Android has determined that the action and category match, it inspects the `Intent` data. The data can be either an explicit MIME type or a combination of scheme, authority, and path. The `Uri` shown in [figure 4.2](#) is an example of using scheme, authority, and path.

[Figure 4.2. The portions of a URI that are used in Android, showing scheme, authority, and path](#)



The following example shows what using an explicit MIME type within a URI looks like:

```
audio/mpeg
```

`IntentFilter` classes describe what combination of type, scheme, authority, and path they accept. Android follows a detailed process to determine whether an `Intent` matches:

- If a scheme is present and type is *not* present, `Intents` with any type will match.
- If a type is present and scheme is *not* present, `Intents` with any scheme will match.
- If neither a scheme nor a type is present, only `Intents` with neither scheme nor type will match.
- If an authority is specified, a scheme must also be specified.
- If a path is specified, a scheme and an authority must also be specified.

Most matches are straightforward, but as you can see, it can get complicated. Think of `Intent` and `IntentFilter` as separate pieces of the same puzzle. When you call an `Intent` in an Android application, the system resolves the `Activity`, `Service`, or `BroadcastReceiver` to handle your request through this process using the actions, categories, and data provided. The system searches all the pieces of the puzzle it has

until it finds one that meshes with the `Intent` you've provided, and then it snaps those pieces together to make the late-binding connection.

Figure 4.3 shows an example of how a match occurs. This example defines an `IntentFilter` with an action and a combination of a scheme and an authority. It doesn't specify a path, so any path will match. The figure also shows an example of an `Intent` with a URI that matches this filter.

Figure 4.3. Example `Intent` and `IntentFilter` matching using a filter defined in XML



If multiple `IntentFilter` classes match the provided `Intent`, the platform chooses which one will handle the `Intent`. For a user-visible action such as an `Activity`, Android usually presents the user with a pop-up menu that lets them select which `Intent` should handle it. For nonvisible actions such as a broadcast, Android considers the declared priority of each `IntentFilter` and gives them an ordered chance to handle the `Intent`.

4.1.5. Taking advantage of Android-provided activities

In addition to the examples in the `RestaurantFinder` application, Android ships with a useful set of core applications that allow access via the formats shown in table 4.2. Using these actions and URIs, you can hook into the built-in maps application, phone application, or browser application. By experimenting with these, you can get a feel for how `Intent` resolution works in Android.

Table 4.2. Common Android application `Intent`, action, and URI combinations

Action	URI	Description
Intent.ACTION_CALL tel:phone_number		Opens the phone application and calls the specified number
Intent.ACTION_DIAL tel:phone_number		Opens the phone application and dials (but doesn't call) the specified number
Intent.ACTION_DIAL voicemail:		Opens the phone application and dials (but doesn't call) the voicemail number
Intent.ACTION_VIEW geo:latitude,longitude		Opens the maps application to the specified latitude and longitude
Intent.ACTION_VIEW geo:o,o?q=street+address		Opens the maps application to the specified address
Intent.ACTION_VIEW http://web_address		Opens the browser application to the specified URL
Intent.ACTION_VIEW https://web_address		Opens the browser application to the specified secure URL

With a handle on the basics of `Intent` resolution and a quick look at built-in `Intents` out of the way, we can move on to a new sample application: WeatherReporter.

4.2. CHECKING THE WEATHER WITH A CUSTOM URI

WeatherReporter, the next sample application we'll build, uses the Yahoo! Weather API to retrieve weather data and then displays the data to the user. This application can also optionally alert users about severe weather for certain locations, based either on the current location of the device or on a specified postal code.

Within this project, you'll see how you can define a custom URI and register it with a matching `Intent` filter to allow any other application to invoke a weather report through an `Intent`. Defining and publishing an `Intent` in this way allows other applications to easily use your application. When the WeatherReporter application is complete, the main screen will look like [figure 4.4](#).

Figure 4.4. The WeatherReporter application, showing the weather forecast for the current location



4.2.1. Offering a custom URI

Let's look more deeply into how to define Intent filters in XML. The manifest for WeatherReporter is shown in the following listing.

Listing 4.3. Android manifest file for the WeatherReporter application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.weather">
    <application android:icon="@drawable/weather_sun_clouds_120"
        android:label="@string/app_name"
        android:theme="@android:style/Theme.Black"
        android:allowClearUserData="true">
        <activity android:name="ReportViewSavedLocations"
            android:label="@string/app_name_view_saved_locations" />
        <activity android:name="ReportSpecifyLocation"
            android:label=
                "@string/app_name_specify_location" />
        <activity android:name="ReportViewDetail"
            android:label="@string/app_name_view_detail">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="weather"
                    android:host="com.msi.manning" />
            </intent-filter>
            <intent-filter>
```

Define activities

1

```

        <action android:name="android.intent.action.VIEW" />
        <data android:scheme="weather"
              android:host="com.msi.manning" />
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name=
                  "android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<receiver android:name=
          ".service.WeatherAlertServiceReceiver">           ← ② Define receiver
    <intent-filter>
        <action android:name=
                  "android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
<service
      android:name=".service.WeatherAlertService" />   ← ③ Define Service
</application>
<uses-permission
      android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
<uses-permission
      android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name=
                  "android.permission.ACCESS_FINE_LOCATION" />           ← ④ Include necessary
<uses-permission android:name=
                  "android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />   permissions
<uses-permission android:name="android.permission.INTERNET" />
</manifest>

```

In the WeatherReporter manifest, we define three activities ①. The first two don't include an `<intent-filter>`, so they can only be explicitly invoked from within this application. The `ReportViewDetail` Activity has multiple `<intent-filter>` tags defined for it, including one denoting it as the `MAIN LAUNCHER` and one with the `weather://com.msi.manning` scheme and authority. Our application supports this custom URI to provide weather access.

You can use any combination of scheme, authority, and path, as shown in [listing 4.3](#), or you can use an explicit MIME type. You'll find out more about MIME types and how they're processed in [chapter 5](#), where we'll look at data sources and use an Android concept known as a `ContentProvider`.

After we define these activities, we use the `<receiver>` element in the manifest file to refer to a `BroadcastReceiver` class ②. We'll examine `BroadcastReceiver` more closely in [section 4.3](#), but for now know that an `<intent-filter>` associates this receiver with

an Intent—in this case, for the `BOOT_COMPLETED` action. This filter tells the platform to invoke the `WeatherAlertServiceReceiver` class after it completes the bootup sequence.

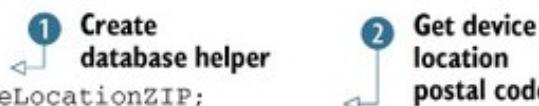
We also define a service **3**. You'll see how this service is built, and how it polls for severe weather alerts in the background, in [section 4.3](#). Finally, our manifest includes a set of required permissions **4**.

4.2.2. Inspecting a custom URI

With the foundation for our sample application in place via the manifest, Android will launch `WeatherReporter` when it encounters a request that uses our custom URI. As usual, it'll invoke the `onStart()` method of the main `Activity` `WeatherReporter` will use. The following listing shows our implementation, where we parse data from the URI and use it to display a weather report.

Listing 4.4. `onStart()` method of the `ReportViewDetail` Activity

```
@Override
public void onStart() {
    super.onStart();
    dbHelper = new DBHelper(this);
    deviceZip = WeatherAlertService.deviceLocationZIP;
    if ((getIntent().getData() != null)
        && (getIntent().getData().getEncodedQuery() != null)
        && (getIntent().getData().getEncodedQuery().length() > 8)) {
        String queryString =
            getIntent().getData().getEncodedQuery();
        reportZip = queryString.substring(4, 9);
        useDeviceLocation = false;
    } else {
        reportZip = deviceZip;
        useDeviceLocation = true;
    }
    savedLocation = dbHelper.get(reportZip);
    deviceAlertEnabledLocation =
        dbHelper.get(DBHelper.DEVICE_ALERT_ENABLED_ZIP);
```



```
if (useDeviceLocation) {
    currentCheck.setText(R.string.view_checkbox_current);
    if (deviceAlertEnabledLocation != null) {
        currentCheck.setChecked(true);
    } else {
        currentCheck.setChecked(false);
    }
} else {
    currentCheck.setText(R.string.view_checkbox_specific);
    if (savedLocation != null) {
        if (savedLocation.alertenabled == 1) {
            currentCheck.setChecked(true);
        } else {
            currentCheck.setChecked(false);
        }
    }
}
loadReport(reportZip);
}
```

Set status of alert-enabled check box ③

You can get the complete `ReportViewDetail Activity` from the source code download for this chapter. In the `onStart()` method shown in this listing, we focus on parsing data from the URI passed in as part of the `Intent` that invokes the `Activity`.

First, we establish a database helper object ①. This object will be used to query a local SQLite database that stores user-specified location data. We'll show more about how data is handled, and the details of this helper class, in [chapter 5](#).

In this method, we also obtain the postal code of the current device location from a `LocationManager` in the `WeatherAlertService` class ②. We want to use the location of the device as the default weather report location. As the user travels with the phone, this location will automatically update. We'll cover location and `LocationManager` in [chapter 11](#).

After obtaining the device location, we move on to the key aspect of obtaining URI data from an `Intent`. We check whether our `Intent` provided specific data; if so, we parse the URI passed in to obtain the `queryString` and embedded postal code to use for the user's specified location. If this location is present, we use it; if not, we default to the device location postal code.

After determining the postal code to use, we set the status of the check box that indicates whether to enable alerts ③. We have two kinds of alerts: one for the device location and another for the user's specified saved locations.

Finally, we call the `loadReport()` method, which makes the call to the Yahoo! Weather API to obtain data; then we use a `Handler` to send a `Message` to update the needed UI `View` elements.

Remember that this `Activity` registered in the manifest to receive `weather://com.msi.manning` `Intents`. Any application can invoke this `Activity` without knowing any details other than the URI. This separation of responsibilities enables late binding. After invocation, we check the URI to see what our caller wanted.

You've now seen the manifest and pertinent details of the main `Activity` class for the `WeatherReporter` application we'll build in the next few sections. We've also discussed how `Intent` and `IntentFilter` classes work together to wire up calls between components. Next, we'll look at some of the built-in Android applications that accept external `Intent` requests. These requests enable you to launch activities by simply passing in the correct URI.

4.3. CHECKING THE WEATHER WITH BROADCAST RECEIVERS

So far, you've seen how to use an `Intent` to communicate within your app and to issue a request that another component will handle. You can also send an `Intent` to any interested receiver. When you do, you aren't requesting the execution of a specific task, but instead you're letting everyone know about something interesting that has happened. Android sends these broadcasts for several reasons, such as when an incoming phone call or text message is received. In this section, we'll look at how events are broadcast and how they're captured using a `BroadcastReceiver`.

We'll continue to work through the `WeatherReporter` sample application we began in [section 4.2](#). The `WeatherReporter` application will display alerts to the user when severe weather is forecast for the user's indicated location. We'll need a background process that checks the weather and sends any needed alerts. This is where the Android `Service` concept will come into play. We need to start the `Service` when the device boots, so we'll listen for the boot through an `Intent` broadcast.

4.3.1. Broadcasting Intent

As you've seen, `Intent` objects let you move from `Activity` to `Activity` in an Android application, or from one application to another. `Intents` can also broadcast events to any configured receiver using one of several methods available from the `Context` class, as shown in [table 4.3](#).

Table 4.3. Methods for broadcasting `Intents`

Method	Description
sendBroadcast(Intent intent)	Simple form for broadcasting an Intent.
sendBroadcast(Intent intent, String receiverPermission)	Broadcasts an Intent with a permission String that receivers must declare in order to receive the broadcast.
sendOrderedBroadcast(Intent intent, String receiverPermission)	Broadcasts an Intent call to the receivers one by one serially, stopping after a receiver consumes the message.
sendOrderedBroadcast(Intent intent, String receiverPermission, BroadcastReceiver resultReceiver, Handler scheduler, int initialCode, String initialData, Bundle initialExtras)	Broadcasts an Intent and gets a response back through the provided BroadcastReceiver. All receivers can append data that will be returned in the BroadcastReceiver. When you use this method, the receivers are called serially.
sendStickyBroadcast(Intent intent)	Broadcasts an Intent that remains a short time after broadcast so that receivers can retrieve data. Applications using this method must declare the BROADCAST_STICKY permission.

When you broadcast `Intents`, you send an event into the background. A broadcast `Intent` doesn't invoke an `Activity`, so your current screen usually remains in the foreground.

You can also optionally specify a permission when you broadcast an `Intent`. Only receivers that have declared that permission will receive the broadcast; all others will remain unaware of it. You can use this mechanism to ensure that only certain trusted applications can listen in on what your app does. You can review permission declarations in [chapter 1](#).

Broadcasting an `Intent` is fairly straightforward; you use the `Context` object to send it, and interested receivers catch it. Android provides a set of platform-related `Intent` broadcasts that use this approach. In certain situations, such as when the time zone on the platform changes, when the device completes booting, or when a package is added or removed, the system broadcasts an event using an `Intent`. [Table 4.4](#) shows some of the specific `Intent` broadcasts the platform provides.

Table 4.4. Broadcast actions provided by the Android platform

Action	Description
ACTION_BATTERY_CHANGED	Sent when the battery charge level or charging state changes
ACTION_BOOT_COMPLETED	Sent when the platform completes booting
ACTION_PACKAGE_ADDED	Sent when a package is added to the platform
ACTION_PACKAGE_REMOVED	Sent when a package is removed from the platform
ACTION_TIME_CHANGED	Sent when the user changes the time on the device
ACTION_TIME_TICK	Sent every minute to indicate that time is ticking
ACTION_TIMEZONE_CHANGED	Sent when the user changes the time

To register to receive an `Intent` broadcast, you implement a `BroadcastReceiver`. You'll make your own implementation to catch the platform-provided `BOOT_COMPLETED` `Intent` to start the weather alert service.

4.3.2. Creating a receiver

Because the weather alert `Service` you're going to create should always run in the background, you need a way to start it when the platform boots. To do this, you'll create a `BroadcastReceiver` that listens for the `BOOT_COMPLETED` `Intent` broadcast.

The `BroadcastReceiver` base class provides a series of methods that let you get and set a result code, result data (in the form of a `String`), and an extra `Bundle`. It also defines a lifecycle-related method to run when the appropriate `Intent` is received.

You can associate a `BroadcastReceiver` with an `IntentFilter` in code or in the manifest XML file. We declared this for the `WeatherReporter` manifest in [listing 4.3](#), where we associated the `BOOT_COMPLETED` broadcast with the `WeatherAlertServiceReceiver` class. This class is shown in the following listing.

Listing 4.5. `WeatherAlertServiceReceiver` `BroadcastReceiver` class

```
public class WeatherAlertServiceReceiver extends BroadcastReceiver {

    @Override

    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals(Intent.ACTION_BOOT_COMPLETED)) {
            context.startService(new Intent(context,
                WeatherAlertService.class));
        }
    }
}
```

```
    }  
}
```

When you create your own `Intent` broadcast receiver, you extend the `BroadcastReceiver` class and implement the abstract `onReceive(Context c, Intent i)` method. In our implementation, we start the `WeatherAlertService`. This `Service` class, which we'll create next, is started using the `Context.startService(Intent i, Bundle b)` method.

Keep in mind that receiver class instances have a short and focused lifecycle. After completing the `onReceive(Context c, Intent i)` method, the instance and process that invoked the receiver are no longer needed and might be killed by the system. For this reason, you can't perform any asynchronous operations in a `BroadcastReceiver`, such as starting a thread or showing a dialog. Instead, you can start a `Service`, as we've done in [listing 4.5](#), and use it to do work.

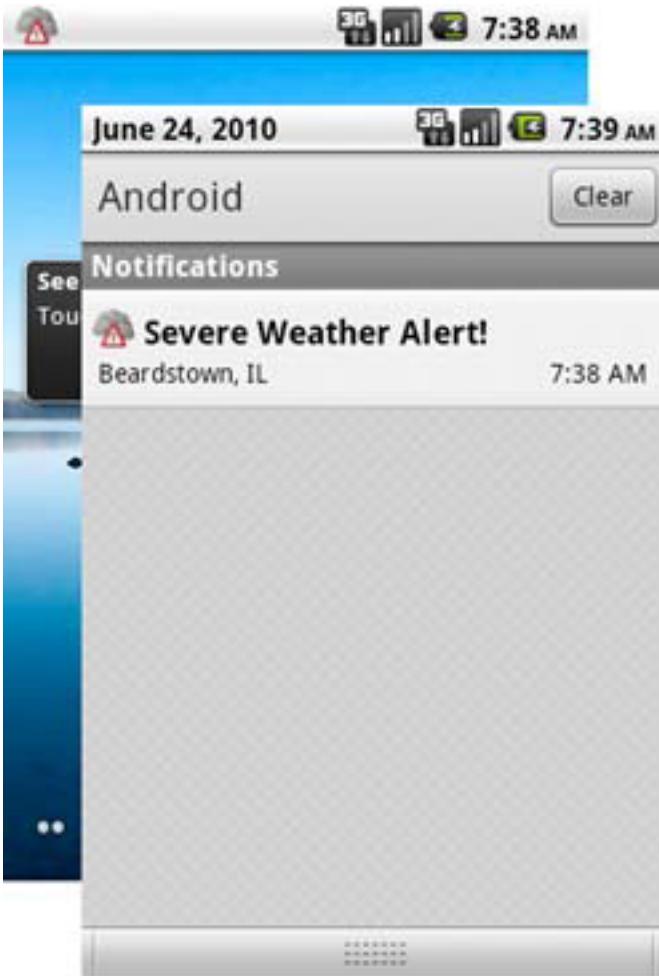
Our receiver has started the `WeatherAlertService`, which will run in the background and warn users of severe weather in the forecast with a `Notification`-based alert. Let's look more deeply into the concept of an Android `Service`.

4.4. BUILDING A BACKGROUND WEATHER SERVICE

In a basic Android application, you create `Activity` classes and move from screen to screen using `Intent` calls, as we've done in previous chapters. This approach works for the canonical Android screen-to-screen foreground application, but it doesn't work for cases like ours where we want to always listen for changes in the weather, even if the user doesn't currently have our app open. For this, we need a `Service`.

In this section, we'll implement the `WeatherAlertService` we launched in [listing 4.4](#). This `Service` sends an alert to the user when it learns of severe weather in a specified location. This alert will display over any application, in the form of a `Notification`, if severe weather is detected. [Figure 4.5](#) shows the notification we'll send.

Figure 4.5. Warning from a background application about severe weather



A background task is typically a process that doesn't involve direct user interaction or any type of UI. This process perfectly describes checking for severe weather. After a Service is started, it runs until it's explicitly stopped or the system kills it. The `WeatherAlertService` background task, which starts when the device boots via the `BroadcastReceiver` from [listing 4.5](#), is shown in the following listing.

Listing 4.6. `WeatherAlertService` class, used to register locations and send alerts

```
public class WeatherAlertService extends Service {
    private static final String LOC = "LOC";
    private static final String ZIP = "ZIP";
    private static final long ALERT QUIET_PERIOD = 10000;
    private static final long ALERT POLL INTERVAL = 15000;
    public static String deviceLocationZIP = "94102";
    private Timer timer;
    private DBHelper dbHelper;
    private NotificationManager nm;
    private TimerTask task = new TimerTask() {
        public void run() {
            List<Location> locations =
                dbHelper.getAllAlertEnabled();
            for (Location loc : locations) {
                WeatherRecord record = loadRecord(loc.zip);
                if (record.isSevere()) {
                    if ((loc.lastalert +
                        WeatherAlertService.ALERT QUIET_PERIOD)
                        < System.currentTimeMillis()) {
                        loc.lastalert = System.currentTimeMillis();
                        dbHelper.update(loc);
                        sendNotification(loc.zip, record);
                    }
                }
            }
            . . . device location alert omitted for brevity
        }
    };
    private Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            notifyFromHandler((String) msg.getData()
                .get(WeatherAlertService.LOC), (String) msg.getData()
                .get(WeatherAlertService.ZIP));
        }
    };
    @Override
    public void onCreate() {
        dbHelper = new DBHelper(this);
        timer = new Timer();           ← 4 Initialize timer
        timer.schedule(task, 5000,
            WeatherAlertService.ALERT POLL INTERVAL);
        nm = (NotificationManager)
            getSystemService(Context.NOTIFICATION_SERVICE);
    }
}
```

1 Get locations with alerts enabled

2 Fire alert if severe

3 Notify UI from handler

```

. . . onStart with LocationManager and LocationListener \
      omitted for brevity
@Override
public void onDestroy() {
    super.onDestroy();
    dbHelper.cleanup();
```

5 Clean up database connection

```

}
@Override
public IBinder onBind(Intent intent) {
    return null;
}
protected WeatherRecord loadRecord(String zip) {
    final YWeatherFetcher ywh =
        new YWeatherFetcher(zip, true);
    return ywh.getWeather();
}
private void sendNotification(String zip,
    WeatherRecord record) {
    Message message = Message.obtain();
    Bundle bundle = new Bundle();
    bundle.putString(WeatherAlertService.ZIP, zip);
    bundle.putString(WeatherAlertService.LOC, record.getCity()
        + ", " + record.getRegion());
    message.setData(bundle);
    handler.sendMessage(message);
}
private void
notifyFromHandler(String location, String zip) {
```

6 Display actionable notification

```

    Uri uri = Uri.parse("weather://com.msi.manning/loc?zip=" + zip);
    Intent intent = new Intent(Intent.ACTION_VIEW, uri);
    PendingIntent pendingIntent =
    PendingIntent.getActivity(this, Intent.FLAG_ACTIVITY_NEW_TASK,
        intent, PendingIntent.FLAG_ONE_SHOT);
    final Notification n =
        new Notification(R.drawable.severe_weather_24,
            "Severe Weather Alert!",
            System.currentTimeMillis());
    n.setLatestEventInfo(this, "Severe Weather Alert!",
        location, pendingIntent);
    nm.notify(Integer.parseInt(zip), n);
}
```

}

`WeatherAlertService` extends `Service`. We create a `Service` in a way that's similar to how we've created activities and broadcast receivers: extend the base class, implement the abstract methods, and override the lifecycle methods as needed.

After the initial class declaration, we define several member variables. First come constants that describe our intervals for polling for severe weather and a quiet period. We've set a low threshold for polling during development—severe weather alerts will spam the emulator often because of this setting. In production, we'd limit this to check every few hours.

Next, our `TimerTask` variable will let us periodically poll the weather. Each time the task runs, it gets all the user's saved locations through a database call 1. We'll examine the specifics of using an Android database in [chapter 5](#).

When we have the saved locations, we parse each one and load the weather report. If the report shows severe weather in the forecast, we update the time of the last alert field and call a helper method to initiate sending a `Notification` 2. After we process the user's saved locations, we get the device's alert location from the database using a postal code designation. If the user has requested alerts for their current location, we repeat the process of polling and sending an alert for the device's current location as well. You can see more details on Android location-related facilities in [chapter 11](#).

After defining our `TimerTask`, we create a `Handler` member variable. This variable will receive a `Message` object that's fired from a non-UI thread. In this case, after receiving the `Message`, our `Handler` calls a helper method that instantiates and displays a `Notification` 3.

Next, we override the `Service` lifecycle methods, starting with `onCreate()`. Here comes the meat of our `Service`: a `Timer` 4 that we configure to repeatedly fire. For as long as the `Service` continues to run, the timer will allow us to update weather information.

After `onCreate()`, you see `onDestroy()`, where we clean up our database connection 5. `Service` classes provide these lifecycle methods so you can control how resources are allocated and deallocated, similar to `Activity` classes.

After the lifecycle-related methods, we implement the required `onBind()` method. This method returns an `IBinder`, which other components that call into `Service` methods will use for communication. `WeatherAlertService` performs only a background task; it doesn't support binding, and so it returns a `null` for `onBind`. We'll add binding and interprocess communication (IPC) in [section 4.5](#).

Next, we implement our helper methods. First, `loadRecord()` calls out to the Yahoo! Weather API via `YWeatherFetcher`. (We'll cover networking tasks, similar to those this class performs, in [chapter 6](#).) Then `sendNotification` configures a `Message` with location details to activate the `Handler` we declared earlier. Last of all, you see the `notifyFromHandler()` method. This method fires off

a `Notification` with `Intent` objects that will call back into the `WeatherReporter Activity` if the user clicks the `Notification` .

A warning about long-running Services

Our sample application starts a `Service` and leaves it running in the background. This `Service` is designed to have a minimal footprint, but Android best practices discourage long-running `Services`. `Services` that run continually and constantly use the network or perform CPU-intensive tasks will eat up the device's battery life and might slow down other operations. Even worse, because they run in the background, users won't know what applications are to blame for their device's poor performance.

The OS will eventually kill running `Services` if it needs to acquire additional memory, but otherwise it won't interfere with poorly designed `Services`. If your use case no longer requires the `Service`, you should stop it. If you do require a long-running `Service`, you might want to give users the option of whether to use it.

Now that we've discussed the purpose of `Services` and you've created a `Service` class and started one via a `BroadcastReceiver`, we can start looking at how other developers can interact with your `Service`.

4.5. COMMUNICATING WITH THE WEATHERALERTSERVICE FROM OTHER APPS

In Android, each application runs within its own process. Other applications can't directly call methods on your weather alert service, because the applications are in different sandboxes. You've already seen how applications can invoke one another by using an `Intent`. Suppose, though, that you wanted to learn something specific from a particular application, like check the weather in a particular region. This type of granular information isn't readily available through simple `Intent` communication, but fortunately Android provides a new solution: IPC through a *bound service*.

We'll illustrate bound services by expanding the weather alert with a remotable interface using AIDL, and then we'll connect to that interface through a proxy that we'll expose using a new `Service`. Along the way, we'll explore the `IBinder` and `Binder` classes Android uses to pass messages and types during IPC.

4.5.1. Android Interface Definition Language

If you want to allow other developers to use your weather features, you need to give them information about the methods you provide, but you might not want to share your application's source code. Android lets you specify your IPC features by using an interface definition language (IDL) to create AIDL files. These files generate a Java interface and an inner `Stubclass` that you can use to create a remotely accessible object, and that your consumers can use to invoke your methods.

AIDL files allow you to define your package, imports, and methods with return types and parameters. Our weather AIDL, which we place in the same package as the `.java` files, is shown in the following listing.

Listing 4.7. IWeatherReporter.aidl remote IDL file

```
package com.msi.manning.weather;

interface IWeatherReporter
{
    String getWeatherFor(in String zip);
    void addLocation(in String zip, in String city, in String region);
}
```

You define the package and interface in AIDL as you would in a regular Java file. Similarly, if you require any imports, you'd list them above the interface declaration. When you define methods, you must specify a directional tag for all nonprimitive types. The possible directions are `in`, `out`, and `inout`. The platform uses this directional tag to generate the necessary code for marshaling and unmarshaling instances of your interface across IPC boundaries.

Our interface `IWeatherReporter` includes methods to look up the current weather from the `Service`, or to add a new location to the `Service`. Other developers could use these features to provide other front-end applications that use our back-end service.

Only certain types of data are allowed in AIDL, as shown in [table 4.5](#). Types that require an import must always list that import, even if they're in the same package as your `.aidl` file.

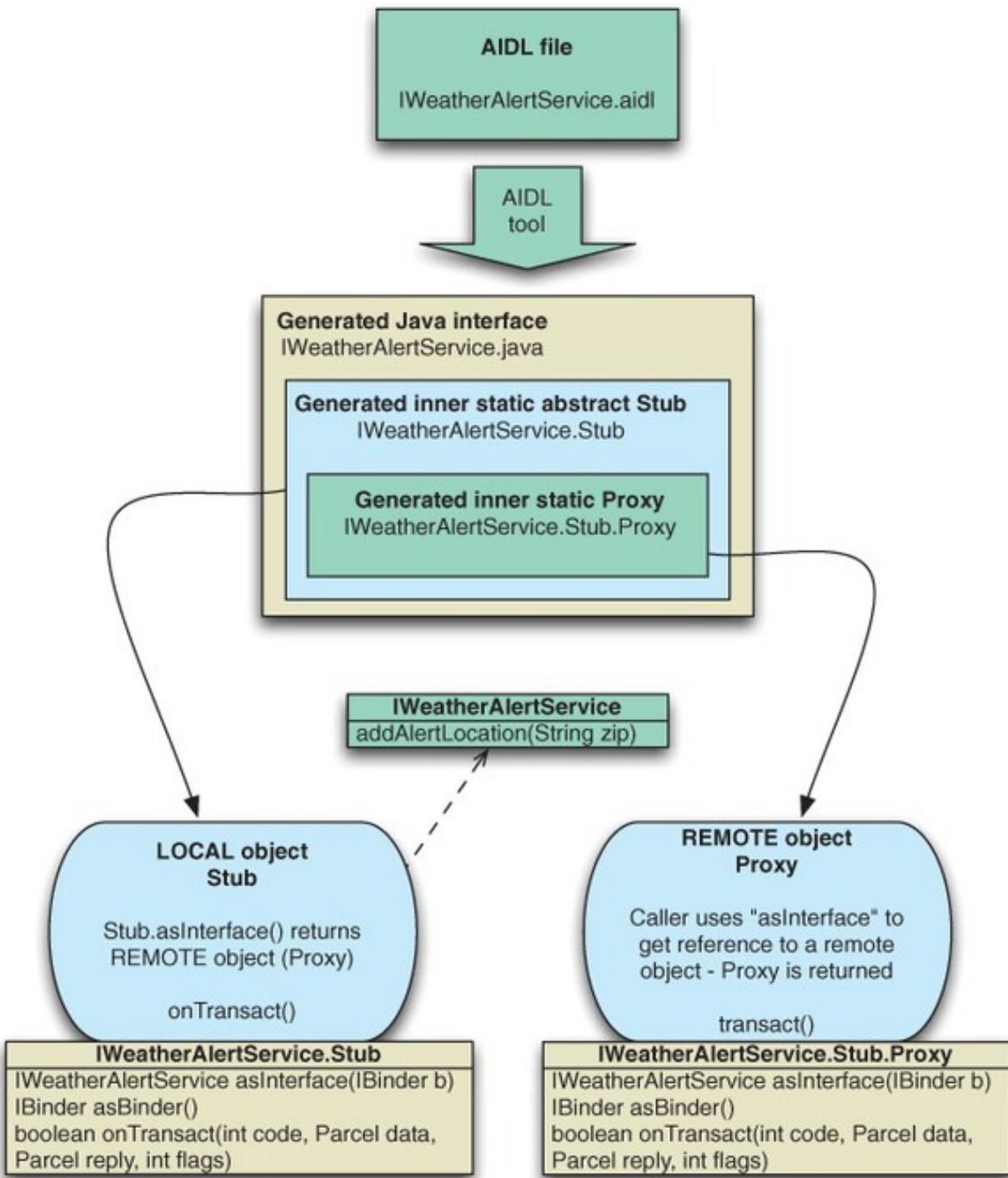
Table 4.5. Android IDL allowed types

Type	Description	Import required
Java primitives	boolean, byte, short, int, float, double, long, char.	No
String	java.lang.String.	No
CharSequence	java.lang.CharSequence.	No
List	Can be generic; all types used in collection must be allowed by IDL. Ultimately provided as an ArrayList.	No
Map	Can be generic, all types used in collection must be one allowed by IDL. Ultimately provided as a HashMap.	No
Other AIDL interfaces	Any other AIDL-generated interface type.	Yes
Parcelable objects	Objects that implement the Android Parcelable interface, described in section 4.5.2 .	Yes

After you've defined your interface methods with return types and parameters, you then invoke the `aidl` tool included in your Android SDK installation to generate a Java interface that represents your AIDL specification. If you use the Eclipse plug-in, it'll automatically invoke the `aidl` tool for you, placing the generated files in the appropriate package in your project's gen folder.

The interface generated through AIDL includes an inner static abstract class named `Stub`, which extends `Binder` and implements the outer class interface. This `Stub` class represents the *local* side of your remotable interface. `Stub` also includes an `asInterface(IBinder binder)` method that returns a *remote* version of your interface type. Callers can use this method to get a handle to the remote object and use it to invoke remote methods. The AIDL process generates a `Proxy` class (another inner class, this time inside `Stub`) that connects all these components and returns to callers from the `asInterface()` method. [Figure 4.6](#) depicts this IPC local/remote relationship.

Figure 4.6. Diagram of the Android AIDL process



After all the required files are generated, create a concrete class that extends from `Stub` and implements your interface. Then, expose this interface to callers through a `Service`. We'll be doing that soon, but first, let's take a quick look under the hood and see how these generated files work.

4.5.2. Binder and Parcelable

The `IBinder` interface is the base of the remoting protocol in Android. As we discussed in the previous section, you don't implement this interface directly; rather, you typically use AIDL to generate an interface which contains a `Stub Binder` implementation.

The `IBinder.transact()` method and corresponding `Binder.onTransact()` method form the backbone of the remoting process. Each method you define using AIDL is handled synchronously through the transaction process, enabling the same semantics as if the method were local.

All the objects you pass in and out through the interface methods that you define using AIDL use this transact process. These objects must be `Parcelable` in order for you to place them inside a `Parcel` and move them across the local/remote process barrier in the `Bindertransaction` methods.

The only time you need to worry about something being `Parcelable` is when you want to send a custom object through Android IPC. If you use only the default allowable types in your interface definition files—primitives, `String`, `CharSequence`, `List`, and `Map`—AIDL automatically handles everything.

The Android documentation describes what methods you need to implement to create a `Parcelable` class. Remember to create an `.aidl` file for each `Parcelable` interface. These `.aidl` files are different from those you use to define `Binder` classes themselves; these shouldn't be generated from the `aidl` tool.

Caution

When you're considering creating your own `Parcelable` types, make sure you actually need them. Passing complex objects across the IPC boundary in an embedded environment is expensive and tedious; you should avoid doing it, if possible.

4.5.3. Exposing a remote interface

Now that you've defined the features you want to expose from the weather app, you need to implement that functionality and make it available to external callers. Android calls this *publishing* the interface.

To publish a remote interface, you create a class that extends `Service` and returns an `IBinder` through the `onBind(Intent intent)` method. Clients will use that `IBinder` to access a particular remote object. As we discussed in [section 4.5.2](#), you can use the AIDL-generated `Stub` class, which itself extends `Binder`, to extend from and return an implementation of a remotable interface. This process is shown in the following listing, where we implement and publish the `IWeatherReporter` service we created in the previous section.

Listing 4.8. Implementing a weather service that publishes a remotable object

```
public class WeatherReportService extends WeatherAlertService {
    private final class WeatherReporter
        extends IWeatherReporter.Stub {
            public String getWeatherFor(String zip) throws RemoteException {
                WeatherRecord record = loadRecord(zip);
                return record.getCondition().getDisplay();
            }
            public void addLocation(String zip, String city, String region)
                throws RemoteException {
                DBHelper db = new DBHelper(WeatherReportService.this);
                Location location = new Location();
                location.alertenabled = 0;
                location.lastalert = 0;
                location.zip = zip;
                location.city = city;
                location.region = region;
                db.insert(location);
            }
        };
        public IBinder onBind(Intent intent) {
            return new WeatherReporter();
        }
}
```

Implement remote interface 1

Return IBinder representing remotable object 2

Our concrete instance of the generated AIDL Java interface must return an `IBinder` to any caller that binds to this `Service`. We create an implementation by extending

the `Stub` class that the `aidl` tool generated 1. Recall that this `Stub` class implements the AIDL interface and extends `Binder`. After we've defined our `IBinder`, we can create and return it from the `onBind()` method 2.

Within the stub itself, we write whatever code is necessary to provide the features advertised by our interface. You can access any other classes within your application. In this example, our `Service` has extended `WeatherAlertService` so we can more easily access the weather functions we've already written, such as the `loadRecord()` method.

You'll need to define this new `WeatherReportService` in your application's manifest, in the same way you define any other `Service`. If you want to bind to the `Service` only from within your own application, no other steps are necessary. But if you want to allow binding from another application, you must provide some extra information within `AndroidManifest.xml`, as shown in the following listing.

Listing 4.9. Exporting a `Service` for other applications to access

```
<service android:name=".service.WeatherReportService"
    android:exported="true">
    <intent-filter>
        <action android:name=
            "com.msi.manning.weather.IWeatherReportService"/>
    </intent-filter>
</service>
```

To allow external applications to find our `Service`, we instruct Android to export this `Service` declaration. Exporting the declaration allows other applications to launch the `Service`, a prerequisite for binding with it. The actual launch will happen through an `<intent-filter>` that we define. In this example, the caller must know the full name of the action, but any `<intent-filter>` we discussed earlier in the chapter can be substituted, such as filtering by scheme or by type.

Now that you've seen how a caller can get a reference to a remotable object, we'll finish that connection by binding to a `Service` from an `Activity`.

4.5.4. Binding to a Service

Let's switch hats and pretend that, instead of writing a weather service, we're another company that wants to integrate weather functions into our own app. Our app will let the user enter a ZIP code and either look up the current weather for that location or save it to the `WeatherReport` application's list of saved locations. We've received the `.aidl` file and learned the name of the `Service`. We generate our own interface from that `.aidl` file, but before we can call the remote methods, we'll need to first bind with the `Service`.

When an `Activity` class binds to a `Service` using the `Context.bindService (Intent i, ServiceConnection connection, int flags)` method, the `ServiceConnection` object that you pass in will send several callbacks from the `Service` back to the `Activity`. The callback `onServiceConnected(ComponentName className, IBinder binder)` lets you know when the binding process completes. The platform automatically injects the `IBinder` returned from the `Service`'s `onBind()` method into this callback, where you

can save it for future calls. The following listing shows an `Activity` that binds to our weather-reporting service and invokes remote methods on it. You can see the complete source code for this project in the chapter downloads.

Listing 4.10. Binding to a Service within an Activity

```
package com.msi.manning.weatherchecker;
. . . Imports omitted for brevity
public class WeatherChecker extends Activity {
    private IWeatherReporter reporter;
    private boolean bound;
    private EditText zipEntry;
    private Handler uiHandler;
    private ServiceConnection connection =
        new ServiceConnection() {
            public void onServiceConnected
                (ComponentName name, IBinder service) {
                    reporter = IWeatherReporter.Stub.
                        asInterface(service);
                    Toast.makeText(WeatherChecker.this, "Connected to Service",
                        Toast.LENGTH_SHORT).show();
                    bound = true;
                }
            public void onServiceDisconnected
                (ComponentName name) {
                    reporter = null;
                    Toast.makeText(WeatherChecker.this, "Disconnected from Service",
                        Toast.LENGTH_SHORT).show();
                    bound = false;
                }
        };
    . . . onCreate method omitted for brevity
    public void checkWeather(View caller) {
        final String zipCode = zipEntry.getText().toString();
        if (zipCode != null & zipCode.length() == 5) {
            new Thread() {
                public void run() {
                    try {
                        final String currentWeather =
                            reporter.getWeatherFor(zipCode);
                        uiHandler.post(new Runnable() {
                            public void run() {
                                Toast.makeText(WeatherChecker.this, currentWeather,
                                    Toast.LENGTH_LONG).show();
                            }
                        });
                    } catch (DeadObjectException e) {
                        e.printStackTrace();
                    } catch (RemoteException e) {
                        e.printStackTrace();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }.start();
        }
    }
}
```

1 Use generated interface

2 Define ServiceConnection behavior

3 Retrieve remotely callable interface

4 Don't block UI thread

4 Invoke remote method

4 Show feedback on UI thread

```

public void saveLocation(View caller) {
    final String zipCode = zipEntry.getText().toString();
    if (zipCode != null & zipCode.length() == 5) {
        new Thread() {
            public void run() {
                try {
                    reporter.addLocation(zipCode, "", "");
                    uiHandler.post(new Runnable() {
                        public void run() {
                            Toast.makeText(
                                WeatherChecker.this, R.string.saved,
                                Toast.LENGTH_LONG).show();
                        }
                    });
                } catch (DeadObjectException e) {
                    e.printStackTrace();
                } catch (RemoteException e) {
                    e.printStackTrace();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }.start();
    }
}

public void onStart() {
    super.onStart();
    if (!bound) {
        bindService(new Intent
            (IWeatherReporter.class.getName()),
            connection,
            Context.BIND_AUTO_CREATE);
    }
}

public void onPause() {
    super.onPause();
    if (bound){
        bound = false;
        unbindService(connection);
    }
}
}

```

Don't block
UI thread

Show feedback
on UI thread

5 Start binding
to Service

In order to use the remotable `IWeatherReporter` we defined in AIDL, we declare a variable with this type ①. We also define a `boolean` to keep track of the current state of the binding. Keeping track of the current state will prevent us from rebinding to the Service if our application is suspended and resumed.

We use the `ServiceConnection` object **2** to bind and unbind using `Context` methods. After a `Service` is bound, the platform notifies us through the `onServiceConnected` callback. This **3** callback returns the remote `IBinder` reference,

which we assign to the remotable type **3** so we can invoke it later. Next, a similar `onServiceDisconnected` callback will fire when a `Service` is unbound.

After we've established a connection, we can use the AIDL-generated interface to perform the operations it defines **4**. When we call `getWeatherFor` (or later, `addLocation`), Android will dispatch our invocation across the process boundary, where the `Service` we created in [listing 4.8](#) will execute the methods. The return values will be sent back across the process boundary and arrive as shown at **4**. This sequence can take a long time, so you should avoid calling remote methods from the UI thread.

In `onStart()`, we establish the binding using `bindService()`; later, in `onPause()`, we use `unbindService()`. The system can choose to clean up a `Service` that's been bound but not started. You should always unbind an unused `Service` so the device can reclaim its resources and perform better. Let's look more closely at the difference between starting and binding a `Service`.

4.5.5. Starting vs. binding

Services serve two purposes in Android, and you can use them in two different ways:

- **Starting**— `Context.startService(Intent service, Bundle b)`
- **Binding**— `Context.bindService(Intent service, ServiceConnection c, int flag)`

Starting a `Service` tells the platform to launch it in the background and keep it running, without any particular connection to any other `Activity` or application. You used the `WeatherAlertService` in this manner to run in the background and issue severe weather alerts.

Binding to a `Service`, as you did with `WeatherReporterService`, gave you a handle to a remote object, which let you call the `Service`'s exported methods from an `Activity`. Because every Android application runs in its own process, using a bound `Service` lets you pass data between processes.

The actual process of marshaling and unmarshaling remotable objects across process boundaries is complicated. Fortunately, you don't have to deal with all the internals,

because Android handles the complexity through AIDL. Instead, you can stick to a simple recipe that will enable you to create and use remotable objects:

1. Define your interface using AIDL, in the form of a .aidl file; see [listing 4.7](#).
2. Generate a Java interface for the .aidl file. This happens automatically in Eclipse.
3. Extend from the generated `Stub` class and implement your interface methods; see [listing 4.8](#).
4. Expose your interface to clients through a `Service` and the `Service onBind(Intent i)` method; see [listing 4.8](#).
5. If you want to make your `Service` available to other applications, export it in your manifest; see [listing 4.9](#).
6. Client applications will bind to your `Service` with a `ServiceConnection` to get a handle to the remotable object; see [listing 4.10](#).

As we discussed earlier in the chapter, `Services` running in the background can have a detrimental impact on overall device performance. To mitigate these problems, Android enforces a special lifecycle for `Services`, which we're going to discuss now.

4.5.6. Service lifecycle

You want the weather-alerting `Service` to constantly lurk in the background, letting you know of potential dangers. On the other hand, you want the weather-reporting `Service` to run only while another application actually needs it. `Services` follow their own well-defined process phases, similar to those followed by an `Activity` or an `Application`. A `Service` will follow a different lifecycle, depending on whether you start it, bind it, or both.

Service-Started Lifecycle

If you start a `Service` by calling `Context.startService(Intent service, Bundle b)`, as shown in [listing 4.5](#), it runs in the background whether or not anything binds to it. If the `Service` hasn't been created, the `Service onCreate()` method is called.

The `onStart(int id, Bundle args)` method is called each time someone tries to start the `Service`, regardless of whether it's already running. Additional instances of the `Service` won't be created.

The `Service` will continue to run in the background until someone explicitly stops it with the `Context.stopService()` method or when the `Service` calls its own `stopSelf()` method. You should also keep in mind that the platform might kill `Services` if resources are running low, so your application needs to be able to react accordingly. You can choose to restart the `Service` automatically, fall back to a more limited feature set without it, or take some other appropriate action.

Service-Bound Lifecycle

If an `Activity` binds a `Service` by calling `Context.bindService(Intent service, ServiceConnection connection, int flags)`, as shown in [listing 4.10](#), it'll run as long as the connection is open. An `Activity` establishes the connection using the `Context` and is also responsible for closing it.

When a `Service` is only bound in this manner and not also started, its `onCreate()` method is invoked, but `onStart(int id, Bundle args)` is *not* used. In these cases, the platform can stop and clean up the `Service` after it's unbound.

Service-Started and Service-Bound Lifecycle

If a `Service` is both started and bound, it'll keep running in the background, much like in the started lifecycle. In this case, both `onStart(int id, Bundle args)` and `onCreate()` are called.

Cleaning Up When a Service Stops

When a `Service` stops, its `onDestroy()` method is invoked. Inside `onDestroy()`, every `Service` should perform final cleanup, stopping any spawned threads, terminating network connections, stopping `Services` it had started, and so on.

And that's it! From birth to death, from invocation to dismissal, you've learned how to wrangle Android `services`. They might seem complex, but they offer extremely powerful capabilities that can go far beyond what a single foregrounded application can offer.

4.6. SUMMARY

In this chapter, we covered a broad swath of Android territory. We first focused on the `Intent` component, seeing how it works, how it resolves using `IntentFilter` objects, and how to take advantage of built-in platform-provided `Intent` handlers. We also looked at the differences between explicit `Intent` invocation and implicit `Intent` invocation, and the reasons you might choose one type over another. Along the way, you completed the `RestaurantFinder` sample application, and with just a bit more code, you drastically expanded the usefulness of that app by tapping into preloaded Android applications.

After we covered the `Intent` class, we moved on to a new sample application, `WeatherReporter`. You saw how a `BroadcastReceiver` could respond to notifications sent by the platform or other applications. You used the receiver to listen for a boot event and start the `Service`. The `Service` sends notification alerts from the background when it learns of severe weather events. You also saw another flavor of `Service`, one that provides communication between different processes. Our other weather service offered an API that third-party developers could use to take advantage of the low-level network

and storage capabilities of the weather application. We covered the difference between starting and binding `Services`, and you saw the moving parts behind the Android IPC system, which uses the AIDL to standardize communication between applications.

By seeing all these components interact in several complete examples, you now understand the fundamentals behind Android `Intents` and `Services`. In the next chapter, you'll see how to make `Services` and other applications more useful by using persistent storage. We'll look at the various options Android provides for retrieving and storing data, including preferences, the file system, databases, and how to create a custom `ContentProvider`.

Chapter 5. Storing and retrieving data

This chapter covers

- Storing and retrieving data with `SharedPreferences`
- Using the filesystem
- Working with a SQLite database
- Accessing and building a `ContentProvider`

Android provides several ways to store and share data, including access to the filesystem, a local relational database through SQLite, and a preferences system that allows you to store simple key/value pairs within applications. In this chapter, we'll start with preferences and you'll create a small sample application to exercise those concepts. From there, you'll create another sample application to examine using the filesystem to store data, both internal to the application and external using the platform's Secure Digital (SD) card support. You'll also see how to create and access a database.

Beyond the basics, Android also allows applications to share data through a clever URI-based approach called a `ContentProvider`. This technique combines several other Android concepts, such as the URI-based style of intents and the `Cursor` result set seen in SQLite, to make data accessible across different applications. To demonstrate how this works, you'll create another small sample application that uses built-in providers, then we'll walk through the steps required to create your own `ContentProvider`.

We'll begin with preferences, the simplest form of data storage and retrieval Android provides.

5.1. USING PREFERENCES

If you want to share simple application data from one `Activity` to another, use a `SharedPreferences` object. You can save and retrieve data, and also choose whether to make preferences private to your application or accessible to other applications on the same device.

5.1.1. Working with `SharedPreferences`

You access a `SharedPreferences` object through your current Context, such as the Activity or Service. Context defines the method `getSharedPreferences(String name, int accessMode)` that allows you to get a preferences handle. The name you specify will be the name for the file that backs these preferences. If no such file exists when you try to get preferences, one is automatically created. The access mode refers to what permissions you want to allow.

The following listing demonstrates allowing the user to input and store data through `SharedPreferences` objects with different access modes.

Listing 5.1. Storing `SharedPreferences` using different modes

```
package com.msi.manning.chapter5.prefs;
// imports omitted for brevity
public class SharedPrefTestInput extends Activity {
    public static final String PREFS_PRIVATE = "PREFS_PRIVATE";
    public static final String PREFS_WORLD_READ = "PREFS_WORLD_READABLE";
    public static final String PREFS_WORLD_WRITE = "PREFS_WORLD_WRITABLE";
    public static final String PREFS_WORLD_READ_WRITE =
        "PREFS_WORLD_READABLE_WRITABLE";
    public static final String KEY_PRIVATE = "KEY_PRIVATE";
    public static final String KEY_WORLD_READ = "KEY_WORLD_READ";
    public static final String KEY_WORLD_WRITE = "KEY_WORLD_WRITE";
    public static final String KEY_WORLD_READ_WRITE =
        "KEY_WORLD_READ_WRITE";
    . . . view element variable declarations omitted for brevity
    private SharedPreferences prefsPrivate;
    private SharedPreferences prefsWorldRead;
    private SharedPreferences prefsWorldWrite;
    private SharedPreferences prefsWorldReadWrite;
    @Override
    public void onCreate(Bundle icicle) {
        . . . view inflation omitted for brevity
        button.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                boolean valid = validate();
                if (valid) {
                    prefsPrivate =
                        getSharedPreferences(
                            SharedPrefTestInput.PREFS_PRIVATE,
                            Context.MODE_PRIVATE);
                    prefsWorldRead =
                        getSharedPreferences(
                            SharedPrefTestInput.PREFS_WORLD_READ,
                            Context.MODE_WORLD_READABLE);
                    prefsWorldWrite =
```

The diagram consists of three numbered callouts pointing to specific parts of the code. Callout 1 points to the declaration of four static final String variables: PREFS_PRIVATE, PREFS_WORLD_READ, PREFS_WORLD_WRITE, and PREFS_WORLD_READ_WRITE. Callout 2 points to the line of code `prefsPrivate = getSharedPreferences(SharedPrefTestInput.PREFS_PRIVATE, Context.MODE_PRIVATE);`. Callout 3 points to the line of code `prefsWorldRead = getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_READ, Context.MODE_WORLD_READABLE);`.

```

        getSharedPreferences(
            SharedPrefTestInput.PREFS_WORLD_WRITE,
            Context.MODE_WORLD_WRITEABLE);
        prefsWorldReadWrite =
            getSharedPreferences(
                SharedPrefTestInput.PREFS_WORLD_READ_WRITE,
                Context.MODE_WORLD_READABLE
                + Context.MODE_WORLD_WRITEABLE);
        Editor prefsPrivateEditor =
            prefsPrivate.edit();
        Editor prefsWorldReadEditor =
            prefsWorldRead.edit();
        Editor prefsWorldWriteEditor =
            prefsWorldWrite.edit();
        Editor prefsWorldReadWriteEditor =
            prefsWorldReadWrite.edit();
        prefsPrivateEditor.putString(
            SharedPrefTestInput.KEY_PRIVATE,
            inputPrivate.getText().toString());
        prefsWorldReadEditor.putString(
            SharedPrefTestInput.KEY_WORLD_READ,
            inputWorldRead.getText().toString());
        prefsWorldWriteEditor.putString(
            SharedPrefTestInput.KEY_WORLD_WRITE,
            inputWorldWrite.getText().toString());
        prefsWorldReadWriteEditor.putString(
            SharedPrefTestInput.KEY_WORLD_READ_WRITE,
            inputWorldReadWrite.getText().toString());
        prefsPrivateEditor.commit();
        prefsWorldReadEditor.commit();
        prefsWorldWriteEditor.commit();
        prefsWorldReadWriteEditor.commit();
        Intent intent =
            new Intent(SharedPrefTestInput.this,
            SharedPrefTestOutput.class);
        startActivity(intent);
    }
}
);
}
. . . validate omitted for brevity
)

```

The diagram illustrates the sequence of operations:

- Get SharedPreferences editor**: Step 4, indicated by a blue circle with the number 4 and an arrow pointing to the line `Editor prefsPrivateEditor = prefsPrivate.edit();`.
- Store values with editor**: Step 5, indicated by a blue circle with the number 5 and an arrow pointing to the line `prefsPrivateEditor.putString(SharedPrefTestInput.KEY_PRIVATE, inputPrivate.getText().toString());`.
- Persist changes**: Step 6, indicated by a blue circle with the number 6 and an arrow pointing to the line `prefsPrivateEditor.commit();`.

After you have a `SharedPreferences` variable ①, you can acquire a reference through the `Context` ②. Note that for each `SharedPreferences` object we get, we use a different constant value for the access mode, and in some cases we also add modes ③. We repeat this coding for each mode we retrieve. Modes specify whether the preferences should be private, world-readable, or world-writable.

To modify preferences, you must get an `Editor` handle **4**. With the `Editor`, you can set `String`, `boolean`, `float`, `int`, and `long` types as key/value pairs **5**. This limited set of types can be restrictive, but often preferences are adequate, and they're simple to use.

After storing with an `Editor`, which creates an in-memory `Map`, you have to call `commit()` to persist it to the preferences backing file **6**. After data is committed, you can easily get it from a `SharedPreferences` object. The following listing gets and displays the data that was stored in listing 5.1.

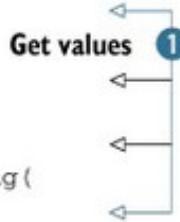
Listing 5.2. Getting `SharedPreferences` data stored in the same application

```
package com.msi.manning.chapter5.prefs;
// imports omitted for brevity
public class SharedPrefTestOutput extends Activity {
    . . . view element variable declarations omitted for brevity
    private SharedPreferences prefsPrivate;
    private SharedPreferences prefsWorldRead;
    private SharedPreferences prefsWorldWrite;
    private SharedPreferences prefsWorldReadWrite;
    . . . onCreate omitted for brevity
    @Override
    public void onStart() {
        super.onStart();
        prefsPrivate =
            getSharedPreferences(SharedPrefTestInput.PREFS_PRIVATE,
                Context.MODE_PRIVATE);
        prefsWorldRead =
```

```

        getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_READ,
            Context.MODE_WORLD_READABLE);
        prefsWorldWrite =
            getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_WRITE,
            Context.MODE_WORLD_WRITEABLE);
        prefsWorldReadWrite =
            getSharedPreferences(
                SharedPrefTestInput.PREFS_WORLD_READ_WRITE,
                Context.MODE_WORLD_READABLE
                + Context.MODE_WORLD_WRITEABLE);
        outputPrivate.setText(prefsPrivate.getString(
            SharedPrefTestInput.KEY_PRIVATE, "NA"));
        outputWorldRead.setText(prefsWorldRead.getString(
            SharedPrefTestInput.KEY_WORLD_READ, "NA"));
        outputWorldWrite.setText(prefsWorldWrite.getString(
            SharedPrefTestInput.KEY_WORLD_WRITE, "NA"));
        outputWorldReadWrite.setText(prefsWorldReadWrite.getString(
            SharedPrefTestInput.KEY_WORLD_READ_WRITE,
            "NA"));
    }
}

```



Get values 1

To retrieve previously stored values, we again declare variables and assign references. When these are in place, we can get values using methods such as `getString(String key, String default)` 1. The `default` value is returned if no data was previously stored with that key.

Setting and getting preferences is straightforward. Access modes, which we'll focus on next, add a little more complexity.

5.1.2. Preference access permissions

You can open and create `SharedPreferences` with any combination of several `Context` mode constants. Because these values are `int` types, you can add them, as in listings 5.1 and 5.2, to combine permissions. The following mode constants are supported:

- `Context.MODE_PRIVATE` (value 0)
- `Context.MODE_WORLD_READABLE` (value 1)
- `Context.MODE_WORLD_WRITEABLE` (value 2)

These modes allow you to tune who can access this preference. If you take a look at the filesystem on the emulator after you've created `SharedPreferences` objects (which themselves create XML files to persist the data), you can see how setting permissions works using a Linux-based filesystem.

Figure 5.1 shows the Android Eclipse plug-in File Explorer view. Within the explorer, you can see the Linux-level permissions for the `SharedPreferences` XML files that we created from the `SharedPreferences` in listing 5.1.

Figure 5.1. The Android File Explorer view showing preferences file permissions

	com.msi.manning.chapter5.prefs	2008-03-12	13:40	drwxrwx--x
▼	shared_prefs	2008-03-12	13:41	drwxrwx--x
	PREFS_PRIVATE.xml	114	2008-03-12	13:41 -rw-rw----
	PREFS_WORLD_READABLE.xml	117	2008-03-12	13:41 -rw-rw-r--
	PREFS_WORLD_READABLE_WRITABLE.xml	126	2008-03-12	13:41 -rw-rw-rw-
	PREFS_WORLD_WRITABLE.xml	119	2008-03-12	13:41 -rw-rw--w-
►	com.other.manning.chapter5.prefs		2008-03-12	13:42 drwxrwx--x
►	download		2008-03-12	13:37 drwxrwxrwx

Each Linux file or directory has a type and three sets of permissions, represented by a `drwxrwxrwx` notation. The first character indicates the type (`d` means directory, `-` means regular file type, and other types such as symbolic links have unique types as well). After the type, the three sets of `rwx` represent the combination of read, write, and execute permissions for user, group, and *world*, in that order. Looking at this notation, you can tell which files are accessible by the user they're owned by, by the group they belong to, or by everyone else on the device. Note that the user and group always have full permission to read and write, whereas the final set of permissions fluctuates based on the preference's mode.

Android puts `SharedPreferences` XML files in the `/data/data/PACKAGE_NAME/shared_prefs` path on the filesystem. An application or package usually has its own user ID. When an application creates files, including `SharedPreferences`, they're owned by that application's user ID. To allow other applications to access these files, you have to set the *world* permissions, as shown in figure 5.1.

Directories with the *world x* permission

In Android, each package directory is created with the *world x* permission. This permission means anyone can search and list the files in the directory, which means that Android packages have directory-level access to one another's files. From there, file-level access determines file permissions.

If you want to access another application's files, you must know the starting path. The path comes from the `Context`. To get files from another application, you have to know and use that application's `Context`. Android doesn't officially condone sharing preferences across multiple applications; in practice, apps should use a content provider to share this kind of data. Even so, looking at `SharedPreferences` does show the underlying data storage models in Android. The following listing shows how to get the `SharedPreferences` we set in listing 5.1 again, this time from a different application (different .apk and different package).

Listing 5.3. Getting `SharedPreferences` data stored in a different application

```
package com.other.manning.chapter5.prefs;
. . . imports omitted for brevity
public class SharedPrefTestOtherOutput extends Activity {
    . . . constants and variable declarations omitted for brevity
    . . . onCreate omitted for brevity
    @Override
    public void onStart() {
        super.onStart();
        Context otherAppsContext = null;
        try {
            otherAppsContext =
                createPackageContext("com.msi.manning.chapter5.prefs",
                    Context.MODE_WORLD_WRITEABLE);
        } catch (NameNotFoundException e) {
            // log and/or handle
        }
        prefsPrivate =
            otherAppsContext.getSharedPreferences(
                SharedPrefTestOtherOutput.PREFS_PRIVATE, 0);
        prefsWorldRead =
            otherAppsContext.getSharedPreferences(
                SharedPrefTestOtherOutput.PREFS_WORLD_READ, 0);
        prefsWorldWrite =
```

- 1 Use different package
- 2 Get another application's context
- 3 Use otherAppsContext

```

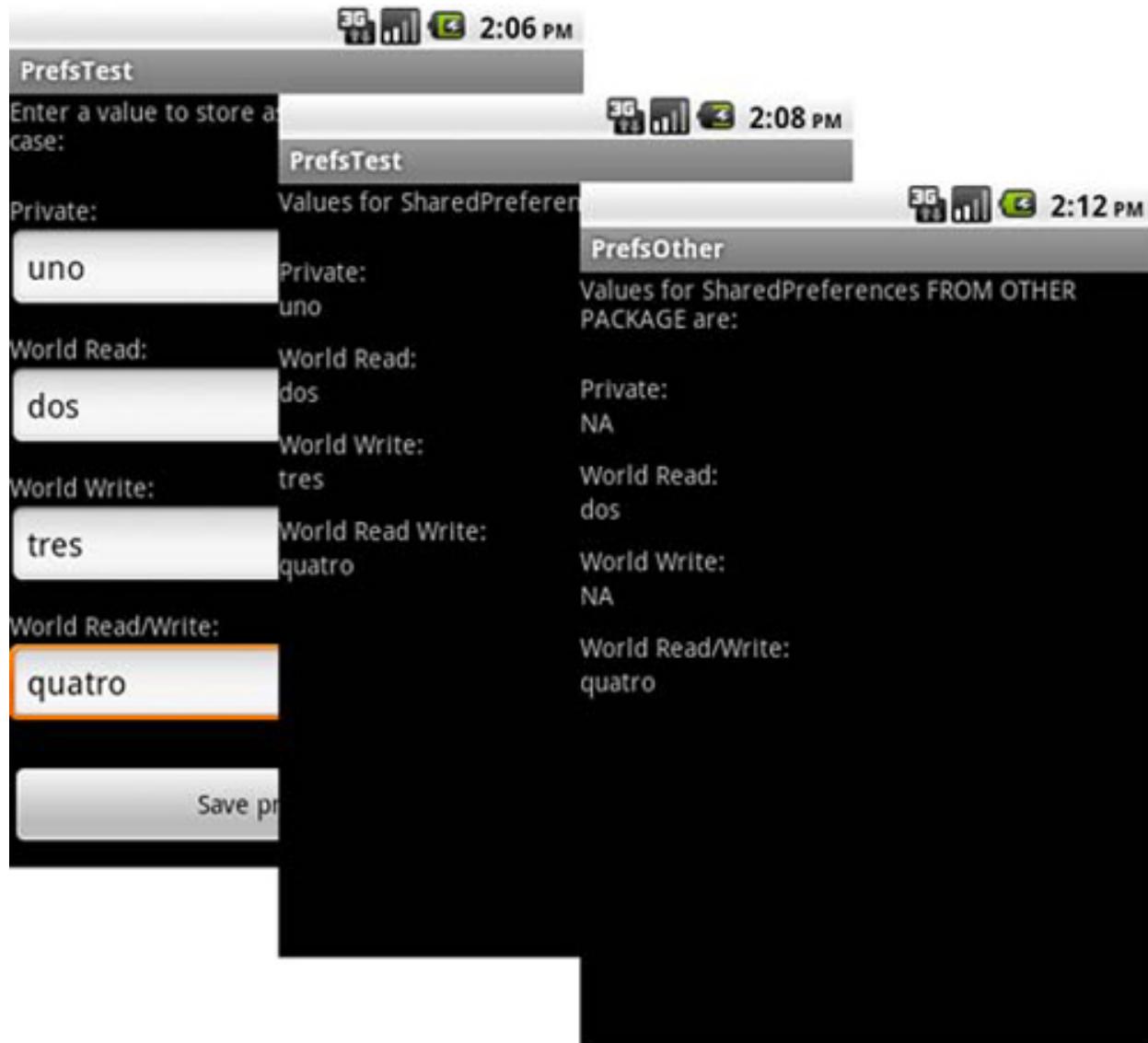
        otherAppsContext.getSharedPreferences(
            SharedPrefTestOtherOutput.PREFS_WORLD_WRITE, 0);
prefsWorldReadWrite =
        otherAppsContext.getSharedPreferences(
            SharedPrefTestOtherOutput.PREFS_WORLD_READ_WRITE, 0);
outputPrivate.setText(
    prefsPrivate.getString(
        SharedPrefTestOtherOutput.KEY_PRIVATE, "NA"));
outputWorldRead.setText(
    prefsWorldRead.getString(
        SharedPrefTestOtherOutput.KEY_WORLD_READ, "NA"));
outputWorldWrite.setText(
    prefsWorldWrite.getString(
        SharedPrefTestOtherOutput.KEY_WORLD_WRITE, "NA"));
outputWorldReadWrite.setText(
    prefsWorldReadWrite.getString(
        SharedPrefTestOtherOutput.KEY_WORLD_READ_WRITE, "NA"));
}
}

```

To get one application's `SharedPreferences` from another application's package ①, we use the `createPackageContext(String contextName, int mode)` method ②. When we have the other application's `Context`, we can use the same names for the `SharedPreferences` objects that the other application created to access those preferences ③.

With these examples, we now have one application that sets and gets `SharedPreferences`, and a second application with a different .apk file that gets the preferences set by the first. The composite screen shot in figure 5.2 shows what the apps look like. NA indicates a preference we couldn't access from the second application, either as the result of permissions that were set or because no permissions had been created.

Figure 5.2. Two separate applications getting and setting `SharedPreferences`



Though `SharedPreferences` are ultimately backed by XML files on the Android filesystem, you can also directly create, read, and manipulate files, as we'll discuss in the next section.

5.2. USING THE FILESYSTEM

Android's filesystem is based on Linux and supports mode-based permissions. You can access this filesystem in several ways. You can create and read files from within applications, you can access raw resource files, and you can work with specially compiled custom XML files. In this section, we'll explore each approach.

5.2.1. Creating files

Android's stream-based system of manipulating files will feel familiar to anyone who's written I/O code in Java SE or Java ME. You can easily create files in Android and store them in your application's data path. The following listing demonstrates how to open a `FileOutputStream` and use it to create a file.

Listing 5.4. Creating a file in Android from an Activity

```
public class CreateFile extends Activity {  
    private EditText createInput;  
    private Button createButton;  
    @Override  
    public void onCreate(Bundle icicle) {  
        super.onCreate(icicle);  
        setContentView(R.layout.create_file);  
        createInput =  
            (EditText) findViewById(R.id.create_input);  
        createButton =  
            (Button) findViewById(R.id.create_button);  
        createButton.setOnClickListener(new OnClickListener() {  
            public void onClick(final View v) {  
                FileOutputStream fos = null;  
                try {  
                    fos = openFileOutput("filename.txt",  
                        Context.MODE_PRIVATE);  
                    fos.write(createInput.getText().  
                        toString().getBytes());  
                } catch (FileNotFoundException e) {  
                    Log.e("CreateFile", e.getLocalizedMessage());  
                } catch (IOException e) {  
                    Log.e("CreateFile", e.getLocalizedMessage());  
                } finally {  
                    if (fos != null) {  
                        try {  
                            fos.flush();  
                            fos.close();  
                        } catch (IOException e) {  
                            // swallow  
                        }  
                    }  
                }  
                startActivity(  
                    new Intent(CreateFile.this, ReadFile.class));  
            }  
        });  
    }  
}
```

The diagram consists of three numbered callouts pointing to specific parts of the code. Callout 1 points to the line `fos = openFileOutput("filename.txt", Context.MODE_PRIVATE);`. Callout 2 points to the line `fos.write(createInput.getText().toString().getBytes());`. Callout 3 points to the code block within the finally block that flushes and closes the stream.

1 Use
openFileOutput

2 Write data
to stream

3 Flush and
close stream

Android provides a convenience method on Context¹ to get a FileOutputStream— namely `openFileOutput(String name, int mode)`². Using this method, you can create a stream to a file. That file will ultimately be stored at the `data/data/[PACKAGE_NAME]/files/file.name` path on the platform. After you have the stream, you can write to it as you would with typical Java³. After you're finished with a stream, you should flush and close it to clean up .

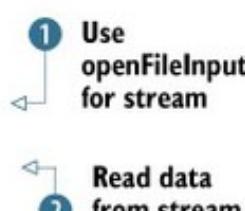
Reading from a file within an application context (within the package path of the application) is also simple; in the next section we'll show you how.

5.2.2. Accessing files

Similar to `openFileOutput()`, the Context also has a convenience `openFileInput()` method. You can use this method to access a file on the filesystem and read it in, as shown in the following listing.

Listing 5.5. Accessing an existing file in Android from an Activity

```
public class ReadFile extends Activity {
    private TextView readOutput;
    private Button gotoReadResource;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_file);
        readOutput =
            (TextView) findViewById(R.id.read_output);
        FileInputStream fis = null;
        try {
            fis = openFileInput("filename.txt");
            byte[] reader = new byte[fis.available()];
            while (fis.read(reader) != -1) {}
            readOutput.setText(new String(reader));
        } catch (IOException e) {
            Log.e("ReadFile", e.getMessage(), e);
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    // swallow
                }
            }
        }
    }
    . . . goto next Activity via startActivity omitted for brevity
}
```



For input, you use `openFileInput(String name, int mode)` to get the stream ①, and then read the file into a byte array as with standard Java ②. Afterward, close the stream properly to avoid hanging on to resources.

With `openFileOutput` and `openFileInput`, you can write to and read from any file within the files directory of the application package you're working in. Also, as we discussed in the previous section, you can access files across different applications if the permissions allow it and if you know the package used to obtain the full path to the file.

Running a bundle of apps with the same user ID

Occasionally, setting the user ID of your application can be extremely useful. For instance, if you have multiple applications that need to share data with one another, but you also don't want that data to be accessible outside that group of applications, you

might want to make the permissions private and share the UID to allow access. You can allow a shared UID by using the `sharedUserId` attribute in your manifest: `android:sharedUserId="YourID"`.



In addition to creating files from within your application, you can push and pull files to the platform using the `adb` tool, described in section 2.2.3. The File Explorer window in Eclipse provides a UI for moving files on and off the device or simulator. You can optionally put such files in the directory for your application; when they're there, you can read these files just like you would any other file. Keep in mind that outside of development-related use, you won't usually push and pull files. Rather, you'll create and read files from within the application or work with files included with an application as raw resources, as you'll see next.

5.2.3. Files as raw resources

If you want to include raw files with your application, you can do so using the `res/raw` resources location. We discussed resources in general in chapter 3. When you place a file in the `res/raw` location, it's not compiled by the platform but is available as a *raw* resource, as shown in the following listing.

Listing 5.6. Accessing a noncompiled raw file from res/raw

```
public class ReadRawResourceFile extends Activity {
    private TextView readOutput;
    private Button gotoReadXMLResource;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_rawresource_file);
        readOutput =
            (TextView) findViewById(R.id.readrawres_output);
        Resources resources = getResources();
        InputStream is = null;
        try {
            is = resources.openRawResource(R.raw.people);
            byte[] reader = new byte[is.available()];
            while (is.read(reader) != -1) {}
            readOutput.setText(new String(reader));
        } catch (IOException e) {
            Log.e("ReadRawResourceFile", e.getMessage(), e);
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    // swallow
                }
            }
        }
    }
    . . . go to next Activity via startActivity omitted for brevity
}
```

1 Hold raw resource with InputStream

2 Use getResources().openRawResource()

Accessing raw resources closely resembles accessing files. You open a handle to an `InputStream` ①. You call `Context.getResources()` to get the `Resources` for your current application's context and then call `openRawResource(int id)` to link to the

particular item you want ②. Android will automatically generate the ID within the `R` class if you place your asset in the `res/raw` directory. You can use any file as a raw resource, including text, images, documents, or videos. The platform doesn't precompile raw resources.

The last type of file resource we need to discuss is the `res/xml` type, which the platform compiles into an efficient binary type accessed in a special manner.

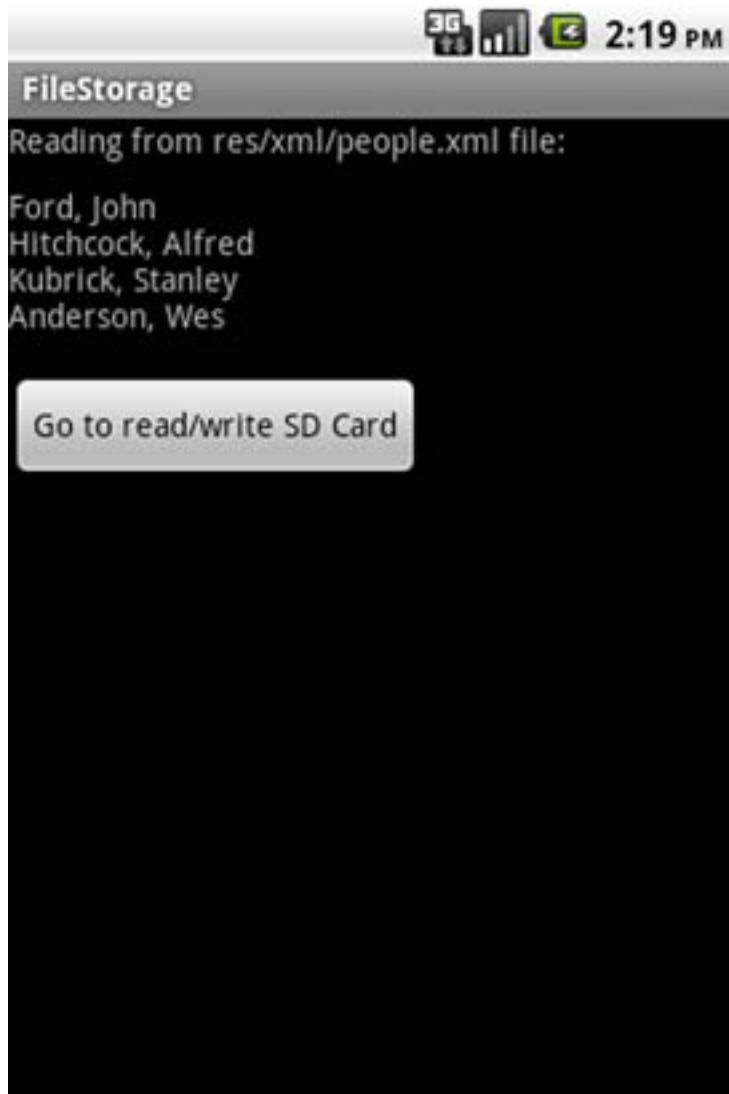
5.2.4. XML file resources

The term *XML resources* sometimes confuses new Android developers. XML resources might mean resources in general that are defined in XML—such as layout files, styles, arrays, and the like—or it can specifically mean res/xml XML files.

In this section, we'll deal with res/xml XML files. These files are different from raw files in that you don't use a stream to access them because they're compiled into an efficient binary form when deployed. They're different from other resources in that they can be of any custom XML structure.

To demonstrate this concept, we're going to use an XML file named people.xml that defines multiple `<person>` elements and uses attributes for `firstname` and `lastname`. We'll grab this resource and display its elements in *last-name, first-name* order, as shown in figure 5.3.

Figure 5.3. The example `ReadXMLResourceFile Activity` that we'll create in [listing 5.8](#), which reads a res/xml resource file



Our data file for this process, which we'll place in `res/xml`, is shown in the following listing.

Listing 5.7. A custom XML file included in `res/xml`

```
<people>
    <person firstname="John" lastname="Ford" />
    <person firstname="Alfred" lastname="Hitchcock" />
    <person firstname="Stanley" lastname="Kubrick" />
    <person firstname="Wes" lastname="Anderson" />
</people>
```

If you're using Eclipse, it'll automatically detect a file in the res/xml path and compile it into a resource asset. You can then access this asset in code by parsing its binary XML, as shown in the following listing.

Listing 5.8. Accessing a compiled XML resource from res/xml

```
public class ReadXMLResourceFile extends Activity {  
    private TextView readOutput;  
    @Override  
    public void onCreate(Bundle icicle) {  
        super.onCreate(icicle);  
        setContentView(R.layout.read_xmlresource_file);  
        readOutput = (TextView)  
            findViewById(R.id.readxmlres_output);  
        XmlPullParser parser =  
            getResources().getXml(R.xml.people);  
        StringBuffer sb = new StringBuffer();  
        try {  
            while (parser.next() != XmlPullParser.END_DOCUMENT) {  
                String name = parser.getName();  
                String first = null;  
                String last = null;  
                if ((name != null) && name.equals("person")) {  
                    int size = parser.getAttributeCount();  
                    for (int i = 0; i < size; i++) {  
                        String attrName =  
                            parser.getAttributeName(i);  
                        String attrValue =  
                            parser.getAttributeValue(i);  
                        if ((attrName != null)  
                            && attrName.equals("firstname")) {  
                            first = attrValue;  
                        } else if ((attrName != null)  
                            && attrName.equals("lastname")) {  
                            last = attrValue;  
                        }  
                    }  
                    if ((first != null) && (last != null)) {  
                        sb.append(last + ", " + first + "\n");  
                    }  
                }  
            }  
            readOutput.setText(sb.toString());  
        } catch (Exception e) {  
            Log.e("ReadXMLResourceFile", e.getMessage(), e);  
        }  
        . . . goto next Activity via startActivity omitted for brevity  
    }  
}
```

The diagram illustrates the flow of the XML parsing process. It starts with step 1, 'Parse XML with XMLPullParser', which is indicated by a blue circle and an arrow pointing to the line 'XmlPullParser parser = getResources().getXml(R.xml.people);'. Step 2, 'Walk XML tree', is indicated by a blue circle and an arrow pointing to the line 'while (parser.next() != XmlPullParser.END_DOCUMENT) {'. Step 3, 'Get attributeCount for element', is indicated by a blue circle and an arrow pointing to the line 'int size = parser.getAttributeCount();'. Step 4, 'Get attribute name and value', is indicated by a blue circle and an arrow pointing to the line 'String attrName = parser.getAttributeName(i);'.

To process a binary XML resource, you use an `XmlPullParser`¹. This class supports SAX-style tree traversal. The parser provides an event type for each element it encounters, such

as `DOCDECL`, `COMMENT`, `START_DOCUMENT`, `START_TAG`, `END_TAG`, `END_DOCUMENT`, and so on. By using the `next()` method, you can retrieve the current event type value and compare it to event constants in the class². Each element encountered has a name, a text value, and an optional set of attributes. You can examine the document contents by getting³ the `attributeCount` for each item and grabbing each name and value⁴. SAX is covered in more detail in chapter 13.

In addition to local file storage on the device filesystem, you have another option that's more appropriate for certain types of content: writing to an external SD card filesystem.

5.2.5. External storage via an SD card

One of the advantages the Android platform provides over some other smartphones is that it offers access to an available SD flash memory card. Not every Android device will necessarily have an SD card, but almost all do, and the platform provides an easy way for you to use it.

SD cards and the emulator

To work with an SD card image in the Android emulator, you'll first need to use the `mksdcard` tool provided to set up your SD image file (you'll find this executable in the tools directory of the SDK). After you've created the file, you'll need to start the emulator with the `-sdcard <path_to_file>` option in order to have the SD image mounted. Alternately, use the Android SDK Manager to create a new virtual device and select the option to create a new SD card.

All applications can read data stored on the SD card. If you want to write data here, you'll need to include the following permission in your `AndroidManifest.xml`:

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

Failing to declare this permission will cause write attempts to the SD card to fail.

Generally, you should use the SD card if you use large files such as images and video, or if you don't need to have permanent secure access to certain files. On the other hand, for permanent application-specialized data, you should use the internal filesystem.

The SD card is removable, and SD card support on most devices (including Android-powered devices) supports the File Allocation Table (FAT) filesystem. The SD card doesn't have the access modes and permissions that come from the Linux filesystem.

Using the SD card is fairly basic. The standard `java.io.File` and related objects can create, read, and remove files on the external storage path, typically `/sdcard`, assuming it's available. You can acquire a `File` for this location by using the method `Environment.getExternalStorageDirectory()`. The following listing shows how to check that the SD card's path is present, create another subdirectory inside, and then write and subsequently read file data at that location.

Listing 5.9. Using standard `java.io.File` techniques with an SD card

```
public class ReadWriteSDCardFile extends Activity {
    private TextView readOutput;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_write_sdcard_file);
        readOutput = (TextView)
            findViewById(R.id.readwritesd_output);
        String fileName = "testfile-"
            + System.currentTimeMillis() + ".txt";
        File sdDir = Environment.getExternalStorageDirectory();
        if (sdDir.exists() && sdDir.canWrite()) {
            File uadDir = new File(sdDir.getAbsolutePath()
                + "/unlocking_android");
            uadDir.mkdir();
            if (uadDir.exists() && uadDir.canWrite()) {
                File file = new File(uadDir.getAbsolutePath()
                    + "/" + fileName);
                try {
                    file.createNewFile();
                } catch (IOException e) {
                    // log and or handle
                }
                if (file.exists() && file.canWrite()) {
                    FileOutputStream fos = null;
                    try {
                        fos = new FileOutputStream(file);
                        fos.write("I fear you speak upon the rack,"
                            + "where men enforced do speak "
                            + "anything.".getBytes());
                    } catch (FileNotFoundException e) {
                        Log.e(ReadWriteSDCardFile.LOGTAG, "ERROR", e);
                    } catch (IOException e) {
                        Log.e(ReadWriteSDCardFile.LOGTAG, "ERROR", e);
                    } finally {
                        if (fos != null) {
                            try {
                                fos.flush();
                                fos.close();
                            } catch (IOException e) {
                                // swallow
                            }
                        }
                    }
                } else {

```

The diagram illustrates the five steps of file handling:

- 1 Establish filename
- 2 Get SD card directory reference
- 3 Instantiate File for path
- 4 Get reference to File
- 5 Write with FileOutputStream

Arrows point from each step to the corresponding code in the Java file.

```

        // log and or handle - error writing to file
    }
} else {
    // log and or handle -
    // unable to write to /sdcard/unlocking_android
}
} else {
    Log.e("ReadWriteSDCardFile.LOGTAG",
        "ERROR /sdcard path not available (did you create "
        + " an SD image with the mksdcard tool,"
        + " and start emulator with -sdcard "
        + <path_to_file> option?");
}

File rFile =
    new File("/sdcard/unlocking_android/" + fileName);
if (rFile.exists() && rFile.canRead()) {
    FileInputStream fis = null;
    try {
        fis = new FileInputStream(rFile);
        byte[] reader = new byte[fis.available()];
        while (fis.read(reader) != -1) {
        }
        readOutput.setText(new String(reader));
    } catch (IOException e) {
        // log and or handle
    } finally {
        if (fis != null) {
            try {
                fis.close();
            } catch (IOException e) {
                // swallow
            }
        }
    }
} else {
    readOutput.setText(
        "Unable to read/write sdcard file, see logcat output");
}
}
}

```

6 Use new File object for reading

7 Read with FileInputStream

We first define a name for the file to create 1. In this example, we append a timestamp to create a unique name each time this example application runs. After we have the filename, we create a `File` object reference to the removable storage directory 2. From there, we create a `File` reference to a new subdirectory, /sdcard/unlocking_android 3. The `File` object can represent both files and

directories. After we have the subdirectory reference, we call `mkdir()` to create it if it doesn't already exist.

With our directory structure in place, we follow a similar pattern to create the actual file.

We instantiate a reference `File` object **4** and then call `createFile()` to create a file on the filesystem. When we have the `File` and know it exists and that we're allowed to write to it, we use a `FileOutputStream` to write data into the file **5**.

After we create the file and have data in it, we create another `File` object with the full path to read the data back **6**. With the `File` reference, we then create a `FileInputStream` and read back the data that was earlier stored in the file **7**.

As you can see, working with files on the SD card resembles standard `java.io.File` code. A fair amount of boilerplate Java code is required to make a robust solution, with permissions and error checking every step of the way, and logging about what's happening, but it's still familiar and powerful. If you need to do a lot of `File` handling, you'll probably want to create some simple local utilities for wrapping the mundane tasks so you don't have to repeat them over and over again. You might want to use or port something like the Apache `commons.io` package, which includes a `FileUtils` class that handles these types of tasks and more.

The SD card example completes our exploration of the various ways to store different types of file data on the Android platform. If you have static predefined data, you can use `res/raw`; if you have XML files, you can use `res/xml`. You can also work directly with the filesystem by creating, modifying, and retrieving data in files, either in the local internal filesystem or on the SD card, if one is available.

A more complex way to deal with data—one that supports more robust and specialized ways to persist information—is to use a database, which we'll cover in the next section.

5.3. PERSISTING DATA TO A DATABASE

Android conveniently includes a built-in relational database.¹ SQLite doesn't have all the features of larger client/server database products, but it includes everything you need for local data storage. At the same time, it's quick and relatively easy to work with.

¹ Check out Charlie Collins' site for Android SQLite basics: www.screaming-penguin.com/node/7742.

In this section, we'll cover working with the built-in SQLite database system, from creating and querying a database to upgrading and working with the `sqlite3` tool

available in the adb shell. We'll demonstrate these features by expanding the WeatherReporter application from chapter 4. This application uses a database to store the user's saved locations and persists user preferences for each location. The screenshot shown in figure 5.4 displays the saved data that the user can select from; when the user selects a location, the app retrieves information from the database and shows the corresponding weather report.

Figure 5.4. The WeatherReporter Saved Locations screen, which pulls data from a SQLite database



We'll start by creating WeatherReporter's database.

5.3.1. Building and accessing a database

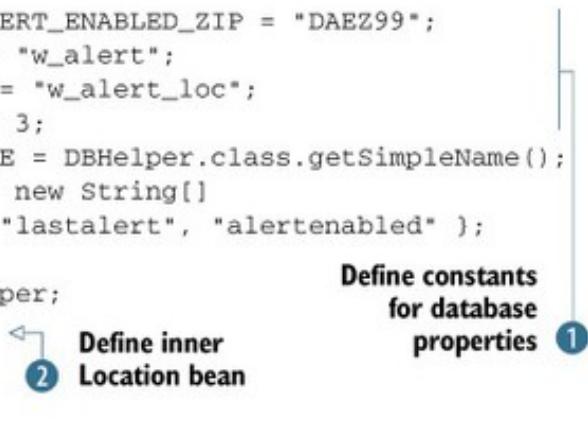
To use SQLite, you have to know a bit about SQL in general. If you need to brush up on the background of the basic commands, such as `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT`, then you might want to take a look at the SQLite documentation at www.sqlite.org/lang.html.

For now, we'll jump right in and build a database helper class for our application. You need to create a helper class so that the details concerning creating and upgrading the database, opening and closing connections, and running through specific queries are all encapsulated in one place and not otherwise exposed or repeated in your application code. Your `Activity` and `Service` classes can use simple `get` and `insert` methods, with specific bean objects representing your model, rather than database-specific abstractions such as the `AndroidCursor` object. You can think of this class as a miniature Data Access Layer (DAL).

The following listing shows the first part of our `DBHelper` class, which includes a few useful inner classes.

Listing 5.10. Portion of the `DBHelper` class showing the `DBOpenHelper` inner class

```
public class DBHelper {  
    public static final String DEVICE_ALERT_ENABLED_ZIP = "DAEZ99";  
    public static final String DB_NAME = "w_alert";  
    public static final String DB_TABLE = "w_alert_loc";  
    public static final int DB_VERSION = 3;  
    private static final String CLASSNAME = DBHelper.class.getSimpleName();  
    private static final String[] COLS = new String[]  
    { "_id", "zip", "city", "region", "lastalert", "alertenabled" };  
    private SQLiteDatabase db;  
    private final DBOpenHelper dbOpenHelper;  
    public static class Location {  
        public long id;  
        public long lastalert;  
        public int alertenabled;  
        public String zip;  
        public String city;  
        public String region;
```



1 Define constants for database properties

2 Define inner Location bean

```

    . . . Location constructors and toString omitted for brevity
}

private static class DBOpenHelper extends
SQLiteOpenHelper {

    private static final String DB_CREATE = "CREATE TABLE "
        + DBHelper.DB_TABLE
        + " (_id INTEGER PRIMARY KEY, zip TEXT UNIQUE NOT NULL,"
        + "city TEXT, region TEXT, lastalert INTEGER,"
        + "alertenabled INTEGER);"

    public DBOpenHelper(Context context, String dbName, int version) {
        super(context, DBHelper.DB_NAME, null, DBHelper.DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        try {
            db.execSQL(DBOpenHelper.DB_CREATE);
        } catch (SQLException e) {
            Log.e("ProviderWidgets", DBHelper.CLASSNAME, e);
        }
    }

    @Override
    public void onOpen(SQLiteDatabase db) {
        super.onOpen(db);
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion,
        int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + DBHelper.DB_TABLE);
        onCreate(db);
    }
}

```

Annotations:

- 3 Define inner DBOpenHelper class
- 4 Define SQL query for database creation
- 5 Override helper callbacks

Within our `DBHelper` class, we first create constants that define important values for the database we want to work with, such as its name, version, and table 1. Then we show several inner classes that we created to support the WeatherReporter application.

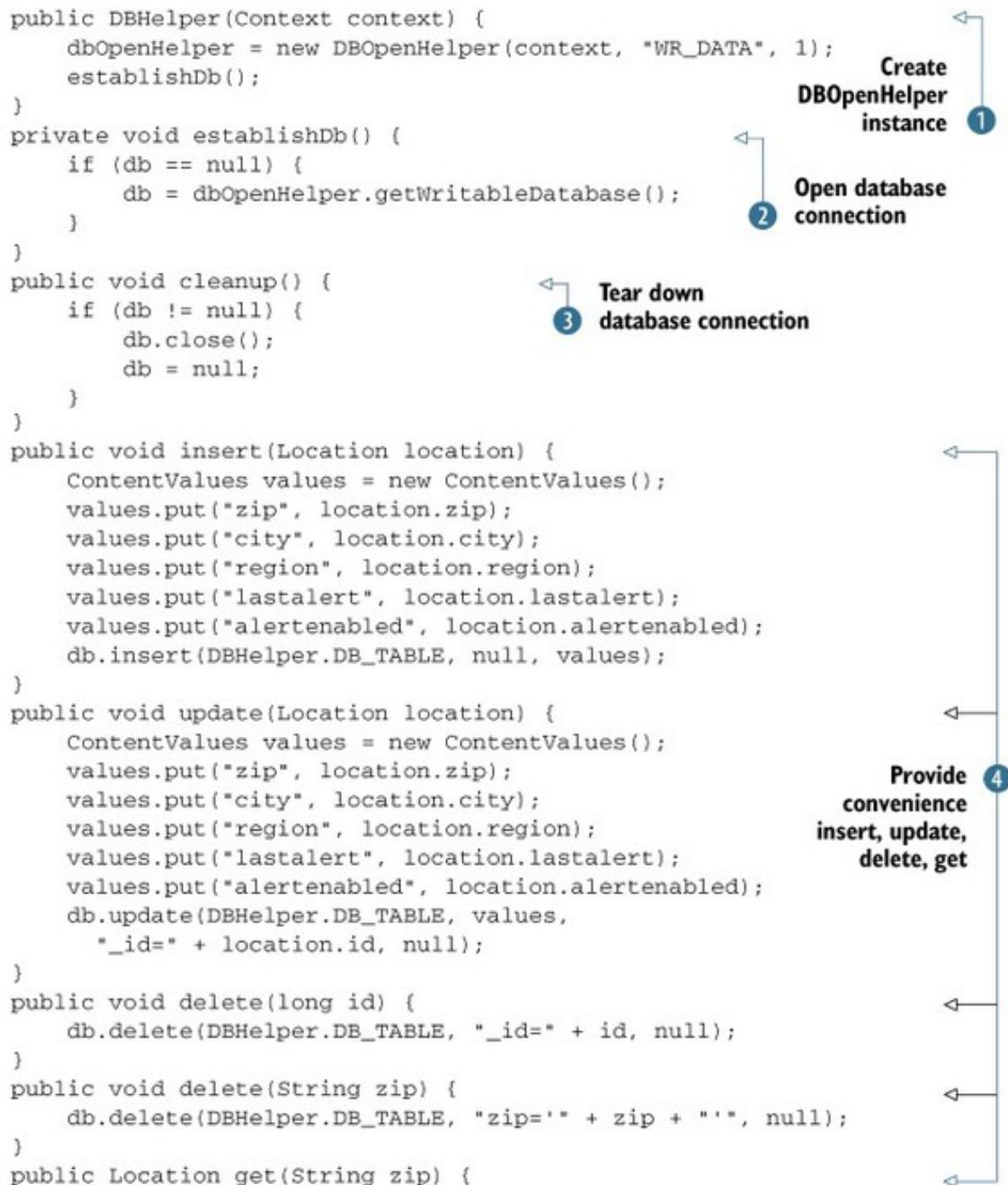
The first inner class is a simple `Location` bean that represents a user's selected location 2. This class intentionally doesn't provide accessors and mutators, because these add overhead and we don't expose the class externally. The second inner class is a `SQLiteOpenHelper` implementation 3.

Our `DBOpenHelper` inner class extends `SQLiteOpenHelper`, which Android provides to help with creating, upgrading, and opening databases. Within this class, we include

a `String` that represents the `CREATE` query we'll use to build our database table; this shows the exact columns and types our table will have   . We also implement several key `SQLiteOpenHelper` callback methods  , notably `onCreate` and `onUpgrade`. We'll explain how these callbacks are invoked in the outer part of our `DBHelper` class, which is shown in the following listing.

Listing 5.11. Portion of the `DBHelper` class showing convenience methods

```
public DBHelper(Context context) {
    dbOpenHelper = new DBOpenHelper(context, "WR_DATA", 1);
    establishDb();
}
private void establishDb() {
    if (db == null) {
        db = dbOpenHelper.getWritableDatabase();
    }
}
public void cleanup() {
    if (db != null) {
        db.close();
        db = null;
    }
}
public void insert(Location location) {
    ContentValues values = new ContentValues();
    values.put("zip", location.zip);
    values.put("city", location.city);
    values.put("region", location.region);
    values.put("lastalert", location.lastalert);
    values.put("alertenabled", location.alertenabled);
    db.insert(DBHelper.DB_TABLE, null, values);
}
public void update(Location location) {
    ContentValues values = new ContentValues();
    values.put("zip", location.zip);
    values.put("city", location.city);
    values.put("region", location.region);
    values.put("lastalert", location.lastalert);
    values.put("alertenabled", location.alertenabled);
    db.update(DBHelper.DB_TABLE, values,
              "_id=" + location.id, null);
}
public void delete(long id) {
    db.delete(DBHelper.DB_TABLE, "_id=" + id, null);
}
public void delete(String zip) {
    db.delete(DBHelper.DB_TABLE, "zip='"
              + zip + "'", null);
}
public Location get(String zip) {
```



1 Create DBOpenHelper instance

2 Open database connection

3 Tear down database connection

4 Provide convenience insert, update, delete, get

```

Cursor c = null;
Location location = null;
try {
    c = db.query(true, DBHelper.DB_TABLE, DBHelper.COLS,
        "zip = '" + zip + "'", null, null, null, null,
        null);
    if (c.getCount() > 0) {
        c.moveToFirst();
        location = new Location();
        location.id = c.getLong(0);
        location.zip = c.getString(1);
        location.city = c.getString(2);
        location.region = c.getString(3);
    }
}

```

Our `DBHelper` class contains a member-level variable reference to a `SQLiteDatabase` object, as you saw in listing 5.10. We use this object as a workhorse to open database connections, to execute SQL statements, and more.

In the constructor, we instantiate the `DBOpenHelper` inner class from the first part of the `DBHelper` class listing 1. Inside the `establishDb` method, we use `dbOpenHelper` to call `openDatabase` with the current `Context`, database name, and database version 2. `db` is established as an instance of `SQLiteDatabase` through `DBOpenHelper`.

Although you can also just open a database connection directly on your own, using the `openHelper` in this way invokes the provided callbacks and makes the process easier. With this technique, when you try to open your database connection, it's automatically created, upgraded, or just returned, through your `DBOpenHelper`. Though using a `DBOpenHelper` requires a few extra steps up front, it's extremely handy when you need to modify your table structure. You can simply increment the database's version number and take appropriate action in the `onUpgrade` callback.

Callers can invoke the `cleanup` method 3 when they pause, in order to close connections and free up resources.

After the `cleanup` method, we include the raw SQL convenience methods that encapsulate our helper's operations. In this class, we have methods to insert, update, delete, and get data 4. We also have a few additional

specialized `get` and `getAll` methods 5. Within these methods, you can see how to use the `db` object to run queries. The `SQLiteDatabase` class itself has many convenience methods, such as `insert`, `update`, and `delete`, and it provides direct `query` access that returns a `Cursor` over a result set.

Databases are application private

Unlike the `SharedPreferences` you saw earlier, you can't make a database `WORLD_READABLE`. Each database is accessible only by the package in which it was created. If you need to pass data across processes, you can use AIDL/Binder (as in chapter 4) or create a `ContentProvider` (as we'll discuss in section 5.4), but you can't use a database directly across the process/package boundary.

You can usually get a lot of mileage and utility from basic uses of the `SQLiteDatabase` class. The final aspect for us to explore is the `sqlite3` tool, which you can use to manipulate data outside your application.

5.3.2. Using the `sqlite3` tool

When you create a database for an application in Android, it creates files for that database on the device in the `/data/data/[PACKAGE_NAME]/database/db.name` location. These files are SQLite proprietary, but you can manipulate, dump, restore, and work with your databases through these files in the adb shell by using the `sqlite3` tool.

Data Permissions Most devices lock down the data directory and will not allow you to browse their content using standalone tools. Use `sqlite3` in the emulator or on a phone with firmware that allows you to access the `/data/data` directory.

You can access this tool by issuing the following commands on the command line. Remember to use your own package name; here we use the package name for the WeatherReporter sample application:

```
cd [ANDROID_HOME]/tools  
adb shell  
sqlite3 /data/data/com.msi.manning.chapter4/databases/w_alert.db
```

When you're in the shell and see the # prompt, you can then issue `sqlite3` commands. Type `.help` to get started; if you need more help, see the tool's documentation at www.sqlite.org/sqlite.html. Using the tool, you can issue basic commands, such as `SELECT` or `INSERT`, or you can go further and `CREATE` or `ALTER` tables. Use this tool to explore, troubleshoot, and `.dump` and `.load` data. As with many command-line SQL tools, it takes some time to get used to the format, but it's the best way to back up or

load your data. Keep in mind that this tool is available only through the development shell; it's not something you can use to load a real application with data.

Now that we've shown you how to use the SQLite support provided in Android, you can do everything from creating and accessing tables to investigating databases with the provided tools in the shell. Next we'll examine the last aspect of handling data on the platform: building and using a `ContentProvider`.

5.4. WORKING WITH CONTENTPROVIDER CLASSES

A *content provider* in Android shares data between applications. Each application usually runs in its own process. By default, applications can't access the data and files of other applications. We explained earlier that you can make preferences and files available across application boundaries with the correct permissions and if each application knows the context and path. This solution applies only to related applications that already know details about one another. In contrast, with a content provider you can publish and expose a particular data type for other applications to query, add, update, and delete, and those applications don't need to have any prior knowledge of paths, resources, or who provides the content.

The canonical content provider in Android is the contacts list, which provides names, addresses, and phone numbers. You can access this data from any application by using the correct URI and a series of methods provided by the `Activity` and `ContentResolver` classes to retrieve and store data. You'll learn more about `ContentResolver` as we explore provider details. One other data-related concept that a content provider offers is the `Cursor`, the same object we used previously to process SQLite database result sets.

In this section, you'll build another application that implements its own content provider and includes a similar explorer-type `Activity` to manipulate that data.

Note

For a review of content providers, please see chapter 1. You can also find a complete example of working with the `Contacts` content provider in chapter 15.

To begin, we'll explore the syntax of URIs and the combinations and paths used to perform different types of operations with the `ContentProvider` and `ContentResolver` classes.

5.4.1. Using an existing ContentProvider

Each `ContentProvider` class exposes a unique `CONTENT_URI` that identifies the content type it'll handle. This URI can query data in two forms, singular or plural, as shown in table 5.1.

Table 5.1. `ContentProvider` URI variations for different purposes

URI	Purpose
<code>content://food/ingredients/</code>	Returns a List of all ingredients from the provider registered to handle <code>content://food</code>
<code>content://food/meals/</code>	Returns a List of all meals from the provider registered to handle <code>content://food</code>
<code>content://food/meals/1</code>	Returns or manipulates a single meal with ID 1 from the provider registered to handle <code>content://food</code>

Managed Cursor

To obtain a `Cursor` reference, you can also use the `managedQuery` method of the `Activity` class. The `Activity` automatically cleans up any managed `Cursor` objects when your `Activity` pauses and restarts them when it starts. If you just need to retrieve data within an `Activity`, you'll want to use a managed `Cursor`, as opposed to a `ContentResolver`.

A provider can offer as many types of data as it likes. By using these formats, your application can either iterate through all the content offered by a provider or retrieve a specific datum of interest.

The `Activity` class has a `managedQuery()` method that makes calls into registered `ContentProvider` classes. When you create your own content provider in section 5.4.2, we'll show you how a provider is registered with the platform. Each provider is required to advertise the `CONTENT_URI` it supports. To query the contacts provider, you have to know this URI and then get a `Cursor` by calling `managedQuery()`. When you have the `Cursor`, you can use it, as we showed you in listing 5.11.

A `ContentProvider` typically supplies all the details of the URI and the types it supports as constants in a class. In the `android.provider` package, you can find classes that correspond to built-in Android content providers, such as the `MediaStore`. These classes have nested inner classes that represent types of data, such as `Audio` and `Images`. Within those classes are additional inner classes, with constants that represent fields or columns of data for each type. The values you need to query and manipulate data come from the inner classes for each type.

For additional information, see the `android.provider` package in the Javadocs, which lists all the built-in providers. Now that we've covered a bit about using a provider, we'll look at the other side of the coin—creating a content provider.

What if the content changes after the fact?

When you use a `ContentProvider` to make a query, you get only the current state of the data. The data could change after your call, so how do you stay up to date? To receive notifications when a `Cursor` changes, you can use the `ContentObserver` API. `ContentObserver` supports a set of callbacks that trigger when data changes. The `Cursor` class provides `register()` and `unregister()` methods for `ContentObserver` objects.

5.4.2. Creating a ContentProvider

In this section, you'll build a provider that handles data responsibilities for a generic `Widget` object you'll define. This simple object includes a name, type, and category; in a real application, you could represent any type of data.

To start, define a provider constants class that declares the `CONTENT_URI` and `MIME_TYPE` your provider will support. In addition, you can place the column names your provider will handle here.

Defining a `CONTENT_URI` and `MIME_TYPE`

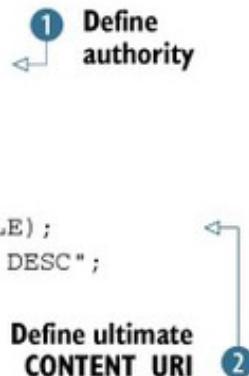
In the following listing, as a prerequisite to extending the `ContentProvider` class for a custom provider, we define necessary constants for our `Widget` type.

Listing 5.12. `WidgetProvider` constants, including columns and URI

```

public final class Widget implements BaseColumns {
    public static final String MIME_DIR_PREFIX =
        "vnd.android.cursor.dir";
    public static final String MIME_ITEM_PREFIX =
        "vnd.android.cursor.item";
    public static final String MIME_ITEM = "vnd.msi.widget";
    public static final String MIME_TYPE_SINGLE =
        MIME_ITEM_PREFIX + "/" + MIME_ITEM;
    public static final String MIME_TYPE_MULTIPLE =
        MIME_DIR_PREFIX + "/" + MIME_ITEM;
    public static final String AUTHORITY =
        "com.msi.manning.chapter5.Widget";
    public static final String PATH_SINGLE = "widgets/#";
    public static final String PATH_MULTIPLE = "widgets";
    public static final Uri CONTENT_URI =
        Uri.parse("content://" + AUTHORITY + "/" + PATH_MULTIPLE);
    public static final String DEFAULT_SORT_ORDER = "updated DESC";
    public static final String NAME = "name";
    public static final String TYPE = "type";
    public static final String CATEGORY = "category";
    public static final String CREATED = "created";
    public static final String UPDATED = "updated";
}

```



In our `Widget`-related provider constants class, we first extend the `BaseColumns` class. Now our class has a few base constants, such as `_ID`. Next, we define the `MIME_TYPE` prefix for a set of multiple items and a single item. By convention, `vnd.android.cursor.dir` represents multiple items, and `vnd.android.cursor.item` represents a single item. We can then define a specific `MIME` item and combine it with the single and multiple paths to create two `MIME_TYPE` representations.

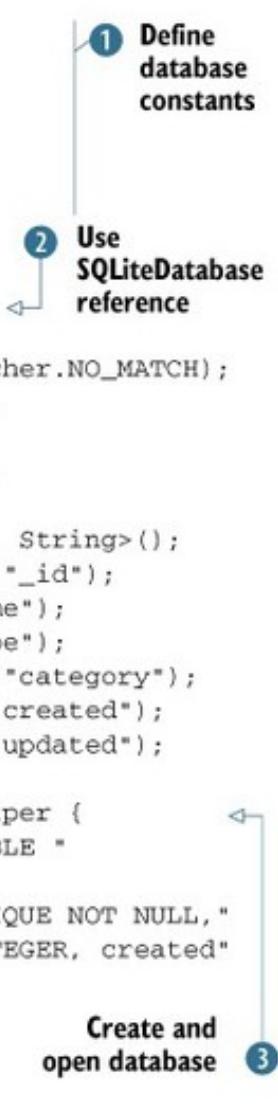
After we have the `MIME` details out of the way, we define the authority ① and path for both single and multiple items that will be used in the `CONTENT_URI` that callers pass in to use our provider. Callers will ultimately start from the multiple-item URI, so we publish this one ②.

After taking care of all the other details, we define column names that represent the variables in our `Widget` object, which correspond to fields in the database table we'll use. Callers will use these constants to get and set specific fields. Now we're on to the next part of the process, extending `ContentProvider`.

The following listing shows the beginning of our `ContentProvider` implementation class, `WidgetProvider`. In this part of the class, we do some housekeeping relating to the database we'll use and the URI we're supporting.

Listing 5.13. The first portion of the `WidgetProvider ContentProvider`

```
public class WidgetProvider extends ContentProvider {
    private static final String CLASSNAME =
        WidgetProvider.class.getSimpleName();
    private static final int WIDGETS = 1;
    private static final int WIDGET = 2;
    public static final String DB_NAME = "widgets_db";
    public static final String DB_TABLE = "widget";
    public static final int DB_VERSION = 1;
    private static UriMatcher URI_MATCHER = null;
    private static HashMap<String, String> PROJECTION_MAP;
    private SQLiteDatabase db;
    static {
        WidgetProvider.URI_MATCHER = new UriMatcher(UriMatcher.NO_MATCH);
        WidgetProvider.URI_MATCHER.addURI(Widget.AUTHORITY,
            Widget.PATH_MULTIPLE, WidgetProvider.WIDGETS);
        WidgetProvider.URI_MATCHER.addURI(Widget.AUTHORITY,
            Widget.PATH_SINGLE, WidgetProvider.WIDGET);
        WidgetProvider.PROJECTION_MAP = new HashMap<String, String>();
        WidgetProvider.PROJECTION_MAP.put(BaseColumns._ID, "_id");
        WidgetProvider.PROJECTION_MAP.put(Widget.NAME, "name");
        WidgetProvider.PROJECTION_MAP.put(Widget.TYPE, "type");
        WidgetProvider.PROJECTION_MAP.put(Widget.CATEGORY, "category");
        WidgetProvider.PROJECTION_MAP.put(Widget.CREATED, "created");
        WidgetProvider.PROJECTION_MAP.put(Widget.UPDATED, "updated");
    }
    private static class DBOpenHelper extends SQLiteOpenHelper {
        private static final String DB_CREATE = "CREATE TABLE "
            + WidgetProvider.DB_TABLE
            + " (_id INTEGER PRIMARY KEY, name TEXT UNIQUE NOT NULL,"
            + " type TEXT, category TEXT, updated INTEGER, created"
            + " INTEGER);";
        public DBOpenHelper(Context context) {
            super(context, WidgetProvider.DB_NAME, null,
                WidgetProvider.DB_VERSION);
        }
    }
}
```



1 Define database constants

2 Use SQLiteDatabase reference

3 Create and open database

```

@Override
public void onCreate(SQLiteDatabase db) {
    try {
        db.execSQL(DBOpenHelper.DB_CREATE);
    } catch (SQLException e) {
        // log and/or handle
    }
}
@Override
public void onOpen(SQLiteDatabase db) {
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
    int newVersion) {
    db.execSQL("DROP TABLE IF EXISTS " +
        + WidgetProvider.DB_TABLE);
    onCreate(db);
}
@Override
public boolean onCreate() {
    DBOpenHelper dbHelper = new DBOpenHelper(getContext());
    db = dbHelper.getWritableDatabase();
    if (db == null) {
        return false;
    } else {
        return true;
    }
}
@Override
public String getType(Uri uri) {
    switch (WidgetProvider.URI_MATCHER.match(uri)) {
        case WIDGETS:
            return Widget.MIME_TYPE_MULTIPLE;
        case WIDGET:
            return Widget.MIME_TYPE_SINGLE;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }
}

```

 4 **Override
onCreate**

 5 **Implement
getType method**

Our provider extends `ContentProvider`, which defines the methods we'll need to implement. We use several database-related constants to define the database name and table we'll use  . After that, we include a `UriMatcher`, which we'll use to match types, and a projection `Map` for field names.

We include a reference to a `SQLiteDatabase` object; we'll use this to store and retrieve the data that our provider handles   We create, open, or upgrade the database using a `SQLiteOpenHelper` in an inner class  . We've used this helper pattern before, when we worked directly with the database in listing 5.10. In the `onCreate()` method, the open helper sets up the database .

After our setup-related steps, we come to the first method `ContentProvider` requires us to implement, `getType()`  . The provider uses this method to resolve each passed-in URI to determine whether it's supported. If it is, the method checks which type of data the current call is requesting. The data might be a single item or the entire set.

Next, we need to cover the remaining required methods to satisfy the `ContentProvider` contract. These methods, shown in the following listing, correspond to the CRUD-related activities: `query`, `insert`, `update`, and `delete`.

Listing 5.14. The second portion of the `WidgetProvider ContentProvider`

```
@Override
public Cursor query(Uri uri, String[] projection,
    String selection, String[] selectionArgs,
    String sortOrder) {
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
    String orderBy = null;
    switch (WidgetProvider.URI_MATCHER.match(uri)) {
        case WIDGETS:
            queryBuilder.setTables(WidgetProvider.DB_TABLE);
            queryBuilder.setProjectionMap(WidgetProvider.PROJECTION_MAP);
            break;
        case WIDGET:
            queryBuilder.setTables(WidgetProvider.DB_TABLE);
            queryBuilder.appendWhere("_id=" +
                + uri.getPathSegments().get(1));
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }
    if (TextUtils.isEmpty(sortOrder)) {
        orderBy = Widget.DEFAULT_SORT_ORDER;
    } else {
        orderBy = sortOrder;
    }
```

Use query builder 

Set up query based on URI 

```

        Cursor c = queryBuilder.query(db, projection,
            selection, selectionArgs, null, null,
            orderBy);
        c.setNotificationUri(
            getContext().getContentResolver(), uri);
        return c;
    }
    @Override
    public Uri insert(Uri uri, ContentValues initialValues) {
        long rowId = 0L;
        ContentValues values = null;
        if (initialValues != null) {
            values = new ContentValues(initialValues);
        } else {
            values = new ContentValues();
        }
        if (WidgetProvider.URI_MATCHER.match(uri) !=
            WidgetProvider.WIDGETS) {
            throw new IllegalArgumentException("Unknown URI " + uri);
        }
        Long now = System.currentTimeMillis();
        . . . omit defaulting of values for brevity
        rowId = db.insert(WidgetProvider.DB_TABLE, "widget_hack",
            values);
        if (rowId > 0) {
            Uri result = ContentUris.withAppendedId(Widget.CONTENT_URI,
                rowId);
            getContext().getContentResolver().
                notifyChange(result, null);
        }
    }
}

```

The diagram illustrates the 8 steps of the WidgetProvider implementation:

- 3 Perform query to get Cursor**: Points to the line `return c;` in the `query()` method.
- 4 Set notification URI on Cursor**: Points to the line `c.setNotificationUri(getContext().getContentResolver(), uri);` in the `query()` method.
- 5 Use ContentValues in insert method**: Points to the line `ContentValues values = null;` in the `insert()` method.
- 6 Call database insert**: Points to the line `rowId = db.insert(WidgetProvider.DB_TABLE, "widget_hack", values);` in the `insert()` method.
- 7 Get URI to return**: Points to the line `Uri result = ContentUris.withAppendedId(Widget.CONTENT_URI, rowId);` in the `insert()` method.
- 8 Notify listeners data was inserted**: Points to the line `getContext().getContentResolver().notifyChange(result, null);` in the `insert()` method.

The last part of our `WidgetProvider` class shows how to implement the `ContentProvider` methods. First, we use a `SQLQueryBuilder` inside the `query()` method to append the projection map passed in ① and any SQL clauses, along with the correct URI based on our matcher ②, before we make the actual query and get a handle on a `Cursor` to return ③.

At the end of the `query()` method, we use the `setNotificationUri()` method to watch the returned URI for changes ④. This event-based mechanism keeps track of when `Cursor` data items change, regardless of who changes them.

Next, you see the `insert()` method, where we validate the passed-in `ContentValues` object and populate it with default values, if the values aren't present ⑤. After we have the values, we call the database `insert()` method ⑥ and

get the resulting URI to return with the appended ID of the new record 7. After the insert is complete, we use another notification system, this time for `ContentResolver`. Because we've made a data change, we inform the `ContentResolver` what happened so that any registered listeners can be updated 8.

After completing the `insert()` method, we come to the `update()` 9 and `delete()` 10 methods. These methods repeat many of the previous concepts. First, they match the URI passed in to a single element or the set, and then they call the respective `update()` and `delete()` methods on the database object. Again, at the end of these methods, we notify listeners that the data has changed.

Implementing the needed provider methods completes our class. After we register this provider with the platform, any application can use it to query, insert, update, or delete data. Registration occurs in the application manifest, which we'll look at next.

Provider Manifests

Content providers must be defined in an application manifest file and installed on the platform so the platform can learn that they're available and what data types they offer. The following listing shows the manifest for our provider.

Listing 5.15. WidgetProvider AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.chapter5.widget">
    <application android:icon="@drawable/icon"
        android:label="@string/app_short_name">
        <activity android:name=".WidgetExplorer"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name=
                    "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider android:name="WidgetProvider"
            android:authorities=
                "com.msi.manning.chapter5.Widget" />
    </application>
</manifest>
```

1 **Declare provider's authority**

Additional ContentProvider manifest properties

The properties of a content provider can configure several important settings beyond the basics, such as specific permissions, initialization order, multiprocess capability, and more. Though most `ContentProvider` implementations won't need to delve into these details, you should still keep them in mind. For complete and up-to-date `ContentProvider` properties, see the SDK documentation.

The `<provider>` element ① defines the class that implements the provider and associates a particular authority with that class.

A completed project that supports inserting, retrieving, updating, and deleting records rounds out our exploration of using and building `ContentProvider` classes. And with that, we've also now demonstrated the ways to locally store and retrieve data on the Android platform.

5.5. SUMMARY

From a simple `SharedPreferences` mechanism to file storage, databases, and finally the concept of a content provider, Android provides myriad ways for applications to retrieve and store data.

As we discussed in this chapter, several storage types can share data across application and process boundaries, and several can't. You can create `SharedPreferences` with a permissions mode, allowing the flexibility to keep things private, or to share data globally with read-only or read-write permissions. The filesystem provides more flexible and powerful data storage for a single application.

Android also provides a relational database system based on SQLite. Use this lightweight, speedy, and capable system for local data persistence within a single application. To share data, you can still use a database, but you need to expose an interface through a content provider. Providers expose data types and operations through a URI-based approach.

In this chapter, we examined each of the data paths available to an Android application. You built several small, focused sample applications to use preferences and the filesystem, and you expanded the WeatherReporter sample application that you began in the last chapter. This Android application uses a SQLite database to access and persist data. You also built your own custom content provider from the ground up.

To expand your Android horizons beyond data, we'll move on to general networking in the next chapter. We'll cover networking basics and the networking APIs Android provides. We'll also expand on the data concepts we've covered in this chapter to use the network itself as a data source.

Chapter 6. Networking and web services

This chapter covers

- Networking basics
- Determining network status
- Using the network to retrieve and store data
- Working with web services

With the ubiquity of high-speed networking, mobile devices are now expected to perform many of the same data-rich functions of traditional computers such as email, providing web access, and the like. Furthermore, because mobile phones offer such items as GPS, microphones, CDMA/GSM, built in cameras, accelerometers, and many others, user demand for applications that leverage all the features of the phone continues to increase.

You can build interesting applications with the open `Intent`- and `service`-based approach you learned about in previous chapters. That approach combines built-in (or custom) intents, such as fully capable web browsing, with access to hardware components, such as a 3D graphics subsystem, a GPS receiver, a camera, removable storage, and more. This combination of open platform, hardware capability, software architecture, and access to network data makes Android compelling.

This doesn't mean that the voice network isn't important—we'll cover telephony explicitly in [chapter 7](#)—but we admit that voice is a commodity—and data is what we'll focus on when talking about the network.

Android provides access to networking in several ways, including mobile *Internet Protocol* (IP), Wi-Fi, and Bluetooth. It also provides some open and closed source third-party implementations of other networking standards such as ZigBee and Worldwide Interoperability for Microwave Access (WiMAX). In this chapter, though, we'll concentrate on getting your Android applications to communicate using IP network data, using several different approaches. We'll cover a bit of networking background, and then we'll deal with Android specifics as we explore communication with the network using sockets and higher-level protocols such as *Hypertext Transfer Protocol* (HTTP).

Android provides a portion of the `java.net` package and the `org.apache.httpclient` package to support basic networking. Other related

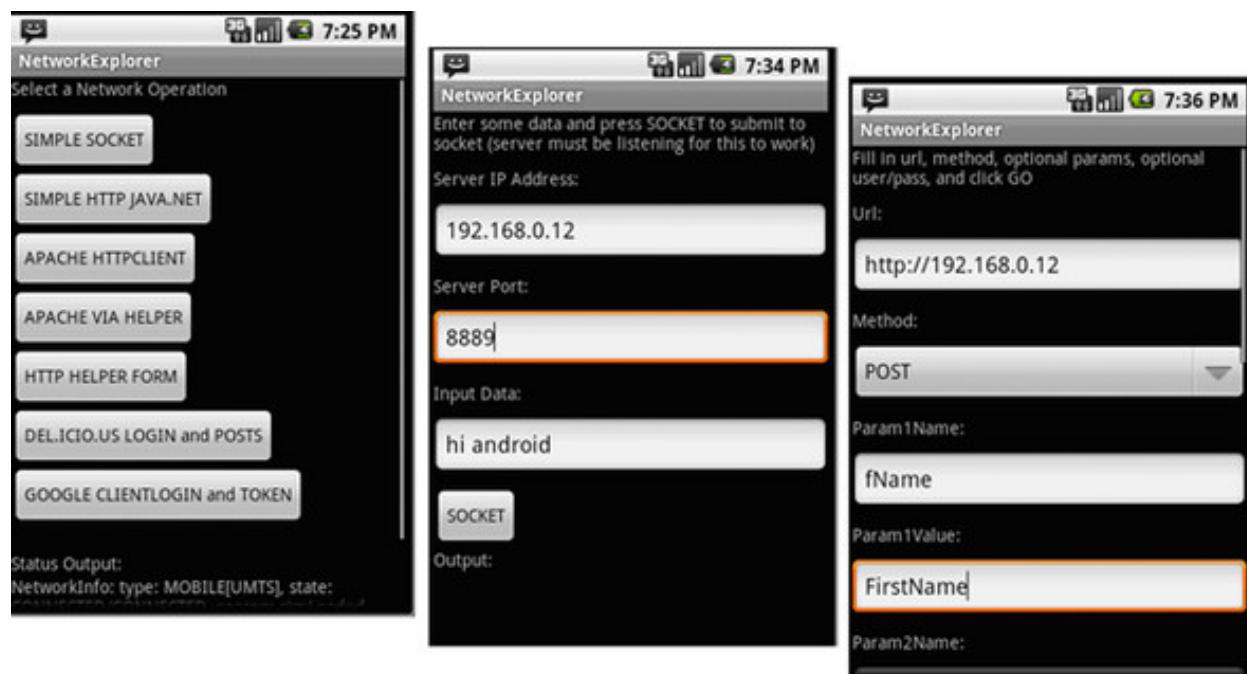
packages, such as `android.net`, address internal networking details and general connectivity properties. You'll encounter all these packages as we progress through networking scenarios in this chapter.

In terms of connectivity properties, we'll look at using the `ConnectivityManager` class to determine when the network connection is active and what type of connection it is: mobile or Wi-Fi. From there, we'll use the network in various ways with sample applications.

One caveat to this networking chapter is that we won't dig into the details concerning the Android Wi-Fi or Bluetooth APIs. Bluetooth is an important technology for close-range wireless networking between devices, but it isn't available in the Android emulator (see [chapter 14](#) for more on Bluetooth). On the other hand, Wi-Fi has a good existing API but also doesn't have an emulation layer. Because the emulator doesn't distinguish the type of network you're using and doesn't know anything about either Bluetooth or Wi-Fi, and because we think the importance lies more in how you use the network, we aren't going to cover these APIs. If you want more information on the Wi-Fi APIs, please see the Android documentation (<http://code.google.com/android/reference/android/net/wifi/package-summary.html>).

The aptly named sample application for this chapter, NetworkExplorer, will look at ways to communicate with the network in Android and will include some handy utilities. Ultimately, this application will have multiple screens that exercise different networking techniques, as shown in [figure 6.1](#).

Figure 6.1. The NetworkExplorer application you'll build to cover networking topics



After we cover general IP networking with regard to Android, we'll discuss turning the server side into a more robust API itself by using web services. On this topic, we'll work with *plain old XML over HTTP (POX)* and *Representational State Transfer* (REST). We'll also discuss the *Simple Object Access Protocol* (SOAP). We'll address the pros and cons of the various approaches and why you might want to choose one method over another for an Android client.

Before we delve into the details of networked Android applications, we'll begin with an overview of networking basics. If you're already well versed in general networking, you can skip ahead to [section 6.2](#), but it's important to have this foundation if you think you need it, and we promise to keep it short.

6.1. AN OVERVIEW OF NETWORKING

A group of interconnected computers is a *network*. Over time, networking has grown from something that was available only to governments and large organizations to the almost ubiquitous and truly amazing internet. Though the concept is simple—allow computers to communicate—networking does involve advanced technology. We won't get into great detail here, but we'll cover the core tenets as a background to the general networking you'll do in the remainder of this chapter.

6.1.1. Networking basics

A large percentage of the time, the APIs you use to program Android applications abstract the underlying network details. This is good. The APIs and the network protocols themselves are designed so that you can focus on your application and not worry about routing, reliable packet delivery, and so on.

Nevertheless, it helps to have some understanding of the way a network works so that you can better design and troubleshoot your applications. To that end, let's cover some general networking concepts, with a focus on *Transmission Control Protocol/Internet Protocol*(TCP/IP).¹ We'll begin with nodes, layers, and protocols.

¹ For an in-depth study of all things TCP/IP related, take a look at Craig Hunt's book, *TCP/IP Network Administration*, Third Edition (O'Reilly, 2002): <http://oreilly.com/catalog/9780596002978>.

Nodes

The basic idea behind a network is that data is sent between connected devices using particular addresses. Connections can be made over wire, over radio waves, and so on. Each addressed device is known as a *node*. A node can be a mainframe, a PC, a fancy toaster, or any other device with a network stack and connectivity, such as an Android-enabled handheld.

Layers and Protocols

Protocols are a predefined and agreed-upon set of rules for communication. Protocols are often layered on top of one another because they handle different levels of responsibility. The following list describes the main layers of the TCP/IP stack, which is used for the majority of web traffic and with Android:

- **Link Layer**— Physical device address resolution protocols such as ARP and RARP
- **Internet Layer**— IP itself, which has multiple versions, the `ping` protocol, and ICMP, among others
- **Transport Layer**— Different types of delivery protocols such as TCP and UDP
- **Application Layer**— Familiar protocols such as HTTP, FTP, SMTP, IMAP, POP, DNS, SSH, and SOAP

Layers are an abstraction of the different levels of a network protocol stack. The lowest level, the Link Layer, is concerned with physical devices and physical addresses. The next level, the Internet Layer, is concerned with addressing and general data details. After that, the Transport Layer is concerned with delivery details. And, finally, the top-level Application Layer protocols, which make use of the stack beneath them, are application-specific for sending files or email or viewing web pages.

IP

IP is in charge of the addressing system and delivering data in small chunks called *packets*. Packets, known in IP terms as *datagrams*, define how much data can go in each chunk, where the boundaries for payload versus header information are, and the like. IP addresses tell where each packet is from (its source) and where it's going (its destination).

IP addresses come in different sizes, depending on the version of the protocol being used, but by far the most common at present is the 32-bit address. 32-bit IP addresses (TCP/IP version 4, or IPv4) are typically written using a decimal notation that separates the 32 bits into four sections, each representing 8 bits (an octet), such as `74.125.45.100`.

Certain IP address classes have special roles and meaning. For example, `127` always identifies a *loopback*² or local address on every machine; this class doesn't communicate with any other devices (it can be used internally, on a single machine only). Addresses that begin with `10` or `192` aren't routable, meaning they can communicate with other devices on the same local network segment but can't connect to other segments. Every address on a particular network segment must be unique, or collisions can occur and it gets ugly.

² The TCP/IP Guide provides further explanation of datagrams and loopbacks: www.tcpipguide.com/index.htm.

The routing of packets on an IP network—how packets traverse the network and go from one segment to another—is handled by *routers*. Routers speak to each other using IP addresses and other IP-related information.

TCP and UDP

TCP and UDP (User Datagram Protocol) are different delivery protocols that are commonly used with TCP/IP. TCP is reliable, and UDP is fire and forget. What does that mean? It means that TCP includes extra data to guarantee the order of packets and to send back an acknowledgment when a packet is received. The common analogy is certified mail: the sender gets a receipt that shows the letter was delivered and signed for, and therefore knows the recipient got the message. UDP, on the other hand, doesn't provide any ordering or acknowledgment. It's more like a regular letter: it's cheaper and faster to send, but you basically just hope the recipient gets it.

Application Protocols

After a packet is sent and delivered, an application takes over. For example, to send an email message, Simple Mail Transfer Protocol (SMTP) defines a rigorous set of procedures that have to take place. You have to say hello in a particular way and introduce yourself; then you have to supply from and to information, followed by a message body in a particular format. Similarly, HTTP defines the set of rules for the internet—which methods are allowed (`GET`, `POST`, `PUT`, `DELETE`) and how the overall request/response system works between a client and a server.

When you're working with Android (and Java-related APIs in general), you typically don't need to delve into the details of any of the lower-level protocols, but you might need to know the major differences we've outlined here for troubleshooting. You should also be well-versed in IP addressing, know a bit more about clients and servers, and understand how connections are established using ports.

6.1.2. Clients and servers

Anyone who's ever used a web browser is familiar with the client/server computing model. Data, in one format or another, is stored on a centralized, powerful server. Clients then connect to that server using a designated protocol, such as HTTP, to retrieve the data and work with it.

This pattern is, of course, much older than the web, and it has been applied to everything from completely dumb terminals that connect to mainframes to modern desktop applications that connect to a server for only a portion of their purpose. A good example is iTunes, which is primarily a media organizer and player, but also has a store where customers can connect to the server to get new content. In any case, the concept is the same: the client makes a type of request to the server, and the server responds. This model is the same one that the majority of Android applications (at least those that

use a server side at all) generally follow. Android applications typically end up as the client.

In order to handle many client requests that are often for different purposes and that come in nearly simultaneously to a single IP address, modern server operating systems use the concept of *ports*. Ports aren't physical; they're a representation of a particular area of the computer's memory. A server can listen on multiple designated ports at a single address: for example, one port for sending email, one port for web traffic, two ports for file transfer, and so on. Every computer with an IP address also supports a range of thousands of ports to enable multiple conversations to happen at the same time.

Ports are divided into three ranges:

- **Well-known ports**— 0 through 1023
- **Registered ports**— 1024 through 49151
- **Dynamic and/or private ports**— 49152 through 65535

The well-known ports are all published and are just that—well known. HTTP is port 80 (and HTTP Secure, or HTTPS, is port 443), FTP is ports 20 (control) and 21 (data), SSH is port 22, SMTP is port 25, and so on.

Beyond the well-known ports, the registered ports are still controlled and published, but for more specific purposes. Often these ports are used for a particular application or company; for example, MySQL is port 3306 (by default). For a complete list of well-known and registered ports, see the Internet Corporation for Assigned Names and Numbers (ICANN) port-numbers document: www.iana.org/assignments/port-numbers.

The dynamic or private ports are intentionally unregistered because they're used by the TCP/IP stack to facilitate communication. These ports are dynamically registered on each computer and used in the conversation. Dynamic port 49500, for example, might be used to handle sending a request to a web server and dealing with the response. When the conversation is over, the port is reclaimed and can be reused locally for any other data transfer.

Clients and servers communicate as nodes with addresses, using ports, on a network that supports various protocols. The protocols Android uses are based on the IP network the platform is designed to participate in and involve the TCP/IP family. Before you can build a full-on client/server Android application using the network, you need to handle the prerequisite task of determining the state of the connection.

6.2. CHECKING THE NETWORK STATUS

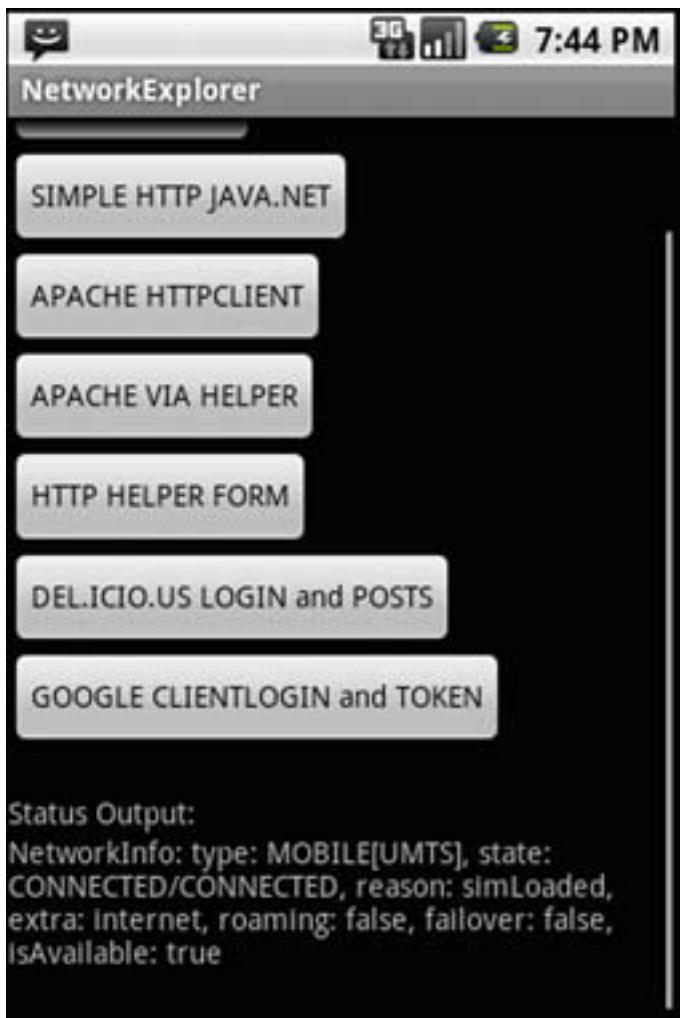
Android provides a host of utilities that determine the device configuration and the status of various services, including the network. You'll typically use the `ConnectivityManager` class to determine whether network connectivity exists and to get notifications of network changes. The following listing, which is a portion of the main Activity in the NetworkExplorer application, demonstrates basic usage of the `ConnectivityManager`.

Listing 6.1. The `onStart` method of the NetworkExplorer main Activity

```
@Override  
public void onStart() {  
    super.onStart();  
  
    ConnectivityManager cMgr = (ConnectivityManager)  
        this.getSystemService(Context.CONNECTIVITY_SERVICE);  
  
    NetworkInfo netInfo = cMgr.getActiveNetworkInfo();  
  
    this.status.setText(netInfo.toString());  
}
```

This short example shows that you can get a handle to the `ConnectivityManager` through the context's `getSystemService()` method by passing the `CONNECTIVITY_SERVICE` constant. When you have the manager, you can obtain network information via the `NetworkInfo` object. The `toString()` method of the `NetworkInfo` object returns the output shown in [figure 6.2](#).

Figure 6.2. The output of the `NetworkInfo toString()` method



Of course, you won't normally just display the `String` output from `NetworkInfo`, but this example does give you a glance at what's available. More often, you'll use the `isAvailable()` or `isConnected()` method (which returns a `boolean` value), or you'll directly query the `NetworkInfo.State` using the `getState()` method. `NetworkInfo.State` is an `enum` that defines the coarse state of the connection. The possible values are `CONNECTED`, `CONNECTING`, `DISCONNECTED`, and `DISCONNECTING`. The `NetworkInfo` object also provides access to more detailed information, but you won't normally need more than the basic state.

When you know that you're connected, either via mobile or Wi-Fi, you can use the IP network. For the purposes of our `NetworkExplorer` application, we're going to start with the most rudimentary IP connection, a raw socket, and work our way up to HTTP and web services.

6.3. COMMUNICATING WITH A SERVER SOCKET

A server socket is a stream that you can read or write raw bytes to, at a specified IP address and port. You can deal with data and not worry about media types, packet sizes, and so on. A server socket is yet another network abstraction intended to make the programmer's job a bit easier. The philosophy that sockets take on—that everything should look like file input/output (I/O) to the developer—comes from the Portable Operating System Interface for UNIX (POSIX) family of standards and has been adopted by most major operating systems in use today.

We'll move on to higher levels of network communication in a bit, but we'll start with a raw socket. For that, we need a server listening on a particular port.

The `EchoServer` code shown in the next listing fits the bill. This example isn't an Android-specific class; rather, it's an oversimplified server that can run on any host machine with Java. We'll connect to it later from an Android client.

Listing 6.2. A simple echo server for demonstrating socket usage

```
public final class EchoServer extends Thread {  
    private static final int PORT = 8889;  
    private EchoServer() {}  
    public static void main(String args[]) {  
        EchoServer echoServer = new EchoServer();  
        if (echoServer != null) {  
            echoServer.start();  
        }  
    }  
    public void run() {  
        try {  
            ServerSocket server = new ServerSocket(PORT, 1);  
            while (true) {  
                Socket client = server.accept();  
            }  
        } catch (Exception e) {}  
    }  
}
```

Use
java.net.ServerSocket

1

```

        System.out.println("Client connected");
        while (true) {
            BufferedReader reader =
                new BufferedReader(new InputStreamReader(
                    client.getInputStream()));
            System.out.println("Read from client");
            String textLine = reader.readLine() + "\n";
            if (textLine.equalsIgnoreCase("EXIT\n")) {
                System.out.println("EXIT invoked, closing client");
                break;
            }
            BufferedWriter writer = new BufferedWriter(
                new OutputStreamWriter(
                    client.getOutputStream()));
            System.out.println("Echo input to client");
            writer.write("ECHO from server: "
                + textLine, 0, textLine.length() + 18);
            writer.flush();
        }
        client.close();
    }
} catch (IOException e) {
    System.err.println(e);
}
}
}

```

Read input with
BufferedReader ②

EXIT, break
the loop ③

The `EchoServer` class we're using is fairly basic Java I/O. It extends `Thread` and implements `run`, so that each client that connects can be handled in its own context.

Then we use a `ServerSocket` ① to listen on a defined port. Each client is then an implementation of a `Socket`. The client input is fed into a `BufferedReader` that each line is read from ②. The only special consideration this simple server has is that if the input is `EXIT`, it breaks the loops and exits ③. If the input doesn't prompt an exit, the server echoes the input back to the client's `OutputStream` with a `BufferedWriter`.

This example is a good, albeit intentionally basic, representation of what a server does. It handles input, usually in a separate thread, and then responds to the client, based on the input. To try out this server before using Android, you can telnet to the specified port (after the server is running, of course) and type some input; if all is well, it will echo the output.

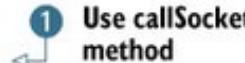
To run the server, you need to invoke it locally with Java. The server has a main method, so it'll run on its own; start it from the command line or from your IDE. Be aware that when you connect to a server from the emulator (this one or any other), you need to connect to the IP address of the host you run the server process on, not the loopback (not `127.0.0.1`). The emulator thinks of itself as `127.0.0.1`, so use the nonloopback

address of the server host when you attempt to connect from Android. (You can find out the IP address of the machine you’re on from the command line by entering `ifconfig` on Linux or Mac and `ipconfig` on Windows.)

The client portion of this example is where NetworkExplorer itself begins, with the `callSocket()` method of the `SimpleSocket Activity`, shown in the next listing.

Listing 6.3. An Android client invoking a raw socket server resource, the echo server

```
public class SimpleSocket extends Activity {  
    . . . View variable declarations omitted for brevity  
    @Override  
    public void onCreate(final Bundle icicle) {  
        super.onCreate(icicle);  
        this.setContentView(R.layout.simple_socket);  
        . . . View inflation omitted for brevity  
        this.socketButton.setOnClickListener(new OnClickListener() {  
            public void onClick(final View v) {  
                socketOutput.setText("");  
                String output = callSocket(  
                    ipAddress.getText().toString(),  
                    port.getText().toString(),  
                    socketInput.getText().toString());  
                socketOutput.setText(output);  
            }  
        });  
    }  
}
```



1 Use `callSocket` method

```

private String callSocket(String ip, String port, String socketData) {
    Socket socket = null;
    BufferedWriter writer = null;
    BufferedReader reader = null;
    String output = null;
    try {
        socket = new Socket(ip, Integer.parseInt(port));
        writer = new BufferedWriter(
            new OutputStreamWriter(
                socket.getOutputStream()));
        reader = new BufferedReader(
            new InputStreamReader(
                socket.getInputStream()));

        String input = socketData;
        writer.write(input + "\n", 0, input.length() + 1);
        writer.flush();
        output = reader.readLine();
        this.socketOutput.setText(output);
        // send EXIT and close
        writer.write("EXIT\n", 0, 5);
        writer.flush();
        . . . catches and reader, writer, and socket closes omitted for brevity
        . . . onCreate omitted for brevity
        return output;
    }
}

```

The code is annotated with four numbered steps:

- 2 Create client Socket**: Points to the line `socket = new Socket(ip, Integer.parseInt(port));`
- 3 Write to socket**: Points to the line `writer.write(input + "\n", 0, input.length() + 1);`
- 4 Get socket output**: Points to the line `output = reader.readLine();`
- Return output**: Points to the final line `return output;`

In this listing, we use the `onCreate()` method to call a private helper `callSocket()` method ① and set the output to a `TextView`. Within the `callSocket()` method, we create a socket to represent the client side of our connection ②, and we establish a writer for the input and a reader for the output. With the housekeeping taken care of, we then write to the socket ③, which communicates with the server, and get the output value to return ④.

A socket is probably the lowest-level networking usage in Android you'll encounter. Using a raw socket, though abstracted a great deal, still leaves many of the details up to you, especially the server-side details of threading and queuing. Although you might run up against situations in which you either have to use a raw socket (the server side is already built) or elect to use one for one reason or another, higher-level solutions such as leveraging HTTP usually have decided advantages.

6.4. WORKING WITH HTTP

As we discussed in the previous section, you can use a raw socket to transfer IP data to and from a server with Android. This approach is an important one to be aware of so that you know you have that option and understand a bit about the underlying details. Nevertheless, you might want to avoid this technique when possible, and instead take advantage of existing server products to send your data. The most common way to do this is to use a web server and HTTP.

Now we're going to take a look at making HTTP requests from an Android client and sending them to an HTTP server. We'll let the HTTP server handle all the socket details, and we'll focus on our client Android application.

The HTTP protocol itself is fairly involved. If you're unfamiliar with it or want the complete details, information is readily available via Requests for Comments (RFCs) (such as for version 1.1: www.w3.org/Protocols/rfc2616/rfc2616.html). The short story is that the protocol is stateless and involves several different methods that allow users to make requests to servers, and those servers return responses. The entire web is, of course, based on HTTP. Beyond the most basic concepts, there are ways to pass data into and out of requests and responses and to authenticate with servers. Here we're going to use some of the most common methods and concepts to talk to network resources from Android applications.

To begin, we'll retrieve data using HTTP `GET` requests to a simple HTML page, using the standard `java.net` API. From there, we'll look at using the Android-included Apache HttpClient API. After we use HttpClient directly to get a feel for it, we'll also make a helper class, `HttpRequestHelper`, that you can use to simplify the process and encapsulate the details. This class—and the Apache networking API in general—has a few advantages over rolling your own networking with `java.net`, as you'll see. When the helper class is in place, we'll use it to make additional HTTP and HTTPS requests, both `GET` and `POST`, and we'll look at basic authentication.

Our first HTTP request will be an HTTP `GET` call using an `HttpURLConnection`.

6.4.1. Simple HTTP and `java.net`

The most basic HTTP request method is `GET`. In this type of request, any data that's sent is embedded in the URL, using the query string. The next class in our NetworkExplorer application, which is shown in the following listing, has an `Activity` that demonstrates the `GET` request method.

Listing 6.4. The `SimpleGet` Activity showing `java.net.UrlConnection`

```
public class SimpleGet extends Activity {
    . . . other portions of onCreate omitted for brevity
    this.getButton.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
            getOutput.setText("");
            String output =
                getHttpResponse(getInput.getText().toString());
            if (output != null) {
                getOutput.setText(output);
            }
        }
    });
    . .
    private String getHttpResponse(String location) {
        String result = null;
        URL url = null;
        try {
            url = new URL(location);
        } catch (MalformedURLException e) {
            // log and or handle
        }

        if (url != null) {
            try {
                HttpURLConnection urlConn =
                    (HttpURLConnection) url.openConnection();
                BufferedReader in =
                    new BufferedReader(
                        new InputStreamReader(
                            urlConn.getInputStream()));
                String inputLine;
                int lineCount = 0; // limit lines for example
                while ((lineCount < 10)
                    && ((inputLine = in.readLine()) != null)) {
                    lineCount++;
                    result += "\n" + inputLine;
                }
                in.close();
                urlConn.disconnect();
            } catch (IOException e) {
                // log and or handle
            }
        } else {
            // log and or handle
        }
        return result;
    }
}
```

1 Invoke **getHttpResponse** method

2 Construct URL object

3 Open connection using **HttpURLConnection**

4 Read data

5 Append to result

To get an HTTP response and show the first few lines of it in our `SimpleGet` class, we call a `getHttpResponse()` method that we've built ¹. Within this method, we construct a `java.net.URL` object ², which takes care of many of the details for us, and then we open a connection to a server using an `HttpURLConnection` ³.

We then use a `BufferedReader` to read data from the connection one line at a time ⁴. Keep in mind that as we're doing this, we're using the same thread as the UI and therefore blocking the UI. This isn't a good idea. We're using the same thread here only to demonstrate the network operation; we'll explain more about how to use a separate thread shortly. After we have the data, we append it to the result `String` that our method returns ⁵, and we close the reader and the connection. Using the plain and simple `java.net` support that has been ported to Android this way provides quick and dirty access to HTTP network resources.

Communicating with HTTP this way is fairly easy, but it can quickly get cumbersome when you need to do more than just retrieve simple data, and, as noted, the blocking nature of the call is bad form. You could get around some of the problems with this approach on your own by spawning separate threads and keeping track of them and by writing your own small framework/API structure around that concept for each HTTP request, but you don't have to. Fortunately, Android provides another set of APIs in the form of the Apache HttpClient³ library that abstract the `java.net` classes further and are designed to offer more robust HTTP support and help handle the separate-thread issue.

³ You'll find more about the Apache HttpClient here: <http://hc.apache.org/httpclient-3.x/>.

6.4.2. Robust HTTP with HttpClient

To get started with HttpClient, we're going to look at using core classes to perform HTTP `GET` and `POST` method requests. We're going to concentrate on making network requests in a `Thread` separate from the UI, using a combination of the Apache `ResponseHandler` and Android `Handler` (for different but related purposes, as you'll see). The following listing shows our first example of using the HttpClient API.

Listing 6.5. Apache HttpClient with Android Handler and Apache ResponseHandler

```
    . . .
private final Handler handler = new Handler() {
    public void handleMessage(Message msg) {
        progressDialog.dismiss();
        String bundleResult =
            msg.getData().getString("RESPONSE");
        output.setText(bundleResult);
    }
};

. . . onCreate omitted for brevity
private void performRequest() {
    final ResponseHandler<String> responseHandler =
        new ResponseHandler<String>() {
            public String handleResponse(HttpResponse response) {
                StatusLine status = response.getStatusLine();
                HttpEntity entity = response.getEntity();
                String result = null;
                try {
                    result = StringUtils.inputStreamToString(
                        entity.getContent());
                    Message message = handler.obtainMessage();
                    Bundle bundle = new Bundle();
                    bundle.putString("RESPONSE", result);
                    message.setData(bundle);
                    handler.sendMessage(message);
                } catch (IOException e) {
                    // log and or handle
                }
            }
        };
}
```

1 Use Handler
to update UI

2 Create
ResponseHandler
for asynchronous
HTTP

3 Get HTTP
response
payload

```

        return result;
    }
};

this.progressDialog =
    ProgressDialog.show(this, "working . . .",
        "performing HTTP request");
new Thread() {
    public void run() {
        try {
            DefaultHttpClient client = new DefaultHttpClient();
            HttpGet httpMethod =
                new HttpGet(
                    urlChooser.getSelectedItem().toString());
            client.execute(
                httpMethod, responseHandler);
        } catch (ClientProtocolException e) {
            // log and or handle
        } catch (IOException e) {
            // log and or handle
        }
    }
}.start();
}

```

Use separate Thread for HTTP call

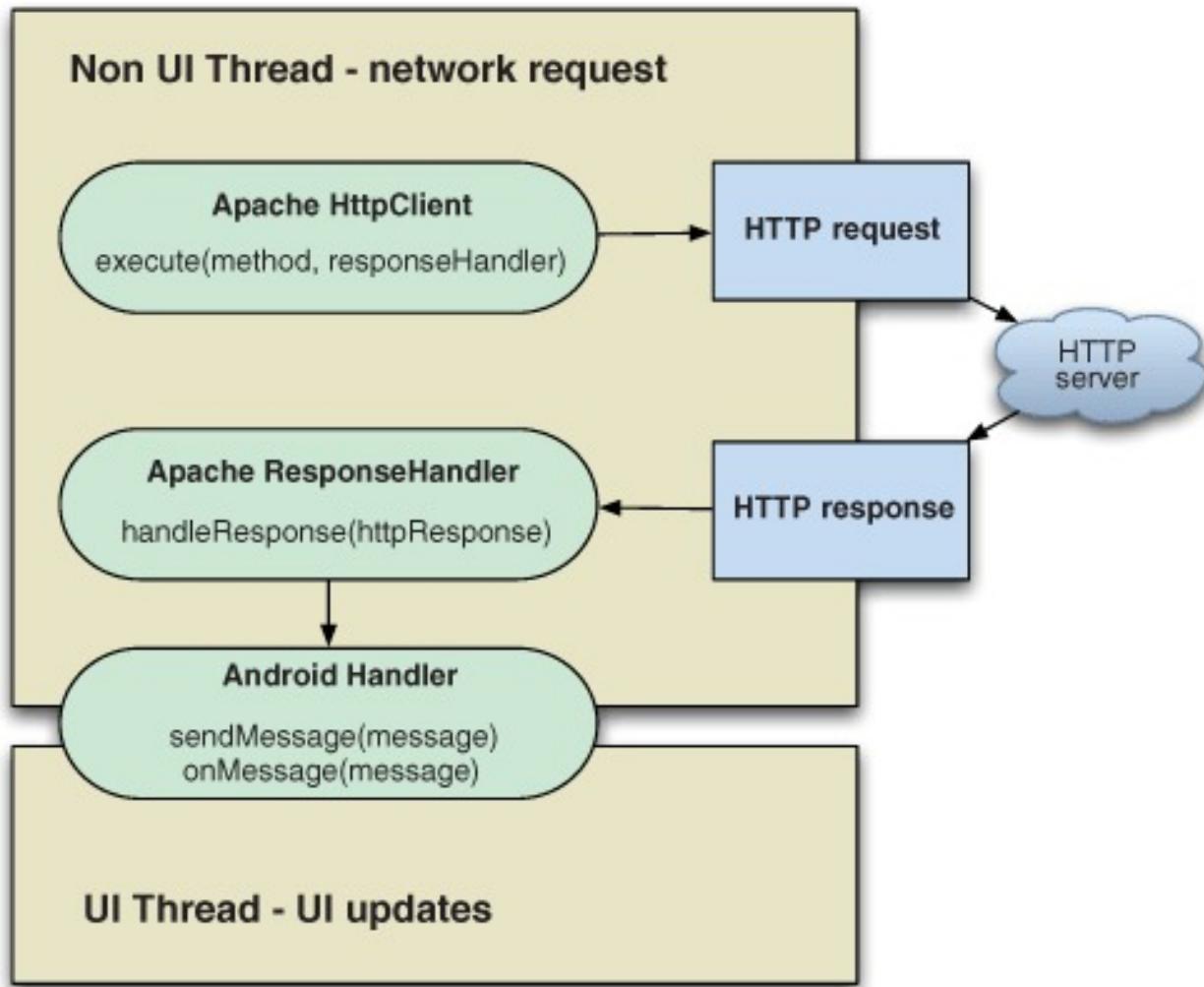
Create HttpGet object

Execute HTTP with HttpClient

The first thing we do in our initial `HttpClient` example is create a `Handler` that we can send messages to from other threads. This technique is the same one we've used in previous examples; it allows background tasks to send `Message` objects to hook back into the main UI thread ①. After we create an Android `Handler`, we create an

Apache `ResponseHandler` ②. This class can be used with `HttpClient` HTTP requests to pass in as a callback point. When an HTTP request that's fired by `HttpClient` completes, it calls the `onResponse()` method if a `ResponseHandler` is used. When the response comes in, we get the payload using the `HttpEntity` the API returns ③. This in effect allows the HTTP call to be made in an asynchronous manner—we don't have to block and wait the entire time between when the request is fired and when it completes. The relationship of the request, response, `Handler`, `ResponseHandler`, and separate threads is diagrammed in [figure 6.3](#).

Figure 6.3. The relationship between `HttpClient`, `ResponseHandler`, and Android `Handler`



Now that you've seen `HttpClient` at work and understand the basic approach, the next thing we'll do is encapsulate a few of the details into a convenient helper class so that we can call it over and over without having to repeat a lot of the setup.

6.4.3. Creating an HTTP and HTTPS helper

The next `Activity` in our `NetworkExplorer` application, which is shown in [listing 6.6](#), is a lot more straightforward and Android-focused than our other HTTP-related classes up to this point. We've used the helper class we mentioned previously, which hides some of the complexity. We'll examine the helper class itself after we look at this first class that uses it.

[Listing 6.6. Using Apache HttpClient via a custom HttpRequestHelper](#)

```

public class ApacheHTTPViaHelper extends Activity {
    . . . other member variables omitted for brevity
    private final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            output.setText(bundleResult);
        }
    };
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        . . . view inflation and setup omitted for brevity
        this.button.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                output.setText("");
                performRequest(
                    urlChooser.getSelectedItem().toString());
            }
        });
    };
    . . . onPause omitted for brevity
    private void performRequest(String url) {
        final ResponseHandler<String> responseHandler =
            HTTPRequestHelper.getResponseHandlerInstance(
                this.handler);
        this.progressDialog =
            ProgressDialog.show(this, "working . . .",
                "performing HTTP request");
        new Thread() {
            public void run() {
                HTTPRequestHelper helper = new
                    HTTPRequestHelper(responseHandler);
                helper.performGet(url, null, null, null);
            }
        }.start();
    }
}

```

The diagram consists of three numbered callouts pointing to specific lines of code. Callout 1 points to the line `performRequest(urlChooser.getSelectedItem().toString());`. Callout 2 points to the line `final ResponseHandler<String> responseHandler =`. Callout 3 points to the line `HTTPRequestHelper helper = new`.

The first thing we do in this class is create another `Handler`. From within it, we update a UI `TextView` based on data in the `Message`. Further on in the code, in the `onCreate()` method, we call a local `performRequest()` method when the Go button is clicked, and we pass a selected `String` representing a URL ①.

Inside the `performRequest()` method, we use a static convenience method to return an `HttpClient ResponseHandler`, passing in the Android `Handler` that it'll use ②. We'll

examine the helper class next to get a look at exactly how this works, but the important part for now is that the `ResponseHandler` is created for us by the static method. With the `ResponseHandler` instance taken care of, we instantiate

an `HttpRequestHelper` instance **3** and use it to make a simple HTTP GET call (passing in only the `String URL`). Similar to what happened in [listing 6.5](#), when the request completes, the `ResponseHandler` fires the `onResponse()` method, and our `Handler` is sent a `Message`, completing the process.

The example `Activity` in [listing 6.6](#) is fairly clean and simple, and it's asynchronous and doesn't block the UI thread. The heavy lifting is taken care of by `HttpClient` itself and by the setup our custom `HttpRequestHelper` makes possible. The first part of the all-important `HttpRequestHelper`, which we'll explore in three listings ([listings 6.7, 6.8, and 6.9](#)), is shown next.

Listing 6.7. The first part of the `HttpRequestHelper` class

```
public class HttpRequestHelper {  
    private static final int POST_TYPE = 1;  
    private static final int GET_TYPE = 2;  
    private static final String CONTENT_TYPE = "Content-Type";  
    public static final String MIME_FORM_ENCODED =  
        "application/x-www-form-urlencoded";  
    public static final String MIME_TEXT_PLAIN = "text/plain";  
    private final ResponseHandler<String> responseHandler;  
    public HttpRequestHelper(ResponseHandler<String> responseHandler) {  
        this.responseHandler = responseHandler;  
    }  
    public void performGet(String url, String user, String pass,  
        final Map<String, String> additionalHeaders) {  
        performRequest(null, url, user, pass,  
            additionalHeaders, null, HttpRequestHelper.GET_TYPE);  
    }  
    public void performPost(String contentType, String url,  
        String user, String pass,  
        Map<String, String> additionalHeaders,  
        Map<String, String> params) {  
        performRequest(contentType, url, user, pass,  
            additionalHeaders, params, HttpRequestHelper.POST_TYPE);  
    }  
    public void performPost(String url, String user, String pass,  
        Map<String, String> additionalHeaders,  
        Map<String, String> params) {  
        performRequest(HttpRequestHelper.MIME_FORM_ENCODED,  
            url, user, pass,  
            additionalHeaders, params, HttpRequestHelper.POST_TYPE);  
    }  
}
```

Annotations for Listing 6.7:

- Require ResponseHandler to construct**: An annotation pointing to the constructor `public HttpRequestHelper(ResponseHandler<String> responseHandler) { this.responseHandler = responseHandler; }`.
- Provide simple GET method**: An annotation pointing to the `performGet` method.
- Provide simple POST methods**: An annotation pointing to the two `performPost` methods.

```

private void performRequest(
    String contentType,
    String url,
    String user,
    String pass,
    Map<String, String> headers,
    Map<String, String> params,
    int requestType) {

    DefaultHttpClient client = new DefaultHttpClient();
    if ((user != null) && (pass != null)) {
        client.getCredentialsProvider().setCredentials(
            AuthScope.ANY,
            new UsernamePasswordCredentials(user, pass));
    }
    final Map<String, String> sendHeaders =
        new HashMap<String, String>();
    if ((headers != null) && (headers.size() > 0)) {
        sendHeaders.putAll(headers);
    }
    if (requestType == HTTPRequestHelper.POST_TYPE) {
        sendHeaders.put(HTTPRequestHelper.CONTENT_TYPE, contentType);
    }
    if (sendHeaders.size() > 0) {
        client.addRequestInterceptor(
            new HttpRequestInterceptor() {
                public void process(
                    final HttpRequest request, final HttpContext context)
                    throws HttpException, IOException {
                    for (String key : sendHeaders.keySet()) {
                        if (!request.containsHeader(key)) {
                            request.addHeader(key,
                                sendHeaders.get(key));
                        }
                    }
                }
            });
    }
    . . . POST and GET execution in listing 6.8
}

```

Annotation 4: Handle combinations in private method. Points to the if statement where credentials are set.

Annotation 5: Use Interceptor for request headers. Points to the client.addRequestInterceptor call.

The first thing of note in the `HttpRequestHelper` class is that a `ResponseHandler` is required to be passed in as part of the constructor ①. This `ResponseHandler` will be used when the `HttpClient` request is ultimately invoked. After the constructor, you see a public HTTP GET-related method ② and several different public HTTP POST-related methods ③. Each of these methods is a wrapper around the `private performRequest()` method that can handle all the HTTP options ④.

The `performRequest()` method supports a content-type header value, URL, username, password, `Map` of additional headers, similar `Map` of request parameters, and request method type.

Inside the `performRequest()` method, a `DefaultHttpClient` is instantiated. Next, we check whether the `user` and `pass` method parameters are present; if they are, we set the request credentials with a `UsernamePasswordCredentials` type (`HttpClient` supports several types of credentials; see the Javadocs for details). At the same time as we set the credentials, we also set an `AuthScope`. The scope represents which server, port, authentication realm, and authentication scheme the supplied credentials are applicable for.

You can set any of the `HttpClient` parameters as finely or coarsely grained as you want; we're using the default `ANY` scope that matches anything. What we notably haven't set in all of this is the specific authentication scheme to use. `HttpClient` supports various schemes, including basic authentication, digest authentication, and a Windows-specific NT Lan Manager (NTLM) scheme. Basic authentication (simple username/password challenge from the server) is the default. Also, if you need to, you can use a preemptive form login for form-based authentication—submit the form you need, get the token or session ID, and set default credentials.

After the security is out of the way, we use an `HttpRequestInterceptor` to add HTTP headers  . Headers are name/value pairs, so adding the headers is pretty easy. After we have all of the properties that apply regardless of our request method type, we then add additional settings that are specific to the method. The following listing, the second part of our helper class, shows the `POST`- and `GET`-specific settings and the `execute` method.

Listing 6.8. The second part of the `HttpRequestHelper` class

```

    .
    .
    if (requestType == HTTPRequestHelper.POST_TYPE) {
        HttpPost method = new HttpPost(url);

        List<NameValuePair> nvps = null;
        if ((params != null) && (params.size() > 0)) {
            nvps = new ArrayList<NameValuePair>();
            for (String key : params.keySet()) {
                nvps.add(new BasicNameValuePair(key,
                    params.get(key)));
            }
        }
        if (nvps != null) {
            try {
                method.setEntity(
                    new UrlEncodedFormEntity(nvps, HTTP.UTF_8));
            } catch (UnsupportedEncodingException e) {
                // log and or handle
            }
        }
        execute(client, method);
    } else if (requestType == HTTPRequestHelper.GET_TYPE) {
        HttpGet method = new HttpGet(url);
        execute(client, method);
    }

    .
    .
    private void execute(HttpClient client, HttpRequestBase method) {
        BasicHttpResponse errorResponse =
            new BasicHttpResponse(
                new ProtocolVersion("HTTP_ERROR", 1, 1),
                500, "ERROR");

        try {
            client.execute(method, this.responseHandler);
        } catch (Exception e) {
            errorResponse.setReasonPhrase(e.getMessage());
        }
        try {
            this.responseHandler.handleResponse(errorResponse);
        } catch (Exception ex) {
            // log and or handle
        }
    }
}

```

1 Create
HttpPost
object

2 Add name/value
parameters

3 Call execute
method

4 Set up an
error handler

When the specified request is a POST type, we create an `HttpPost` object to deal with it 1. Then we add POST request parameters, which are another set of name/value pairs and are built with the `BasicNameValuePair` object 2. After adding the parameters,

we're ready to perform the request, which we do with our local `private execute()` method using the method object and the client .

Our `execute()` method sets up an error response handler (we want to return a response, error or not, so we set this up just in case)  and wraps the `HttpClient execute()` method, which requires a method object (either `POST` or `GET` in our case, pre-established) and a `ResponseHandler` as input. If we don't get an exception when we invoke `HttpClient execute()`, all is well and the response details are placed into the `ResponseHandler`. If we do get an exception, we populate the error handler and pass it through to the `ResponseHandler`.

We call the local private `execute()` method with the established details for either a `POST` or a `GET` request. The `GET` method is handled similarly to the `POST`, but we don't set parameters (with `GET` requests, we expect parameters encoded in the URL itself). Right now, our class supports only `POST` and `GET`, which cover 98 percent of the requests we generally need, but it could easily be expanded to support other HTTP method types.

The final part of the request helper class, shown in the following listing, takes us back to the first example ([listing 6.7](#)), which used the helper. [Listing 6.9](#) outlines exactly what the convenience `getResponseBodyInstance()` method returns (constructing our helper requires a `ResponseHandler`, and this method returns a default one).

Listing 6.9. The final part of the `HttpRequestHelper` class

```
public static ResponseHandler<String>
    getResponseBodyInstance(final Handler handler) {           ← 1 Require Handler
                                                               parameter
        final ResponseHandler<String> responseHandler =
            new ResponseHandler<String>() {
                public String handleResponse(final HttpResponse response) {
                    Message message = handler.obtainMessage();
                    Bundle bundle = new Bundle();
                    StatusLine status = response.getStatusLine();
                    HttpEntity entity = response.getEntity();
                    String result = null;
                    if (entity != null) {                                     ← 2 Get response
                                                               content as String
                        try {
                            result = StringUtils.inputStreamToString(
                                entity.getContent());
                            bundle.putString(
                                "RESPONSE", result);
                            message.setData(bundle);
                        }
                    }
                }
            }
    }
```

Put result value into Bundle

```

        handler.sendMessage(message);    ← Set Bundle as
    } catch (IOException e) {           data into Message
        bundle.putString(
            "RESPONSE", "Error - " + e.getMessage());
        message.setData(bundle);
        handler.sendMessage(message);
    }
} else {
    bundle.putString("RESPONSE", "Error - "
        + response.getStatusLine().getReasonPhrase());
    message.setData(bundle);
    handler.sendMessage(message);    ← Send Message
}                                     via Handler
}
return result;
}
};

return responseHandler;
}
}

```

As we discuss the `getResponseHandlerInstance()` method of our helper, we should note that although we find it helpful, it's entirely optional. You can still use the helper class without using this method. To do so, construct your own `ResponseHandler` and pass it in to the helper constructor, which is a perfectly plausible case.

The `getResponseHandlerInstance()` method builds a convenient

default `ResponseHandler` that hooks in a `Handler` via a parameter ① and parses the response as a `String` ②. The `response String` is sent back to the caller using the `Handler`, `Bundle`, and `Message` pattern we've seen used time and time again to pass messages between threads in our Android screens.

With the gory `HttpRequestHelper` details out of the way, and having already explored basic usage, we'll next turn to more involved uses of this class in the context of web service calls.

6.5. WEB SERVICES

The term *web services* means many different things, depending on the source and the audience. To some, it's a nebulous marketing term that's never pinned down; to others, it's a rigid and specific set of protocols and standards. We're going to tackle it as a general concept, without defining it in depth, but not leaving it entirely undefined either.

Web services are a means of exposing an API over a technology-neutral network endpoint. They're a means to call a remote method or operation that's not tied to a specific platform or vendor and get a result. By this definition, POX over the network is included; so are REST and SOAP—and so is any other method of exposing operations and data on the wire in a neutral manner.

POX, REST, and SOAP are by far the most common web services around, so they're what we'll focus on in this section. Each provides a general guideline for accessing data and exposing operations, each in a more rigorous manner than the previous. POX basically exposes chunks of XML over the wire, usually over HTTP. REST is more detailed in that it uses the concept of *resources* to define data and then manipulates them with different HTTP methods using a URL-style approach (much like the Android `Intent` system in general, which we explored in previous chapters). SOAP is the most formal of them all, imposing strict rules about types of data, transport mechanisms, and security.

All these approaches have advantages and disadvantages, and these differences are amplified on a mobile platform like Android. Though we can't possibly cover all the details here, we'll touch on the differences as we discuss each of these concepts. We'll examine using a POX approach to return recent posts from the Delicious API (formerly del.icio.us), and then we'll look at using REST with the Google GData AtomPub API. Up first is probably the most ubiquitous type of web service in use on the internet today, and therefore one you'll come across again and again when connecting Android applications—POX.

6.5.1. POX: putting it together with HTTP and XML

To work with POX, we're going to make network calls to the popular Delicious online social bookmarking site. We'll specify a username and password to log in to an HTTPS resource and return a list of recent posts, or *bookmarks*. This service returns raw XML data, which we'll parse into a JavaBean-style class and display as shown in [figure 6.4](#).

Figure 6.4. The Delicious recent posts screen from the NetworkExplorer application



The following listing shows the Delicious login and HTTPS POST Activity code from our NetworkExplorer application.

Listing 6.10. The Delicious HTTPS POX API with authentication from an Activity

```
public class DeliciousRecentPosts extends Activity {
    private static final String CLASSTAG =
        DeliciousRecentPosts.class.getSimpleName();
    private static final String URL_GET_POSTS_RECENT =
        "https://api.del.icio.us/v1/posts/recent?";
    . . . member var declarations for user, pass, output,
          and button (Views) omitted for brevity,
    private final Handler handler = new Handler() {
        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            output.setText(parseXMLResult(bundleResult));
        }
    };
    @Override
    public void onCreate(final Bundle icicle) {
        super.onCreate(icicle);
        this.setContentView(R.layout.delicious_posts);
        . . . inflate views omitted for brevity
        this.button.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                output.setText("");
                performRequest(user.getText().toString(),
                    pass.getText().toString());
            }
        });
    };
    . . . onPause omitted for brevity
    private void performRequest(String user, String pass) {
        this.progressDialog = ProgressDialog.show(this,
            "working . . .", "performing HTTP post to del.icio.us");
        final ResponseHandler<String> responseHandler =
            HTTPRequestHelper.getResponseHandlerInstance(this.handler);
        new Thread() {

```

1 Include
Delicious
URL

2 Provide Handler
to update UI

3 Pass credentials to
performRequest

```

        public void run() {
            HTTPRequestHelper helper =
                new HTTPRequestHelper(responseHandler);
            helper.performPost(URL_GET_POSTS_RECENT,
                user, pass, null, null);

        }
    }.start();
}
private String parseXMLResult(String xmlString) {
    StringBuilder result = new StringBuilder();
    try {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        SAXParser sp = spf.newSAXParser();
        XMLReader xr = sp.getXMLReader();
        DeliciousHandler handler = new DeliciousHandler();
        xr.setContentHandler(handler);
        xr.parse(new InputSource(new StringReader(xmlString)));
        List<DeliciousPost> posts = handler.getPosts();
        for (DeliciousPost p : posts) {
            result.append("\n" + p.getHref());
        }
    } catch (Exception e) {
        // log and or handle
    }
    return result.toString();
}

```

4 Use helper for HTTP

5 Parse XML String result

To use a POX service, we need to know a bit about it, beginning with the URL

endpoint 1. To call the Delicious service, we again use a Handler to update the UI 2, and we use the `HttpRequestHelper` we previously built and walked through in the last section. Again in this example, we have many fewer lines of code than if we didn't use the helper—lines of code we'd likely be repeating in different Activity classes. With the helper instantiated, we call the `performRequest()` method with a username and password 3. This method, via the helper, will log in to Delicious and return an XML chunk representing the most recently bookmarked items 4.

To turn the raw XML into useful types, we then also include

a `parseXMLResult()` method 5. Parsing XML is a subject in its own right, and we'll cover it in more detail in chapter 13, but the short takeaway with this method is that we walk the XML structure with a parser and return our own `DeliciousPost` data beans for each record. That's it—that's using POX to read data over HTTPS.

Building on the addition of XML to HTTP, above and beyond POX, is the REST architectural principle, which we'll explore next.

6.5.2. REST

While we look at REST, we'll also try to pull in another useful concept in terms of Android development: working with the various Google GData APIs (<http://code.google.com/apis/gdata/>). We used the GData APIs for our RestaurantFinder review information in [chapter 3](#), but there we didn't authenticate, and we didn't get into the details of networking or REST. In this section, we'll uncover the details as we perform two distinct tasks: authenticate and retrieve a Google `ClientLogin` token and retrieve the Google Contacts data for a specified user. Keep in mind that as we work with the GData APIs in any capacity, we'll be using a REST-style API.

The main REST concepts are that you specify resources in a URI form and you use different protocol methods to perform different actions. The *Atom Publishing Protocol* (AtomPub) defines a REST-style protocol, and the GData APIs are an implementation of AtomPub (with some Google extensions). As we noted earlier, the entire `Intent` approach of the Android platform is a lot like REST. A URI such as `content://contacts/1` is in the REST style. It includes a path that identifies the type of data and a particular resource (contact number 1).

That URI doesn't say what to do with contact 1, though. In REST terms, that's where the method of the protocol comes into the picture. For HTTP purposes, REST uses various methods to perform different tasks: `POST` (create, update, or in special cases, delete), `GET` (read), `PUT` (create, replace), and `DELETE` (delete). True HTTP REST implementations use all the HTTP method types and resources to construct APIs.

In the real world, you'll find few true REST implementations. It's much more common to see a REST-style API. This kind of API doesn't typically use the HTTP `DELETE` method (many servers, proxies, and so on, have trouble with `DELETE`) and overloads the more common `GET` and `POST` methods with different URLs for different tasks (by encoding a bit about what's to be done in the URL, or as a header or parameter, rather than relying strictly on the method). In fact, though many people refer to the GData APIs as REST, they're technically only REST-like, not true REST. That's not necessarily a bad thing; the idea is ease of use of the API rather than pattern purity. All in all, REST is a popular architecture or style because it's simple, yet powerful.

The following listing is an example that focuses on the network aspects of authentication with GData to obtain a `ClientLogin` token and use that token with a subsequent REST-style request to obtain Contacts data by including an email address as a resource.

Listing 6.11. Using the Google Contacts AtomPub API with authentication

```
public class GoogleClientLogin extends Activity {
    private static final String URL_GET_GTOKEN =
        "https://www.google.com/accounts/ClientLogin";
    private static final String URL_GET_CONTACTS_PREFIX =
        "http://www.google.com/m8/feeds/contacts/";
    private static final String URL_GET_CONTACTS_SUFFIX = "/full";
    private static final String GTOKEN_AUTH_HEADER_NAME = "Authorization";
    private static final String GTOKEN_AUTH_HEADER_VALUE_PREFIX =
        "GoogleLogin auth=";
    private static final String PARAM_ACCOUNT_TYPE = "accountType";
    private static final String PARAM_ACCOUNT_TYPE_VALUE =
        "HOSTED_OR_GOOGLE";
    private static final String PARAM_EMAIL = "Email";
    private static final String PARAM_PASSWD = "Passwd";
    private static final String PARAM_SERVICE = "service";
    private static final String PARAM_SERVICE_VALUE = "cp";
    private static final String PARAM_SOURCE = "source";
    private static final String PARAM_SOURCE_VALUE =
        "manning-unlockingAndroid-1.0";
    private String tokenValue;
    . . . View member declarations omitted for brevity
    private final Handler tokenHandler = new Handler() {

        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            String authToken = bundleResult;
            authToken = authToken.substring(authToken.indexOf("Auth=")
                + 5, authToken.length()).trim();
            tokenValue = authToken;
            GtokenText.setText(authToken);
        }
    };
    private final Handler contactsHandler =
        new Handler() {

        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            output.setText(bundleResult);
        }
    };
    . . . onCreate and onPause omitted for brevity
    private void getToken(String email, String pass) {
        final ResponseHandler<String> responseHandler =
            HTTPRequestHelper.getResponseHandlerInstance(
                this.tokenHandler);
    }
}
```

1 Create
Handler
token
request

2 Set
tokenValue

3 Implement
getToken

```

this.progressDialog = ProgressDialog.show(this,
    "working . . .", "getting Google ClientLogin token");
new Thread() {
    public void run() {
        HashMap<String, String> params =
            new HashMap<String, String>();
        params.put(GoogleClientLogin.PARAM_ACCOUNT_TYPE,
            GoogleClientLogin.PARAM_ACCOUNT_TYPE_VALUE);
        params.put(GoogleClientLogin.PARAM_EMAIL, email);
        params.put(GoogleClientLogin.PARAM_PASSWD, pass);
        params.put(GoogleClientLogin.PARAM_SERVICE,
            GoogleClientLogin.PARAM_SERVICE_VALUE);
        params.put(GoogleClientLogin.PARAM_SOURCE,
            GoogleClientLogin.PARAM_SOURCE_VALUE);
        HTTPRequestHelper helper =
            new HTTPRequestHelper(responseHandler);
        helper.performPost(HTTPRequestHelper.MIME_FORM_ENCODED,
            GoogleClientLogin.URL_GET_GTOKEN,
            null, null, null, params);
    }
}.start();
}

private void getContacts(final String email, final String token) {
    final ResponseHandler<String> responseHandler =
        HTTPRequestHelper.getResponseHandlerInstance(
            this.contactsHandler);
    this.progressDialog = ProgressDialog.show(this,
        "working . . .", "getting Google Contacts");
    new Thread() {
        public void run() {
            HashMap<String, String> headers =
                new HashMap<String, String>();
            headers.put(GoogleClientLogin.GTOKEN_AUTH_HEADER_NAME,
                GoogleClientLogin.GTOKEN_AUTH_HEADER_VALUE_PREFIX
                + token);
            String encEmail = email;
            try {
                encEmail = URLEncoder.encode(encEmail,
                    "UTF-8");
            } catch (UnsupportedEncodingException e) {
                // log and or handle
            }
            String url =
                GoogleClientLogin.URL_GET_CONTACTS_PREFIX + encEmail
                + GoogleClientLogin.URL_GET_CONTACTS_SUFFIX;
            HTTPRequestHelper helper = new
                HTTPRequestHelper(responseHandler);
            helper.performGet(url, null, null, headers);
        }
    }.start();
}
}

Required parameters for ClientLogin 4
Perform POST to get token 5
Implement getContacts 6
Add token as header 7
Encode email address in URL 8
Make GET request for Contacts 9

```

After a host of constants that represent various `String` values we'll use with the GData services, we have several `Handler` instances in this class, beginning with

1 a `tokenHandler`. This handler updates a UI `TextView` when it receives a message, like similar examples you saw previously, and updates a non-UI

2 member `tokenValue` variable that other portions of our code will use . The next `Handler` we have is the `contactsHandler` that will be used to update the UI after the contacts request.

Beyond the handlers, we have the `getToken()` method 3. This method includes all the required parameters for obtaining a `ClientLogin` token from the GData servers

(<http://code.google.com/apis/gdata/auth.html>) 4. After the setup to obtain the token, we make a `POST` request via the request helper 5.

After the token details are taken care of, we have the `getContacts()` method 6. This method uses the token obtained via the previous method as a header 7. After you have the token, you can cache it and use it with all subsequent requests; you don't need to obtain the token every time. Next, we encode the email address portion of the Contacts API URL 8, and we make a `GET` request for the data—again using the `HttpRequestHelper` 9.

With this approach, we're making several network calls (one as HTTPS to get the token and another as HTTP to get data) using our previously defined helper class. When the results are returned from the GData API, we parse the XML block and update the UI.

GData ClientLogin and CAPTCHA

Though we included a working `ClientLogin` example in [listing 6.11](#), we also skipped over an important part—CAPTCHA. Google might optionally require a CAPTCHA with the `ClientLogin` approach. To fully support `ClientLogin`, you need to handle that response and display the CAPTCHA to the user, and then resend a token request with the CAPTCHA value that the user entered. For details, see the GData documentation.

Now that we've explored some REST-style networking, the last thing we need to discuss with regard to HTTP and Android is SOAP. This topic comes up frequently in discussions of networking mobile devices, but sometimes the forest gets in the way of the trees in terms of framing the real question.

6.5.3. To SOAP or not to SOAP, that is the question

SOAP is a powerful protocol that has many uses. We would be remiss if we didn't at least mention that though it's possible to use SOAP on a small, embedded device such as a smartphone, regardless of the platform, it's not recommended. The question within the limited resources environment Android inhabits is really more one of *should* it be done rather than *can* it be done.

Some experienced developers, who might have been using SOAP for years on other devices, might disagree. The things that make SOAP great are its support for strong types (via XML Schema), its support for transactions, its security and encryption, its support for message orchestration and choreography, and all the related WS-* standards. These things are invaluable in many server-oriented computing environments, whether or not they involve the enterprise. They also add a great deal of overhead, especially on a small, embedded device. In fact, in many situations where people use SOAP on embedded devices, they often don't bother with the advanced features—and they use plain XML with the overhead of an envelope at the end of the day anyway. On an embedded device, you often get better performance, and a simpler design, by using a REST- or POX-style architecture and avoiding the overhead of SOAP.

Even with the increased overhead, it makes sense in some situations to investigate using SOAP directly with Android. When you need to talk to existing SOAP services that you have no control over, SOAP might make sense. Also, if you already have J2ME clients for existing SOAP services, you might be able to port those in a limited set of cases. Both these approaches make it easier only on you, the developer; they have either no effect or a negative one in terms of performance on the user. Even when you're working with existing SOAP services, remember that you can often write a POX- or REST-style proxy for SOAP services on the server side and call that from Android, rather than use SOAP directly from Android.

If you feel like SOAP is still the right choice, you can use one of several ports of the kSOAP toolkit (<http://ksoap2.sourceforge.net/>), which is specially designed for SOAP on an embedded Java device. Keep in mind that even the kSOAP documentation states, "SOAP introduces some significant overhead for web services that may be problematic for mobile devices. If you have full control over the client and the server, a REST-based architecture may be more adequate." In addition, you might be able to write your own parser for simple SOAP services that don't use fancy SOAP features and just use a POX approach that includes the SOAP XML portions you require (you can always roll your own, even with SOAP).

All in all, to our minds the answer to the question is to not use SOAP on Android, even though you can. Our discussion of SOAP, even though we don't advocate it, rounds out our more general web services discussion, and that wraps up our networking coverage.

6.6. SUMMARY

In this chapter, we started with a brief background of basic networking concepts, from nodes and addresses to layers and protocols. With that general background in place, we covered details about how to obtain network status information and showed several different ways to work with the IP networking capabilities of the platform.

In terms of networking, we looked at using basic sockets and the `java.net` package. Then we also examined the included Apache HttpClient API. HTTP is one of the most common—and most important—networking resources available to the Android platform. Using `HttpClient`, we covered a lot of territory in terms of different request types, parameters, headers, authentication, and more. Beyond basic HTTP, we also explored POX and REST, and we discussed a bit of SOAP—all of which use HTTP as the transport mechanism.

Now that we've covered a good deal of the networking possibilities, and hopefully given you at least a glint of an idea of what you can do with server-side APIs and integration with Android, we're going to turn to another important part of the Android world—telephony.

Chapter 7. Telephony

This chapter covers

- Making and receiving phone calls
- Capturing call-related events
- Obtaining phone and service information
- Using SMS

People use Android devices to surf the web, download and store data, access networks, find location information, and use many types of applications. Android can even make phone calls.

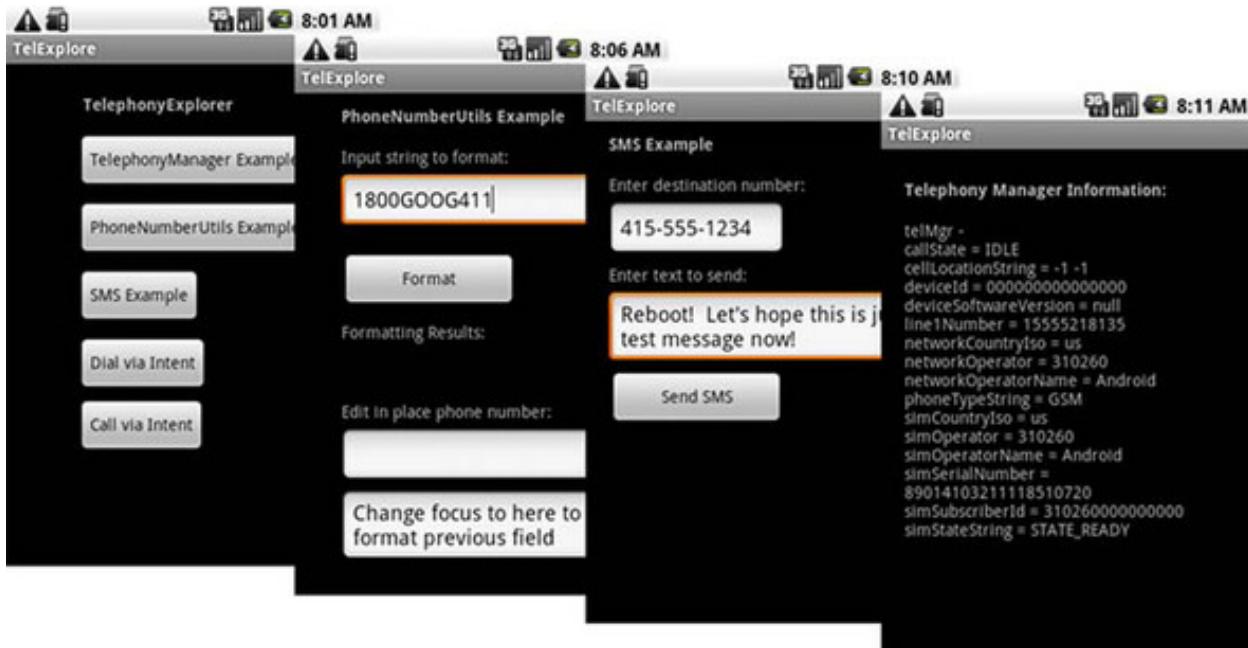
Android phones support dialing numbers, receiving calls, sending and receiving text and multimedia messages, and other related telephony services. In contrast to other smartphone platforms, all these items are accessible to developers through simple-to-use APIs and built-in applications. You can easily leverage Android's telephony support into your own applications.

In this chapter, we'll discuss telephony in general and cover terms related to mobile devices. We'll move on to basic Android telephony packages, which handle calls using built-in `Intent` actions, and more advanced operations via the `TelephonyManager` and `PhoneStateListener` classes. The `Intent` actions can initiate basic phone calls in your applications. `TelephonyManager` doesn't make phone calls directly but is used to retrieve all kinds of telephony-related data, such as the state of the voice network, the device's own phone number, and other details. `TelephonyManager` supports adding a `PhoneStateListener`, which can alert you when call or phone network states change.

After covering basic telephony APIs, we'll move on to sending and receiving SMS messages. Android provides APIs that allow you to send SMS messages and be notified when SMS messages are received. We'll also touch on emulator features that allow you to test your app by simulating incoming phone calls or messages.

Once again, a sample application will carry us through the concepts related to the material in this chapter. You'll build a sample `TelephonyExplorer` application to demonstrate dialing the phone, obtaining phone and service state information, adding listeners to the phone state, and working with SMS. Your `TelephonyExplorer` application will have several basic screens, as shown in [figure 7.1](#).

Figure 7.1. TelephonyExplorer main screen, along with the related activities the sample application performs



TelephonyExplorer exercises the telephony-related APIs while remaining simple and uncluttered. Before we start to build TelephonyExplorer, let's first define telephony itself.

7.1. EXPLORING TELEPHONY BACKGROUND AND TERMS

Whether you're a new or an experienced mobile developer, it's important to clarify terms and set out some background for discussing telephony.

First, *telephony* is a general term that refers to electrical voice communications over telephone networks. Our scope is, of course, the mobile telephone networks that Android devices¹ participate in, specifically the Global System for Mobile Communications (GSM) and Code Division Multiple Access (CDMA) networks.

¹ For a breakdown of all Android devices by year of release, go here: www.androphones.com/all-androidphones.php.

GSM and CDMA are cellular telephone networks. Devices communicate over radio waves and specified frequencies using cell towers. The standards must define a few important things, such as identities for devices and cells, along with all the rules for making communications possible.

7.1.1. Understanding GSM

We won't delve into the underlying details of the networks, but it's important to know some key facts. GSM is based on Time Division Multiple Access (TDMA), a technology that slices time into smaller chunks to allow multiple callers to share the same frequency range. GSM was the first network that the Android stack supported for voice calls; it's ubiquitous in Europe and very common in North America. GSM devices use Subscriber Identity Module (SIM) cards to store important network and user settings.

A SIM card is a small, removable, secure smart card. Every device that operates on a GSM network has specific unique identifiers, which are stored on the SIM card or on the device itself:

- **Integrated Circuit Card Identifier (ICCID)**— Identifies a SIM card; also known as a SIM Serial Number, or SSN.
- **International Mobile Equipment Identity (IMEI)**— Identifies a physical device. The IMEI number is usually printed underneath the battery.
- **International Mobile Subscriber Identity (IMSI)**— Identifies a subscriber (and the network that subscriber is on).
- **Location Area Identity (LAI)**— Identifies the region within a provider network that's occupied by the device.
- **Authentication key (Ki)**— A 128-bit key used to authenticate a SIM card on a provider network.

GSM uses these identification numbers and keys to validate and authenticate a SIM card, the device holding it, and the subscriber on the network and across networks.

Along with storing unique identifiers and authentication keys, SIM cards often store user contacts and SMS messages. Users can easily move their SIM card to a new device and carry along contact and message data. Currently, the Android platform handles the SIM interaction, and developers can get read-only access via the telephony APIs.

7.1.2. Understanding CDMA

The primary rival to GSM technology is CDMA, which uses a different underlying technology that's based on using different encodings to allow multiple callers to share the same frequency range. CDMA is widespread in the United States and common in some Asian countries.

Unlike GSM phones, CDMA devices don't have a SIM card or other removable module. Instead, certain identifiers are burned into the device, and the carrier must maintain the link between each device and its subscriber. CDMA devices have a separate set of unique identifiers:

- **Mobile Equipment Identifier (MEID)**— Identifies a physical device. This number is usually printed under the battery and is available from within device menus. It corresponds to GSM's IMEI.
- **Electronic Serial Number (ESN)**— The predecessor to the MEID, this number is shorter and identifies a physical device.
- **Pseudo Electronic Serial Number (pESN)**— A hardware identifier, derived from the MEID, that's compatible with the older ESN standard. The ESN supply was exhausted several years ago, so pESNs provide a bridge for legacy applications built around ESN. A pESN always starts with 0x80 in hex format or 128 in decimal format.

Unlike GSM phones, which allow users to switch devices by swapping out SIM cards, CDMA phones require you to contact your carrier if you want to transfer an account to a new device. This process is often called an *ESN swap* or *ESN change*. Some carriers make this easy, and others make it difficult. If you'll be working on CDMA devices, learning how to do this with your carrier can save you thousands of dollars in subscriber fees.

Note

A few devices, sometimes called *world phones*, support both CDMA and GSM. These devices often have two separate radios and an optional SIM card. Currently, such devices operate only on one network or the other at any given time. Additionally, these devices are often restricted to using only particular carriers or technologies in particular countries. You generally don't need to do anything special to support these devices, but be aware that certain phones might appear to change their network technology from time to time.

Fortunately, few applications need to deal with the arcana of GSM and CDMA technology. In most cases, you only need to know that your program is running on a device that in turn is running on a mobile network. You can leverage that network to make calls and inspect the device to find unique identifiers. You can locate this sort of information by using the `TelephonyManager` class.

7.2. PHONE OR NOT?

Starting with version 2.1 of the Android OS, devices no longer need to support telephony features. Expect more and more non-phone devices to reach the market, such as set-top boxes, auto devices, and certain tablets. If you want to reach the largest possible market

with your app, you should include telephony features but fail gracefully if they're not available. If your application makes sense only when running on a phone, go ahead and use any phone features you require.

If your application requires telephony to function, you should add the following declaration to your `AndroidManifest.xml`:

```
<uses-feature android:name="android.hardware.telephony"  
    android:required="true"/>
```

This will let Android Market and other storefronts know not to offer your app to non-phone devices; otherwise, expect many complaints and queries from disappointed customers. If your application supports telephony but can operate without it, set `android:required` to "false".

7.3. ACCESSING TELEPHONY INFORMATION

Android provides an informative manager class that supplies information about many telephony-related details on the device. Using `TelephonyManager`, you can access phone properties and obtain phone network state information.

Note

Starting with version 2.1 of the Android OS, devices no longer need to support telephony features. Expect more and more non-phone devices to reach the market, such as set-top boxes and auto devices. If you want to reach the largest possible market with your app, you should leverage telephony features but fail gracefully if they're not available. If your application makes sense only when running on a phone, go ahead and use any phone features you require.

You can attach a `PhoneStateListener` event listener to the phone by using the manager. Attaching a `PhoneStateListener` makes your applications aware of when the phone gains and loses service, and when calls start, continue, or end.

Next, we'll examine several parts of the `TelephonyExplorer` example application to look at both these classes. We'll start by obtaining a `TelephonyManager` instance and using it to query useful telephony information.

7.3.1. Retrieving telephony properties

The `android.telephony` package contains the `TelephonyManager` class, which provides details about the phone status. Let's retrieve and display a small subset of that information to demonstrate the approach. First, you'll build an `Activity` that displays a simple screen showing some of the information you can obtain via `TelephonyManager`, as shown in [figure 7.2](#).

Figure 7.2. Displaying device and phone network meta-information obtained from `TelephonyManager`



The `TelephonyManager` class is the information hub for telephony-related data in Android. The following listing demonstrates how you obtain a reference to this class and use it to retrieve data.

[Listing 7.1. Obtaining a `TelephonyManager` reference and using it to retrieve data](#)

```
// . . . start of class omitted for brevity
final TelephonyManager telMgr =
    (TelephonyManager) getSystemService(
        Context.TELEPHONY_SERVICE);
// . . . onCreate method and others omitted for brevity
public String getTelephonyOverview(
    TelephonyManager telMgr) {
    String callStateString = "NA";
    int callState = telMgr.getCallState();
    switch (callState) {
        case TelephonyManager.CALL_STATE_IDLE:
            callStateString = "IDLE";
            break;
        case TelephonyManager.CALL_STATE_OFFHOOK:
            callStateString = "OFFHOOK";
            break;
        case TelephonyManager.CALL_STATE_RINGING:
            callStateString = "RINGING";
            break;
    }

    CellLocation cellLocation = (CellLocation)telMgr.getCellLocation();
    String cellLocationString = null;
    if (cellLocation instanceof GsmCellLocation)
    {
        cellLocationString = ((GsmCellLocation)cellLocation).getLac()
            + " " + ((GsmCellLocation)cellLocation).getCid();
    }
    else if (cellLocation instanceof CdmaCellLocation)
    {
        cellLocationString = ((CdmaCellLocation)cellLocation).
            getBaseStationLatitude() + " " +
            ((CdmaCellLocation)cellLocation).getBaseStationLongitude();
    }
}
```

1 Get TelephonyManager from Context

2 Implement information helper method

3 Obtain call state information

 Get device information

```
String deviceId = telMgr.getDeviceId();
String deviceSoftwareVersion =
    telMgr.getDeviceSoftwareVersion();
String line1Number = telMgr.getLine1Number();
String networkCountryIso = telMgr.getNetworkCountryIso();
String networkOperator = telMgr.getNetworkOperator();
String networkOperatorName = telMgr.getNetworkOperatorName();

String phoneTypeString = "NA";
int phoneType = telMgr.getPhoneType();
switch (phoneType) {
    case TelephonyManager.PHONE_TYPE_GSM:
        phoneTypeString = "GSM";
        break;
    case TelephonyManager.PHONE_TYPE_CDMA:
        phoneTypeString = "CDMA";
        break;
    case TelephonyManager.PHONE_TYPE_NONE:
        phoneTypeString = "NONE";
        break;
}

String simCountryIso = telMgr.getSimCountryIso();
String simOperator = telMgr.getSimOperator();
String simOperatorName = telMgr.getSimOperatorName();
String simSerialNumber = telMgr.getSimSerialNumber();
String simSubscriberId = telMgr.getSubscriberId();
String simStateString = "NA";
int simState = telMgr.getSimState();
switch (simState) {
    case TelephonyManager.SIM_STATE_ABSENT:
        simStateString = "ABSENT";
        break;
    case TelephonyManager.SIM_STATE_NETWORK_LOCKED:
        simStateString = "NETWORK_LOCKED";
        break;
    // . . . other SIM states omitted for brevity
}

StringBuilder sb = new StringBuilder();
sb.append("telMgr - ");
sb.append("\ncallState = " + callStateString);
// . . . remainder of appends omitted for brevity
return sb.toString();
}
```

We use the current Context, through the `getSystemService` method with a constant, to obtain an instance of the `TelephonyManager` class . After you have the manager, you

can use it as needed. In this case, we create a helper method to get data from the manager and return it as a `String` that we later display on the screen .

The manager allows you to access phone state data, such as whether a call is in progress , the device ID and software version , the phone number registered to the current user/SIM, and other SIM details, such as the subscriber ID (IMSI) and the current SIM state. `TelephonyManager` offers even more properties; see the Javadocs for complete details.

Note

Methods generally return `null` if they don't apply to a particular device; for example, `getSimOperatorName()` returns `null` for CDMA phones. If you want to know in advance what type of device you're working with, try using the method `getPhoneType()`.

For this class to work, you must set the `READ_PHONE_STATE` permission in the manifest. Without it, security exceptions will be thrown when you try to read data from the manager. Phone-related permissions are consolidated in [table 7.1](#).

Table 7.1. Phone-related manifest permissions and their purpose

Phone-related permission	Purpose
<code>android.permission.CALL_PHONE</code>	Initiates a phone call without user confirmation in dialer
<code>android.permission.CALL_PRIVILEGED</code>	Calls any number, including emergency, without confirmation in dialer
<code>android.permission.MODIFY_PHONE_STATE</code>	Allows the application to modify the phone state: for example, to turn the radio on or off
<code>android.permission.PROCESS_OUTGOING_CALLS</code>	Allows the application to receive broadcast for outgoing calls and modify
<code>android.permission.READ_PHONE_STATE</code>	Allows the application to read the phone state

In addition to providing telephony-related information, including metadata about the device, network, and subscriber, `TelephonyManager` allows you to attach a `PhoneStateListener`, which we'll describe in the next section.

7.3.2. Obtaining phone state information

A phone can be in any one of several conditions. The primary phone states include *idle* (waiting), in a call, or initiating a call. When you're building applications on a mobile device, sometimes you not only need to know the current phone state, but you also want to know when the state changes.

In these cases, you can attach a listener to the phone and subscribe to receive notifications of published changes. With Android, you use a `PhoneStateListener`, which attaches to the phone through `TelephonyManager`. The following listing demonstrates a sample usage of both these classes.

Listing 7.2. Attaching a `PhoneStateListener` via the `TelephonyManager`

```
@Override  
  
public void onStart() {  
  
    super.onStart();  
  
    final TelephonyManager telMgr =  
        (TelephonyManager) getSystemService(  
            Context.TELEPHONY_SERVICE);  
  
    PhoneStateListener phoneStateListener =  
        new PhoneStateListener() {  
  
            public void onCallStateChanged(  
                int state, String incomingNumber) {  
  
                telMgrOutput.setText(getTelephonyOverview(telMgr));  
            }  
        };  
  
    telMgr.listen(phoneStateListener,  
        PhoneStateListener.LISTEN_CALL_STATE);  
  
    String telephonyOverview = getTelephonyOverview(telMgr);
```

```
    telMgrOutput.setText(telephonyOverview);  
}
```

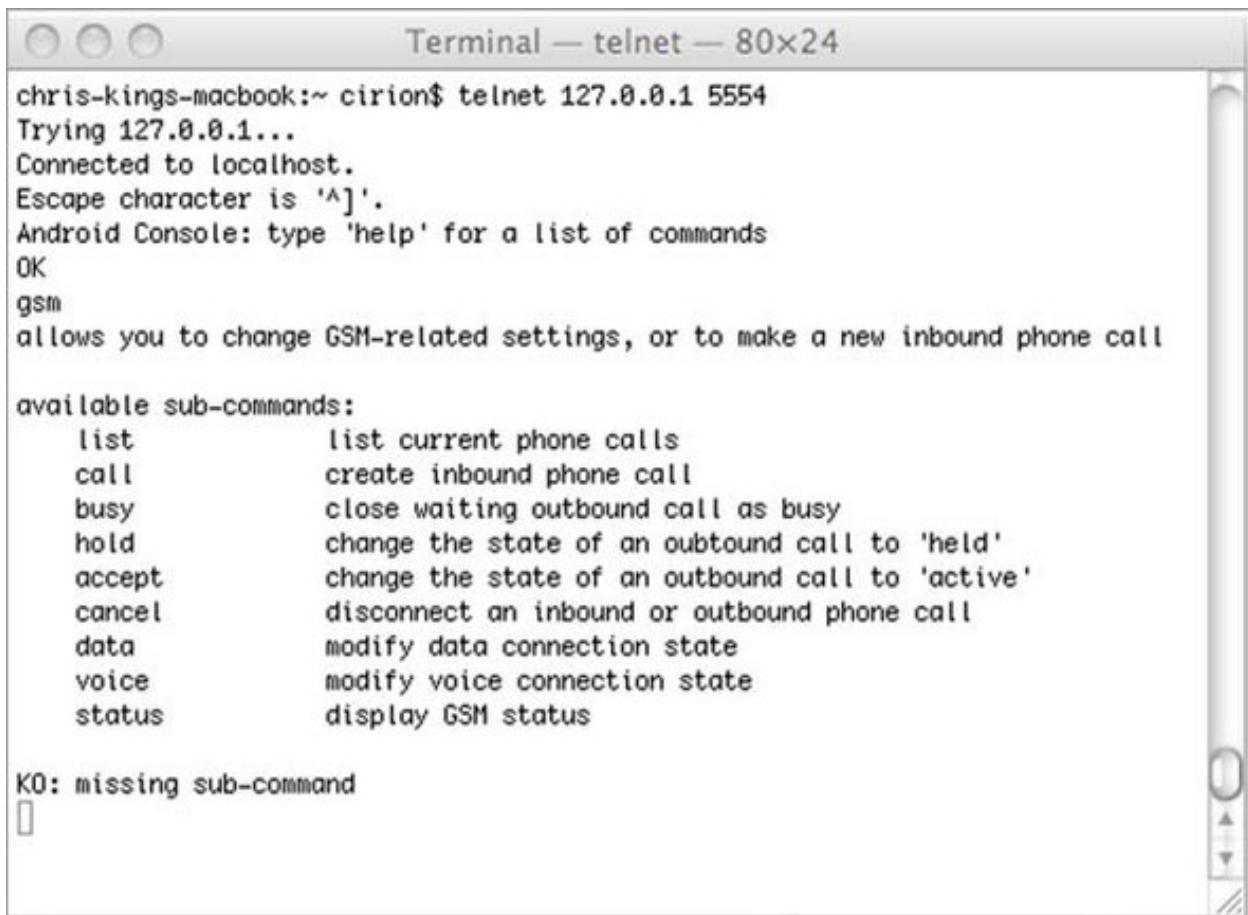
To start working with a `PhoneStateListener`, you need to acquire an instance of `TelephonyManager`. `PhoneStateListener` itself is an interface, so you need to create an implementation, including the required `onCallStateChanged()` method. When you have a valid `PhoneStateListener` instance, you attach it by assigning it to the manager with the `listen()` method.

[Listing 7.2](#) shows how to listen for any `PhoneStateListener.LISTEN_CALL_STATEchange` in the phone state. This constant value comes from a list of available states that are in `PhoneStateListener` class. You can use a single value when assigning a listener with the `listen()` method, as demonstrated in [listing 7.2](#), or you can combine multiple values to listen for multiple states.

If a call state change does occur, it triggers the action defined in the `onCallStateChanged()` method of your `PhoneStateListener`. In this example, we reset the details on the screen using the `getTelephonyOverview()` method from [listing 7.1](#). You can filter this method further, based on the passed-in `int state`.

To see the values in this example change while you're working with the emulator, you can use the SDK tools to send incoming calls or text messages and change the state of the voice connection. You can access these options from the DDMS perspective in Eclipse. Additionally, the emulator includes a mock GSM modem that you can manipulate using the `gsm` command from the console. [Figure 7.3](#) shows an example session from the console that demonstrates using the `gsm` command. For complete details, see the emulator telephony documentation at <http://code.google.com/android/reference/emulator.html#telephony>.

Figure 7.3. An Android console session demonstrating the `gsm` command and available subcommands



A screenshot of a terminal window titled "Terminal — telnet — 80x24". The window shows a telnet session to localhost port 5554. The session starts with basic connection information, followed by an "Android Console" prompt which suggests using "help" for commands. It then lists "OK", "gsm", and "available sub-commands". A detailed list of sub-commands is provided, each with a brief description:

list	list current phone calls
call	create inbound phone call
busy	close waiting outbound call as busy
hold	change the state of an outbound call to 'held'
accept	change the state of an outbound call to 'active'
cancel	disconnect an inbound or outbound phone call
data	modify data connection state
voice	modify voice connection state
status	display GSM status

The session ends with an error message "K0: missing sub-command".

Now that we've covered the major elements of telephony, let's start exploring basic uses of the telephony APIs and other related facilities. We'll intercept calls, leverage telephony utility classes, and make calls from within applications.

7.4. INTERACTING WITH THE PHONE

In regular development, you'll often want to use your Android device as a phone. You might dial outbound calls through simple built-in intents, or intercept calls to modify them in some way. In this section, we'll cover these basic tasks and examine some of the phone-number utilities Android provides for you.

One of the more common things you'll do with Android telephony support doesn't even require using the telephony APIs directly: making calls using built-in `Intents`.

7.4.1. Using Intents to make calls

As we demonstrated in [chapter 4](#), to invoke the built-in dialer and make a call all you need to use is the `Intent.ACTION_CALL` action and the `tel: Uri`. This approach invokes the dialer application, populates the dialer with the provided telephone number (taken from the URI), and initiates the call.

Alternatively, you can invoke the dialer application with the `Intent.ACTION_DIAL` action, which also populates the dialer with the supplied phone number but stops short of initiating the call. The following listing demonstrates both techniques using their respective actions.

Listing 7.3. Using Intent actions to dial and call using the built-in dialer application

```
dialintent = (Button) findViewById(R.id.dialintent_button);

dialintent.setOnClickListener(new OnClickListener() {

    public void onClick(View v) {

        Intent intent =
            new Intent(Intent.ACTION_DIAL,
            Uri.parse("tel:" + NUMBER));
        startActivity(intent);
    }
});

callintent = (Button) findViewById(R.id.callintent_button);

callintent.setOnClickListener(new OnClickListener() {

    public void onClick(View v) {

        Intent intent =
            new Intent(Intent.ACTION_CALL,
            Uri.parse("tel:" + NUMBER));
        startActivity(intent);
    }
});
```

By now you should feel quite comfortable using `Intents` in the Android platform. In this listing, we again take advantage of Android's loose coupling, in this case to make outgoing calls to specified numbers. First, you set the action you want to take place, either populating the dialer with `ACTION_DIAL` or populating the dialer *and* initiating a

call with `ACTION_CALL`. In either case, you also need to specify the telephone number you want to use with the `Intent` URI.

Dialing calls also requires the proper permissions, which your application manifest includes in order to access and modify the phone state, dial the phone, or intercept phone calls (shown in section 7.3.3). [Table 7.1](#) lists the relevant phone-related permissions and their purposes. For more detailed information, see the security section of the Android documentation at <http://code.google.com/android-devel/security.html>.

Android makes dialing simple with built-in handling via `Intents` and the dialer application. The `PhoneNumberUtils` class, which you can use to parse and validate phone number strings, helps simplify dialing even more, while keeping numbers human-readable.

7.4.2. Using phone number–related utilities

Applications running on mobile devices that support telephony deal with a lot of `String` formatting for phone numbers. Fortunately, the Android SDK provides a handy utility class that helps to mitigate the risks associated with this task and standardize the numbers you use—`PhoneNumberUtils`.

The `PhoneNumberUtils` class parses `String` data into phone numbers, transforms alphabetical keypad digits into numbers, and determines other properties of phone numbers. The following listing shows an example of using this class.

Listing 7.4. Working with the `PhoneNumberUtils` class

```

// Imports omitted for brevity
private TextView pnOutput;
private EditText pnInput;
private EditText pnInPlaceInput;
private Button pnFormat;
// Other instance variables and methods omitted for brevity
pnFormat.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        String phoneNumber = PhoneNumberUtils.formatNumber(
            pnInput.getText().toString());

        phoneNumber = PhoneNumberUtils.convertKeypadLettersToDigits(
            pnInput.getText().toString());

        StringBuilder result = new StringBuilder();
        result.append(phoneNumber);
        result.append("\nisGlobal - "
            + PhoneNumberUtils.isGlobalPhoneNumber(phoneNumber));
        result.append("\nisEmergency - "
            + PhoneNumberUtils.isEmergencyNumber(phoneNumber));
        result.append("\ncompare to 415-555-1234 - " +
            PhoneNumberUtils.compare(phoneNumber, "415-555-1234"));
        pnOutput.setText(result.toString());
        pnInput.setText("");
    }
});

```

1 Format as phone number

2 Convert alpha characters to digits

3 Compare to another number

The `PhoneNumberUtils` class offers several static helper methods for parsing phone numbers, including the useful `formatNumber`. This method takes a single `String` as input

and uses the default locale settings to return a formatted phone number 1. Additional methods format a number using a locale you specify, parse different segments of a number, and so on. Parsing a number can be combined with another helpful method, `convertKeypadLettersToDigits()`, to convert any alphabetic keypad letter

characters into digits 2. The conversion method won't work unless it already recognizes the format of a phone number, so you should run the format method first.

Along with these basic methods, you can also check properties of a number string, such as whether the number is global and whether it represents an emergency call.

The `compare()` method lets you see whether a given number matches another

number 3, which is useful for user-entered numbers that might include dashes or dots.

Note

Android defines a *global* number as any string that contains one or more digits; it can optionally be prefixed with a + symbol, and can optionally contain dots or dashes. Even strings like 3 and +4-2 are considered global numbers. Android makes no guarantee that a phone can even dial such a number; this utility simply provides a basic check for whether something that looks like it could be a phone number in some country.



You can also format a phone number with the overloaded `formatNumber()` method. This method is useful for any `Editable`, such as the common `EditText` (or `TextView`). This method updates the provided `Editable` in-place, as shown in the following listing.

Listing 7.5. Using in-place `Editable` View formatting via `PhoneNumberUtils`

```
pnInPlaceInput.setOnFocusChangeListener(  
  
    new OnFocusChangeListener() {  
  
        public void onFocusChange(View v, boolean hasFocus) {  
  
            if (v.equals(pnInPlaceInput) && (!hasFocus)) {  
  
                PhoneNumberUtils.formatNumber(  
                    pnInPlaceInput.getText(),  
  
                    PhoneNumberUtils.FORMAT_NANP);  
  
            }  
        }  
    });
```

The in-place editor can be combined with a dynamic update using various techniques. You can make the update happen automatically when the focus changes from a phone-number field. The in-place edit does not provide the keypad alphabetic character-to-number conversion automatically. To ensure that the conversion occurs, we've implemented an `OnFocusChangeListener`. Inside the `onFocusChange()` method, which filters for the correct `View` item, we call the `formatNumber()` overload, passing in the respective `Editable` and the formatting style we want to use. NANP stands for North American Numbering Plan, which includes an optional country and area code and a 7-digit local phone number.



Note

`PhoneNumberUtils` also defines a Japanese formatting plan and might add others in the future.

Now that you can use the phone number utilities and make calls, we can move on to the more challenging and interesting task of call interception.

7.4.3. Intercepting outbound calls

Imagine writing an application that catches outgoing calls and decorates or aborts them, based on certain criteria. The following listing shows how to perform this type of interception.

Listing 7.6. Catching and aborting an outgoing call

```
public class OutgoingCallReceiver extends BroadcastReceiver {
    public static final String ABORT_PHONE_NUMBER = "1231231234";
    @Override
    public void onReceive(Context context, Intent intent) {           ← ① Override
        if (intent.getAction().equals(                                onReceive
            Intent.ACTION_NEW_OUTGOING_CALL)) {                      ← ② Filter Intent for action
            String phoneNumber =
                intent.getExtras().getString(Intent.EXTRA_PHONE_NUMBER);
            if ((phoneNumber != null)
                && phoneNumber.equals(
                    OutgoingCallReceiver.ABORT_PHONE_NUMBER)) {
                Toast.makeText(context,
                    "NEW_OUTGOING_CALL intercepted to number "
                    + "123-123-1234 - aborting call",
                    Toast.LENGTH_LONG).show();
                abortBroadcast();
            }
        }
    }
}
```

Our interception class starts by extending `BroadcastReceiver`. The new subclass implements the `onReceive()` method ①. Within this method, we filter on the `Intent` action we want ②, and then we get the `Intent` data using the phone number key. If the phone number matches, we send a `Toast` alert to the UI and abort the outgoing call by calling the `abortBroadcast()` method.

Beyond dialing out, formatting numbers, and intercepting calls, Android also provides support for sending and receiving SMS. Managing SMS can seem daunting but provides significant rewards, so we're going to focus on it for the rest of the chapter.

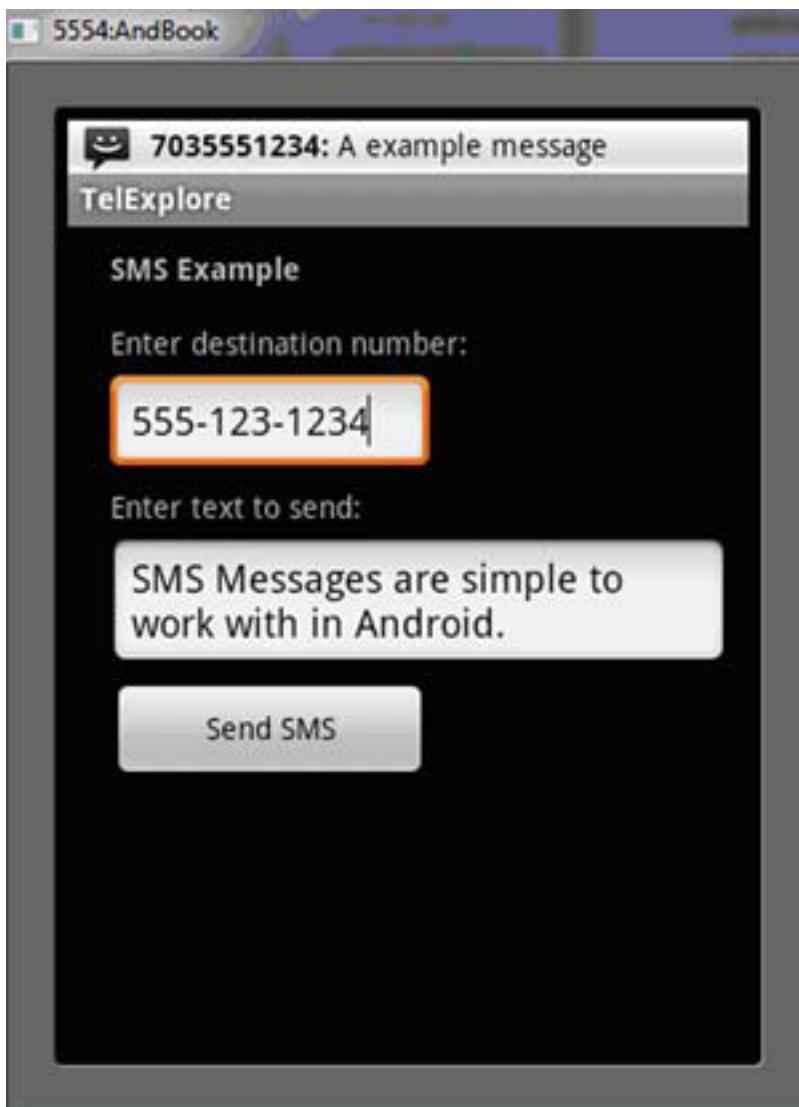
7.5. WORKING WITH MESSAGING: SMS

Mobile devices use the Short Message Service (SMS), a hugely popular and important means of communication, to send simple text messages with small amounts of data. Android includes a built-in SMS application that allows users to send, view, and reply to SMS messages. Along with the built-in user-facing apps and the related `ContentProvider` for interacting with the default text-messaging app, the SDK provides APIs for developers to send and receive messages programmatically.

Because Android now supplies an excellent built-in SMS message application, you might wonder why anyone would bother building another one. The Android market sells several superior third-party SMS messaging applications, but SMS can do a lot more than text your contacts. For example, you could build an application that, upon receiving a special SMS, sends back another SMS containing its location information. Due to the nature of SMS, this strategy might succeed, while another approach like trying to get the phone to transmit its location in real time would fail. Alternately, adding SMS as another communications channel can enhance other applications. Best of all, Android makes working with SMS relatively simple and straightforward.

To explore Android's SMS support, you'll create an app that sends and receives SMS messages. The screen in [figure 7.4](#) shows the SMS-related `Activity` you'll build in the `TelephonyExplorer` application.

Figure 7.4. An `Activity` that sends SMS messages



To get started working with SMS, you'll first build a class that programmatically sends SMS messages, using the `SmsManager`.

7.5.1. Sending SMS messages

The `android.telephony` package contains the `SmsManager` and `SmsMessage` classes. The `SmsManager` defines many important SMS-related constants, and also provides the `sendDataMessage`, `sendMultipartTextMessage`, and `sendTextMessage` methods.

Note

Early versions of Android provided access to SMS only through the `android.telephony.gsm` subpackage. Google has deprecated this usage, but if you must target older versions of the OS, look there for SMS-related functions. Of course, such classes work only on GSM-compatible devices.

The following listing shows an example from our `TelephonyExplorer` application that uses the SMS manager to send a simple text message.

Listing 7.7. Using `SmsManager` to send SMS messages

```
// . . . start of class omitted for brevity
private Button smsSend;
private SmsManager smsManager;
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.smsexample);
    // . . . other onCreate view item inflation omitted for brevity
    smsSend = (Button) findViewById(R.id.smssend_button);
    smsManager = SmsManager.getDefault();
    final PendingIntent sentIntent =
        PendingIntent.getActivity(
            this, 0, new Intent(this,
                SmsSendCheck.class), 0);
```

① Get `SmsManager` handle

```
smsSend.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        String dest = smsInputDest.getText().toString();
        if (PhoneNumberUtils.
            isWellFormedSmsAddress(dest)) {
            smsManager.sendTextMessage(
                smsInputDest.getText().toString, null,
                smsInputText.getText().toString(),
                sentIntent, null);
            Toast.makeText(SmsExample.this,
                "SMS message sent",
                Toast.LENGTH_LONG).show();
        } else {
            Toast.makeText(SmsExample.this,
                "SMS destination invalid - try again",
                Toast.LENGTH_LONG).show();
        }
    }
});
```

② Create `PendingIntent` for post action

```
});
```

③ Check that destination is valid

Before doing anything with SMS messages, we must **1** obtain an instance of the `SmsManager` with the static `getDefault()` method. The manager will also send the message later. Before we can send the message, we need to create a `PendingIntent` to provide to the send method.

A `PendingIntent` can specify an `Activity`, a `Broadcast`, or a `Service` that it requires. In our case, we use the `getActivity()` method, which requests an `Activity`, and then we specify the context, a request code (not used for this case), the `Intent` to execute, and **2** additional flags. The flags indicate whether the system should create a new instance of the referenced `Activity` (or `Broadcast` or `Service`), if one doesn't already exist.

Next, we check that the destination address is valid for SMS **3**, and we send the message using the manager's `sendTextMessage()` method.

This `send` method takes several parameters. The following snippet shows the signature of this method:

```
sendDataMessage(String destinationAddress, String scAddress,  
    short destinationPort, byte[] data, PendingIntent sentIntent,  
    PendingIntent deliveryIntent)
```

What is a `PendingIntent`?

A `PendingIntent` specifies an action to take in the future. It lets you pass a future `Intent` to another application and allow that application to execute that `Intent` as if it had the same permissions as your application, whether or not your application is still around when the `Intent` is eventually invoked. A `PendingIntent` provides a means for applications to work, even after their process exits. It's important to note that even after the application that created the `PendingIntent` has been killed, that `Intent` can still run.

The method requires the following parameters:

- `destinationAddress`— The phone number to receive the message.
- `scAddress`— The messaging center address on the network. You should almost always leave this as `null`, which uses the default.
- `destinationPort`— The port number for the recipient handset.

- ***data***— The payload of the message.
- ***sentIntent***— The `PendingIntent` instance that's fired when the message is successfully sent.
- ***deliveryIntent***— The `PendingIntent` instance that's fired when the message is successfully received.

Note

GSM phones generally support receiving SMS messages to a particular port, but CDMA phones generally don't. Historically, port-directed SMS messages have allowed text messages to be delivered to a particular application. Modern phones support better solutions; in particular, if you can use a server for your application, consider using Android Cloud to Device Messaging (C2DM)² for Android phones with software version 2.2 or later.

² Read Wei Huang's detailed article for more about C2DM: <http://android-developers.blogspot.com/2010/05/android-cloud-to-device-messaging.html>.

Much like the phone permissions listed in [table 7.1](#), SMS-related tasks also require manifest permissions. SMS permissions are shown in [table 7.2](#).

Table 7.2. SMS-related manifest permissions and their purpose

Phone-related permission	Purpose
<code>android.permission.READ_SMS</code>	Allows the application to read SMS messages
<code>android.permission.RECEIVE_SMS</code>	Allows the application to monitor incoming SMS messages
<code>android.permission.SEND_SMS</code>	Allows the application to send SMS messages
<code>android.permission.WRITE_SMS</code>	Writes SMS messages to the built-in SMS provider (not related to sending messages directly)

The `AndroidManifest.xml` file for the `TelephonyExplorer` application contains these permissions:

```
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.WRITE_SMS" />
<uses-permission android:name="android.permission.SEND_SMS" />
```

Along with sending text and data messages via `SmsManager`, you can create an `SMS BroadcastReceiver` to receive incoming SMS messages.

7.5.2. Receiving SMS messages

You can receive an SMS message programmatically by registering for the appropriate broadcast. To demonstrate how to receive SMS messages in this way with our `TelephonyExplorer` application, we'll implement a receiver, as shown in the following listing.

Listing 7.8. Creating an SMS-related `BroadcastReceiver`

```
public class SmsReceiver extends BroadcastReceiver {
    private static final String SMS_REC_ACTION =
        "android.provider.Telephony.SMS_RECEIVED";

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().
            equals(SmsReceiver.SMS_REC_ACTION)) {
            StringBuilder sb = new StringBuilder();
            Bundle bundle = intent.getExtras();
            if (bundle != null) {
                Object[] pdus = (Object[])
                    bundle.get("pdus");
                for (Object pdu : pdus) {
                    SmsMessage smsMessage =
                        SmsMessage.createFromPdu
                            ((byte[]) pdu);
                    sb.append("body - " + smsMessage.
                        getDisplayMessageBody());
                }
            }
            Toast.makeText(context, "SMS RECEIVED - "
                + sb.toString(), Toast.LENGTH_LONG).show();
        }
    }
}
```



To react to an incoming SMS message, we again create a custom `BroadcastReceiver` by extending that class. Our receiver defines a local constant for the `Intent` action it wants to catch, in this case, `android.provider.Telephony.SMS_RECEIVED`.

Next, we filter for the action we want on the `onReceive()` method **1**, and we get the SMS data from the `Intent extras Bundle` using the key `pdus` **2**. The `Bundle` is a hash that contains Android data types.

What's a PDU?

PDU, or protocol data unit, refers to one method of sending information along cellular networks. SMS messaging, as described in the 3rd Generation Partnership Project (3GPP) Specification, supports two different ways of sending and receiving messages. The first is text mode, which some phones don't support. Text mode encodes message content as a simple bit stream. The other is PDU mode, which contains not only the SMS message, but also metadata about the SMS message, such as text encoding, the sender, SMS service center address, and much more. To access this metadata, mobile SMS applications almost always use PDUs to encode the contents of a SMS message. For more information about PDUs and the metadata they provide, refer to the specification titled "Technical Realization of the Short Message Service (SMS)" which you can find at www.3gpp.org/ftp/Specs/html-info/23040.htm. This document, part of the 3GPP TS 23.040 Specification, is extremely technical but will help you with developing more sophisticated SMS applications.

For every `pdu Object` that we receive, we need to construct an `SmsMessage` by casting the data to a byte array **3**. After this conversion, we can use the methods in that class, such as `getDisplayMessageBody()`.

Note

If you run the example shown in [listing 7.8](#), you'll see that even though the receiver does properly report the message, the message still arrives in the user's inbox. Some applications might process specific messages themselves and prevent the user from ever seeing them; for example, you might implement a play-by-SMS chess program that uses text messages to report the other players' moves. To consume the incoming SMS message, call `abortBroadcast` from within your `onReceive()` method. Note that your receiver must have a priority level higher than that of the inbox. Also, certain versions of the Android OS don't honor this request, so test on your target devices if this behavior is important to your app.

Congratulations! Now that you've learned how to send SMS messages programmatically, set permissions appropriately, and receive and work with incoming SMS messages, you can incorporate useful SMS features into your application.

7.6. SUMMARY

Our trip through the Android telephony-related APIs covered several important topics. After a brief overview of some telephony terms, we examined Android-specific APIs.

You accessed telephony information with the `TelephonyManager`, including device and SIM card data and phone state. From there, we addressed hooking in a `PhoneStateListener` to react to phone state changes.

Besides retrieving data, you also learned how to dial the phone using built-in intents and actions, intercept outgoing phone calls, and format numbers with the `PhoneNumberUtils` class. After we covered standard voice usages, we looked at how to send and receive SMS messages using the `SmsManager` and `SmsMessage` classes.

In the next chapter, we'll turn to the specifics of interacting with notifications and alerts on the Android platform. We'll also revisit SMS, and you'll learn how to notify users of events, such as an incoming SMS, by putting messages in the status bar, flashing a light, or even making the phone vibrate.

Chapter 8. Notifications and alarms

This chapter covers

- Building an SMS notification application
- Working with `Toasts`
- Working with the `NotificationManager`
- Using alarms and the `AlarmManager`
- Setting an alarm

Today's cell phones and tablets are expected to be not only phones but personal assistants, cameras, music and video players, and instant-messaging clients, as well as to do just about everything else a computer might do. With all these applications running on phones and tablets, applications need a way to notify users to get their attention or to take some sort of action, whether in response to an SMS, a new voicemail, or an alarm reminding them of a new appointment. With Android 3.1 Google has updated notifications, refined them, and made them richer. These notifications will be part of the next version of Android, currently code-named "ice cream sandwich," which will run on handsets as well.

In this chapter, we're going to look at how to use the Android `BroadcastReceiver` and the `AlarmManager` to notify users of these sorts of events. First, we'll discuss how to display quick, unobtrusive, and nonpersistent messages called `Toasts`, based on an event. Second, we'll talk about how to create persistent messages, LED flashes, phone vibrations, and other events to alert the user. These events are called notifications. Finally, we'll look at how to trigger events by making alarm events through the `AlarmManager`. Before we go too deeply into how notifications work, let's first create a simple example application.

8.1. INTRODUCING TOAST

For this example, you'll create a simple interface that has two buttons that pop up a message, called a `Toast`, on the screen. A `Toast` is a simple, nonpersistent message designed to alert the user of an event. `Toasts` are a great way to let a user know that a call is coming in, an SMS or email has arrived, or some other event has just happened. `Toasts` are designed to take up minimal space, allowing the user to continue to interact with the system without having to stop what they're doing. `Toasts`, after popping up, fade away without user intervention. A `Toast` is different from a message,

such as a status bar notification, which persists even when a phone is turned off or until the user selects the notification or the Clear Notification button.

First let's define a simple layout.

Listing 8.1. 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="fill_parent"
    android:layout_height="fill_parent"
    >

    <TextView    android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
        />

    <Button    android:id="@+id/button_short"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Short Message"
        android:layout_x="50px"
        android:layout_y="200px"
        />

    <Button    android:id="@+id/button_long"
        android:text="Long Message"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_x="150px"
        android:layout_y="200px"
        />
```

```
</LinearLayout>
```

Next let's create the `Activity` that will display the `Toast` messages.

Listing 8.2. SimpleToast.java

```
package com.msi.manning.chapter8.SimpleToast;

import android.app.Activity;
import android.os.Bundle;
import android.widget.Button;
import android.view.View;
import android.widget.Toast;

public class SimpleToast extends Activity
{
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button button = (Button) findViewById(R.id.button_short);
        button.setOnClickListener(new Button.OnClickListener()
        {

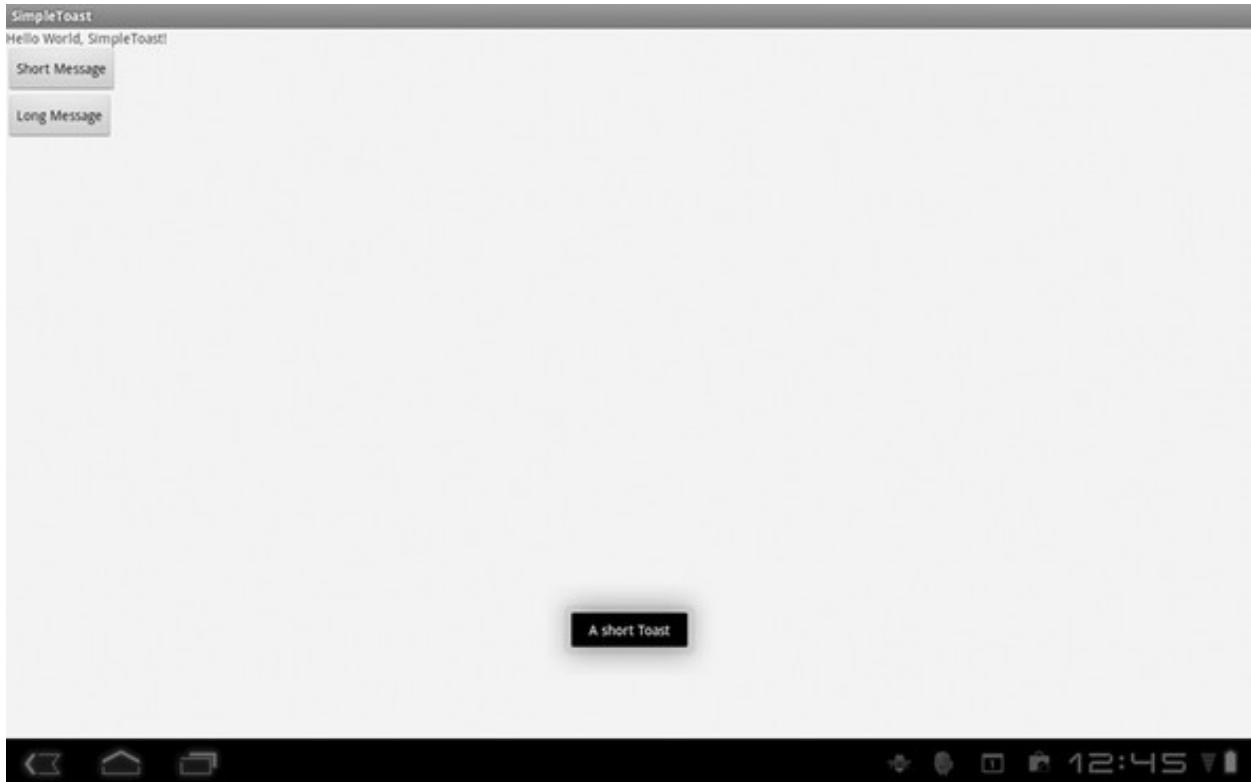
            public void onClick(View v)
            {
                Toast.makeText(SimpleToast.this, "A short Toast",
                →Toast.LENGTH_SHORT).show();           ↗ 1 Simple Toast created with
            }
        });
        button = (Button) findViewById(R.id.button_long);
        button.setOnClickListener(new Button.OnClickListener()
        {
            public void onClick(View v)
            {
                Toast.makeText(SimpleToast.this, "A Longer
                →Toast",Toast.LENGTH_LONG).show();      ↗ 2 Toast with
            }
        });
    }
}
```

As you can see, `Toasts` are simple to create. Generally all you need to do is instantiate a `Toast` object with either of the `makeText()` methods. The `makeText()` methods take three parameters: the application context, the text to display, and the length of time to display the message. Normally, the syntax looks like this:

```
Toast toast = Toast.makeText(context, text, duration);
```

The duration is always either `LENGTH_SHORT` or `LENGTH_LONG`, and `text` can be a resource id or a string. You can display the `Toast` by calling the `show()` method. In this example, we have chained the methods ①. If you run this project and click one of the buttons, you should see something like [figure 8.1](#).

Figure 8.1. Simple example of a `Toast` message on a Xoom tablet



Although `Toasts` are simple, they can be useful for providing information to users. With Android 3.0, they're more flexible, allowing custom positioning and styling that was lacking in earlier versions of Android. To show off some of these newer features, let's make a few changes to the application. First, let's look at how to reposition the `Toast` message so that instead of appearing in the default position, it shows in either the upper-right corner or the lower-left corner.

8.2. PLACING YOUR TOAST MESSAGE

We want to display our short message in the upper-right corner. To do that, we can use one of the `Toast`'s other methods: `setGravity()`. The `setGravity()` method allows you to define exactly where you would like a `Toast` message to appear. It takes three

parameters: the `Gravity` constant, an x-position offset, and a y-position offset. The syntax looks like this:

```
toast.setGravity(Gravity.TOP|Gravity.LEFT, 0, 0);
```

To use it in the example code, change the first `Toast` from

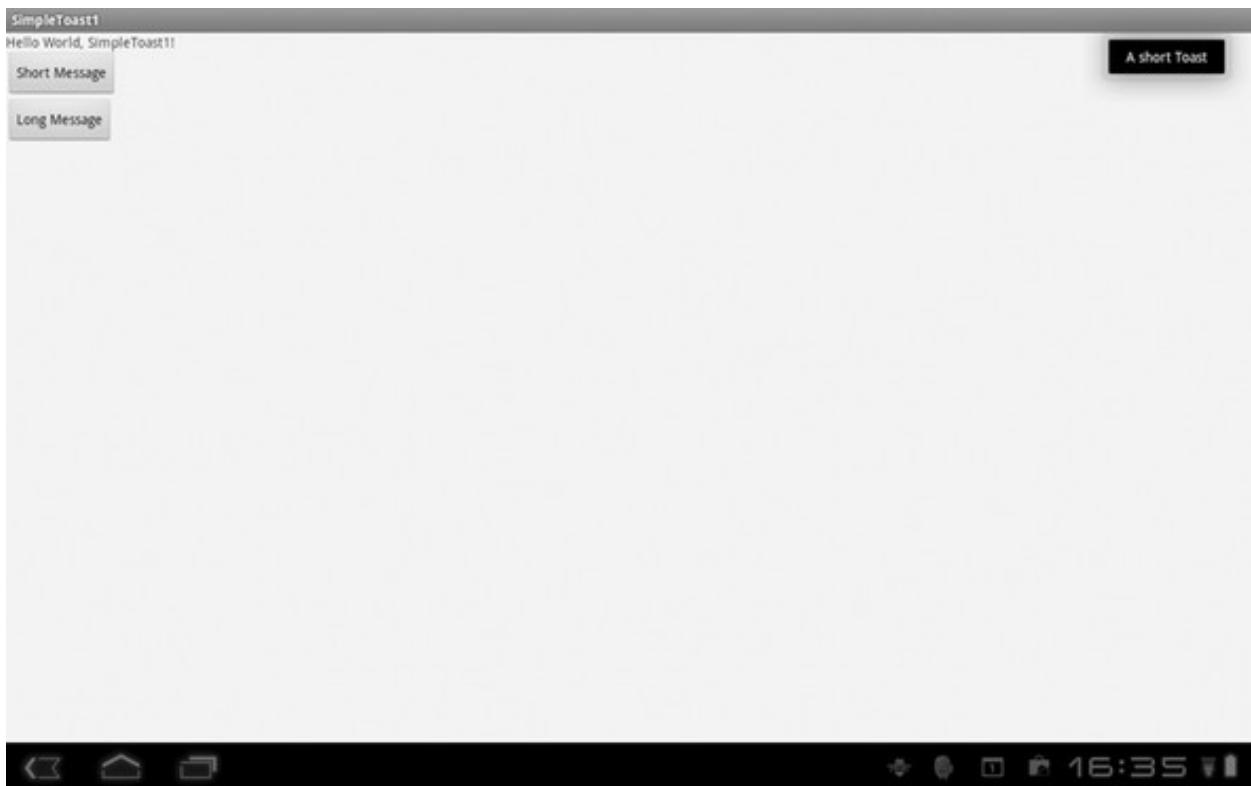
```
Toast.makeText(SimpleToast.this, "A short Toast", Toast.LENGTH_SHORT).show();
```

to

```
Toast toast = Toast.makeText(SimpleToast.this, "A short Toast",
    Toast.LENGTH_SHORT);
toast.setGravity(Gravity.TOP|Gravity.RIGHT, 0, 0);
toast.show();
```

If you run this code, you should now see the `Toast` message in the upper-right corner of your device, as shown in [figure 8.2](#).

Figure 8.2. Custom positioning of a `Toast` message



You can make the positioning much more specific by using the x and y offsets. Now that you know how to position a `Toast` wherever you want, let's make a truly custom `Toast` by making a specialized `Toast` view.



Note

For more information, see the `Gravity` class: <http://developer.android.com/reference/android/view/Gravity.html>.



8.3. MAKING A CUSTOM TOAST VIEW

Making a custom `Toast` view is a little more involved than specifying its position, but as you'll see, it's still straightforward. To make the custom `Toast` view, you first need to define a new layout specifically for that `Toast` view. You can do this a number of ways, including in your application's code or in XML. Then all you need to do is pass the view to `setView(View)` when you create the `Toast` message to display.

Let's create a new XML layout called `customtoast.xml`, in the `layout` directory.

Listing 8.3. Customtoast.xml

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

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toast_layout_root"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="10dp"
    android:background="#DAAA" android:orientation="horizontal">

    <ImageView android:id="@+id/mandsm"
        android:layout_width="wrap_content"
        android:layout_height="fill_parent"
        android:layout_marginRight="10dp">

```

```

    />

<TextView android:id="@+id/text"
          android:layout_width="wrap_content"
          android:layout_height="fill_parent"
          android:textColor="#FFFF"
    />

</LinearLayout>

```

As you can see, this is a simple layout. We define an `ImageView` to present a graphic and a `TextView` to replace our need to use the `makeText()` method.

Now we need to change the application code. For this example, we'll only change the first `Toast` message to use the custom view. Assuming you're editing our original code, you can change the code in the first `onClick()` method to look like the follow listing.

Listing 8.4. Modified application code

```

protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    LayoutInflator inflater = getLayoutInflator();           ① Instantiate layout XML
    final View layout = inflater.inflate(R.layout.customtoast,
                                         (ViewGroup) findViewById(R.id.toast_layout_root));  ② Inflate XML

    ImageView image = (ImageView) layout.findViewById(R.id.mandsm);
    image.setImageResource(R.drawable.mandsm);
    TextView text = (TextView) layout.findViewById(R.id.text);
    text.setText("Short custom message");

    Button button = (Button) findViewById(R.id.button_short);
    button.setOnClickListener(new Button.OnClickListener()
    {
        public void onClick(View v)
        {
            Toast toast = new Toast(getApplicationContext());
            toast.setGravity(Gravity.TOP|Gravity.RIGHT, 200, 200);
            toast.setDuration(Toast.LENGTH_LONG);
            toast.setView(layout);                                ③ Hold new
            toast.show();                                       Toast view
        }
    });
}

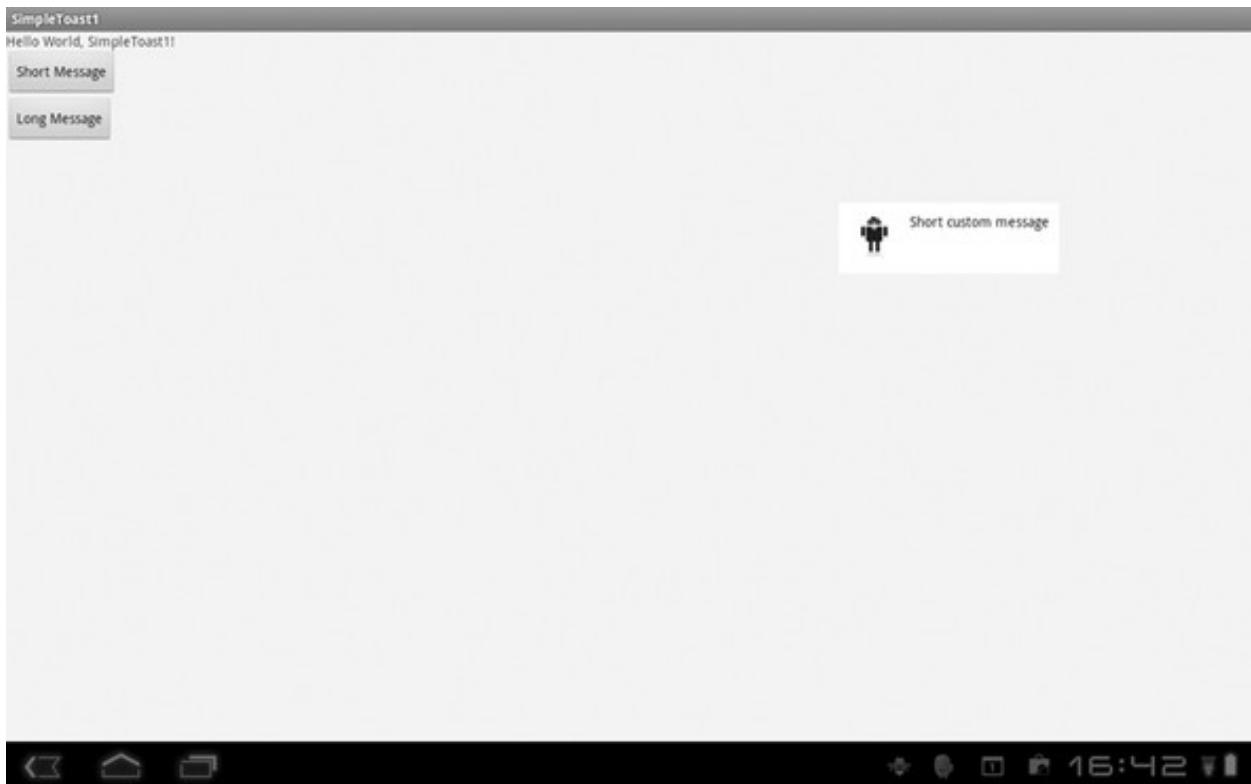
```

We use several classes and methods that you should be familiar with from earlier chapters. The `getLayoutInflator()` method retrieves the `LayoutInflater` to instantiate

the customtoast layout XML ①. Then we use the inflator to inflate the XML ②. Next we set the ImageView and the TextView, from the custom layout, and after that we create a Toast. Note that we don't use `makeText()` when we create the Toast, because we've defined a custom TextView and ImageView ③.

When you run the code, you should get a result like that shown in [figure 8.3](#).

Figure 8.3. A custom `Toast` message with embedded graphic



Toasts are used in almost all major Android applications. But sometimes you need to call the user's attention to an event until the user takes some sort of action; a `Toast` can't do this, because it goes away on its own. Such persistent messages to the user are called notifications; in the next section, we'll look at how they work and what you can do with them.

8.4. INTRODUCING NOTIFICATIONS

In the previous section, we showed how simple it is to create a quick, unobtrusive message to let the user know that some event has happened or to provide them some useful information. In this section, we're going to look at how to create a persistent notification that not only shows up in the status bar, but stays in a notification area until

the user deletes it. To do that, we need to use the classes `Notification` and `NotificationManager`.

8.4.1. The Notification class

A notification on Android can be many things, ranging from a pop-up message, to a flashing LED, to a vibration, but all these actions start with and are represented by the `Notification` class. The `Notification` class defines how you want to represent a notification to a user. This class has three constructors, one public method, and a number of fields. Table 8.1 summarizes the class.

Table 8.1. `Notification` fields

Access Type	Method	Description
public int	audioStreamType	Stream type to use when playing a sound
public RemoteViews	contentView	View to display when the statusBar-Icon is selected in the status bar
public PendingIntent	contentIntent	Intent to execute when the icon is clicked
public int	defaults	Defines which values should be taken from defaults
public int	deleteIntent	Intent to execute when the user clicks the Clear All Notifications button
public flags		Places all flags in the flag fields as bits
public PendingIntent	fullScreenIntent	Intent to launch instead of posting the notification in status bar
public int	icon	Resource ID of a drawable to use as the icon in the status bar
public Int	iconLevel	Level of an icon in the status bar
public bitmap	largeIcon	Bitmap that may be bigger than the bounds of the panel
public int	ledARGB	Color of the LED notification
public int	ledOffMS	Number of milliseconds for the LED to be off between flashes
public int	ledOnMS	Number of milliseconds for the LED to be on for each flash
public Int	number	Number of events represented by this notification
public ContentURI	sound	Sound to play
public CharSequence	tickerText	Text to scroll across the screen when this item is added to the status bar
public RemoteViews	tickerView	View shown by the ticker notification in the status bar

Access Type	Method	Description
public long[]	vibrate	Vibration pattern to use
public long	when	Timestamp for the notification

As you can see, the `Notification` class has numerous fields; it has to describe every way you can notify a user. The `NotificationManager` class, though, is required in order to use the `Notification` class, because it's the system service that executes and manages notifications. Using a notification follows these steps:

```
NotificationManager myNotificationManager;

private static final int NOTIFICATION_ID = 1;

myNotificationManager =
(NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
```

Here we set up a `NotificationManager` and instantiate it.

Next we use the `Notification.Builder` to set `Notification` objects such as the message icon for the notification, the title, and much more. The `Notification.Builder` provides a much simpler mechanism for building notifications than in previous versions of Android:

```
Notification.Builder builder = new Notification.Builder(this);

builder.setTicker("Message to Show when Notification pops up");

builder.setContentTitle ("Title of Message");

builder.setSmallIcon(R.drawable.icon);

builder.setContentText("- Message for the User -");

Intent notificationIntent = new Intent(this, SimpleNotification.class);

PendingIntent contentIntent = PendingIntent.getActivity(this, 0, notificationIntent,
0);

builder.setContentIntent(contentIntent);
```

Next we create `PendingIntent` for the `Builder`. You must create a `PendingIntent` for all notifications.

Finally, to send the notification, all you have to do is use the `notify()` method and supply the `Notification` ID as well as the builder:

```
myNotificationManager.notify(NOTIFICATION_ID, builder.getNotification());
```

Here the `notify()` method wakes up a thread that performs the notification task you have defined. You can use either an `Activity` or a `Service` to trigger the notification, but generally you'll want to use a `Service` because a `Service` can trigger a notification in the background regardless of whether it's the active application at the time.

8.4.2. Notifying a user with a simple button press

Based on the previous example, let's make a simple interface with two buttons: one that will trigger the notification and one that will clear it. Make a new project, and first define the layout in `main.xml` as in the following listing.

Listing 8.5. 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="fill_parent"

    android:layout_height="fill_parent"

>

<TextView    android:layout_width="fill_parent"

    android:layout_height="wrap_content"

    android:text="@string/hello"

/>

<Button    android:id="@+id/button_cn"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="Create Notification"

    android:layout_x="50px"

    android:layout_y="200px"

/>

<Button    android:id="@+id/button_dn"

    android:text="Clear Notification"
```

```

    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_x="150px"
    android:layout_y="200px"
/>
</LinearLayout>

```

Now that we have our layout, let's create the `Activity` that will trigger the notification. (We're using an `Activity` in this example for simplicity's sake.) Make an `Activity` called `SimpleNotification`, as shown in the following listing.

Listing 8.6. SimpleNotification.java

```

public class SimpleNotification extends Activity
{
    NotificationManager myNotificationManager;
    private static final int NOTIFICATION_ID = 1;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Button myGen = (Button) findViewById(R.id.button_cn);
        myGen.setOnClickListener(myGenOnClickListener);
        Button myClear = (Button) findViewById(R.id.button_dn);
        myClear.setOnClickListener(myClearOnClickListener);
    }

    private void GenerateNotification(){
        myNotificationManager =
        ↪(NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
        Notification.Builder builder = new Notification.Builder(this);
        builder.setContentTitle ("Attention Please!");
        builder.setTicker("**** Notification ***");
        builder.setSmallIcon(R.drawable.notand);
        builder.setContentText("- Message for the User -");
    }
}

```

The code in Listing 8.6 contains two annotations:

- An annotation pointing to the line `myNotificationManager =` with the text "Get reference to NotificationManager" and a circled number "1".
- An annotation pointing to the line `Notification.Builder builder = new Notification.Builder(this);` with the text "Set up new Notification.Builder" and a circled number "2".

```

Intent notificationIntent =
↳new Intent(this, SimpleNotification.class);
PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
↳notificationIntent, 0);
builder.setContentIntent(contentIntent);

myNotificationManager.notify(NOTIFICATION_ID,
↳builder.getNotification());
}

Button.OnClickListener myGenOnClickListener =
new Button.OnClickListener(){
    public void onClick(View v) {
        GenerateNotification();
    }
};

Button.OnClickListener myClearOnClickListener =
new Button.OnClickListener(){
    public void onClick(View v) {
        myNotificationManager.cancel(NOTIFICATION_ID);
    }
};

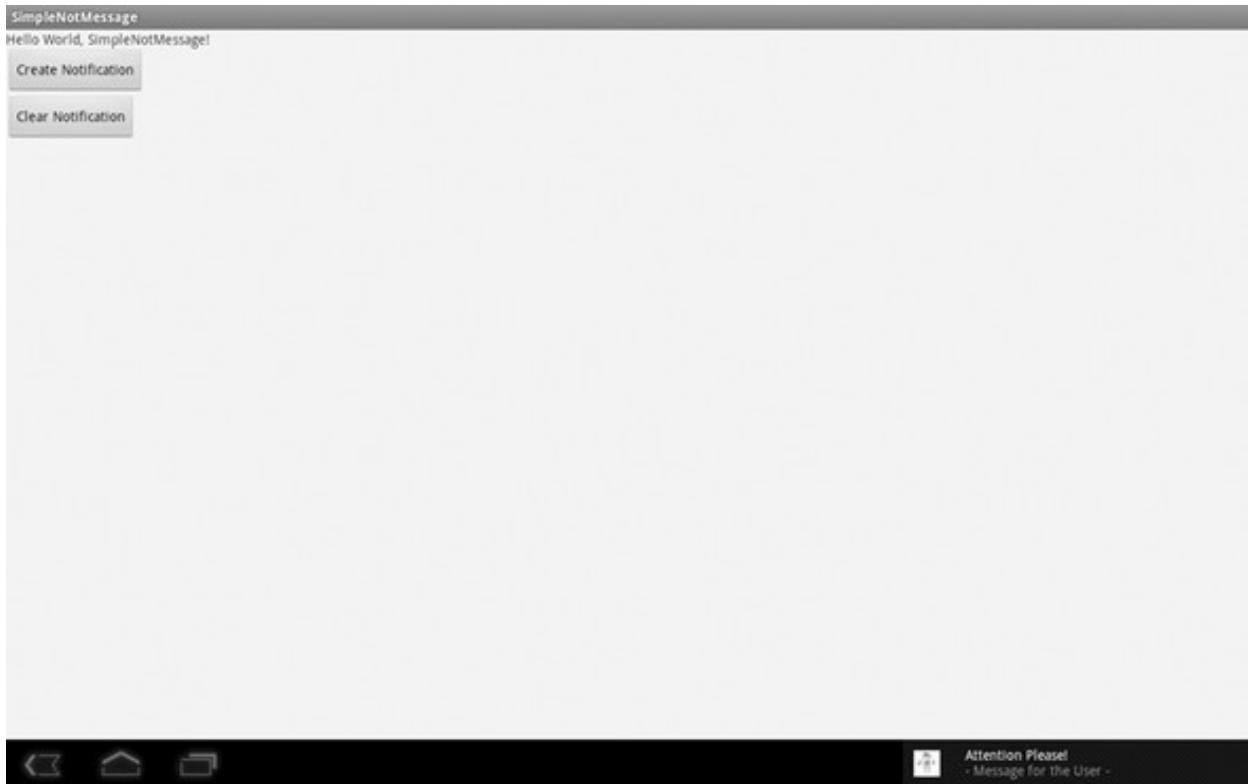
```

3 Set up Intent and PendingIntent

As you can see, the code for creating a notification is straightforward and follows the same process we outlined earlier. In this example, we have two buttons. The first calls the `GenerateNotification()` method, where we first get a reference to

the `NotificationManager` ①. Then we build the message that we'll pass to the user ②. Next we set up a `PendingIntent` ③ and then send the notification. If you build the project, run it, and click the Create Notification button, you should see something like [figure 8.4](#).

Figure 8.4. Notification being displayed on a Xoom tablet



8.5. MAKING A CUSTOM NOTIFICATION VIEW

Just like `Toasts`, you can make custom views for notifications. One excellent example is the Gmail application that ships with the Xoom. If you check your email, the notifications that pop up in the notification area include not only the subject of the email but, if available, the image associated with the person. Much as you do with `Toasts`, to create a custom view for a `Notification` you need to first define a layout either in the application code or in XML. For this example, we'll use XML. In the example, create the `customnotification.xml` file in your layout directory.

Listing 8.7. `customnotification.xml`

```
<?xml version="1.0" encoding="utf-8"?>  
  
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
        android:layout_width="match_parent"  
        android:layout_height="match_parent"  
        android:background="#000" android:orientation="horizontal">  
    >
```

```
<ImageView android:id="@+id/avatar"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:layout_marginRight="10dp"
    />

<TextView android:id="@+id/text"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:textColor="#FFF"
    />

<TextView android:id="@+id/textTicker"
    android:layout_width="wrap_content"
    android:layout_height="fill_parent"
    android:textColor="#FFF"
    />

</LinearLayout>
```

Now we'll change the application code somewhat from the previous notification example, to use the new custom layout.

Listing 8.8. SimpleNotification.java

```

private void GenerateNotification(){

    String contenttitle = "Attention Please!";
    String contenttext = "- Message for the User -" ;

    RemoteViews layout = new RemoteViews(getApplicationContext(),
        R.layout.customnotification); 1 Define image and  
text for layout
    layout.setTextViewText(R.id.text, contenttitle);
    layout.setTextViewText(R.id.textTicker, contenttext);
    layout.setImageResource(R.id.avatar, R.drawable.mandsm);

    myNotificationManager =
        (NotificationManager) getSystemService(Context.NOTIFICATION_SERVICE);
    Notification.Builder builder = new Notification.Builder(this);
    builder.setContentTitle (contenttitle);
    builder.setTicker("**** Notification ****");
    builder.setSmallIcon(R.drawable.notand);
    builder.setContentText(contenttext); 2 Pass RemoteViews  
reference to  
Notification.Builder
    builder.setContent(layout);

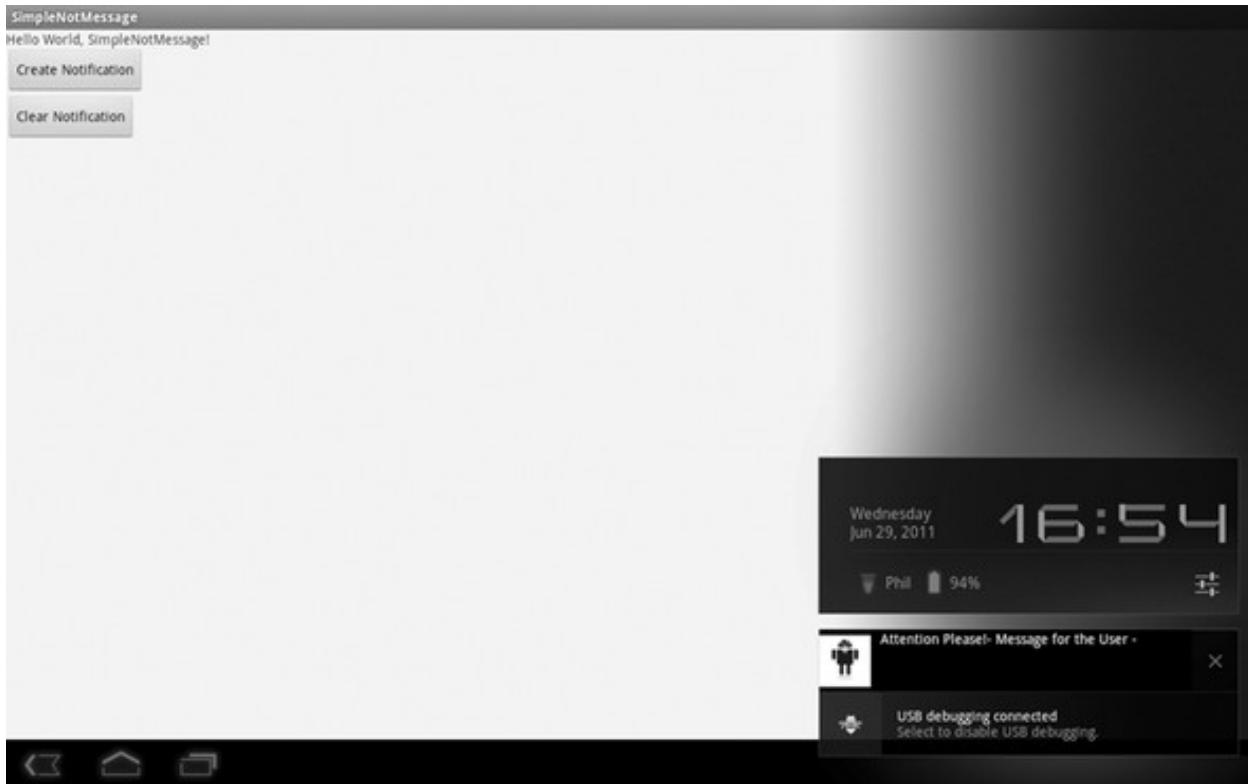
    Intent notificationIntent = new Intent(this, SimpleNotMessage.class);
    PendingIntent contentIntent = PendingIntent.getActivity(this, 0,
        notificationIntent, 0);
    builder.setContentIntent(contentIntent);

    myNotificationManager.notify(NOTIFICATION_ID, builder.getNotification());
}

```

We use a `RemoteViews` to set the text and image we plan to use inside the notification 1. When we've done that, we can pass the reference to the `RemoteViews` to the `Notification.Builder` that will create the customer notification when the new code is run 2. Run the code and click Create Notification to open the notification shown in [figure 8.5](#).

Figure 8.5. Custom styling for a notification on the Xoom



As you can see, creating a custom notification is straightforward. That being said, you can make a notification even more sophisticated by having it flash an LED, play sounds, vibrate the device, or perform any other number of actions by using the `Notification.Builder`'s various setters. For example, you could have the previous code turn on an LED on the device by adding this single line:

```
builder.setLights(0xFFff0000,1000,100);
```

The first parameter is the Red, Green, Blue (RGB) value of the LED, the second value is the number of milliseconds the LED should stay on, and the last value is how long the LED should stay off before going back on.

You could also add sound to a notification using the `Notification.Builder` like this:

```
builder.setSound(Uri.parse("File:///sdcard/music/Travis-Sing.mp3"));
```

There are also numerous other options. For now, we'll move on and look at alarms in Android 3.0.

8.6. INTRODUCING ALARMS

In Android, alarms allow you to schedule your application to run at some point in the future. Alarms can be used for a wide range of applications, from notifying a user of an appointment to something more sophisticated, such as having an application start, checking for software updates, and then shutting down. An alarm works by registering an `Intent` with the alarm; at the scheduled time, the alarm broadcasts the `Intent`. Android automatically starts the targeted application, even if the Android handset is asleep.

Android manages all alarms somewhat as it manages the `NotificationManager`—via an `AlarmManager` class. The `AlarmManager` has the methods described [table 8.2](#).

Table 8.2. `AlarmManager` public methods

Returns Method	description
<code>void cancel(PendingIntent intent)</code>	Remove alarms with matching Intent.
<code>void set(int type, long triggerAtTime, PendingIntent operation)</code>	Set an alarm.
<code>void setInexactRepeating(int type, long triggerAtTime, long interval, PendingIntent operation)</code>	Repeating alarm that has inexact trigger requirements.
<code>void setRepeating(int type, long triggerAtTime, long interval, PendingIntent operation)</code>	Set a repeating alarm.
<code>void setTime(long milliseconds)</code>	Set the time for an alarm.
<code>void setTimeZone(String TimeZone)</code>	Set the time zone for the alarm.

You retrieve the `AlarmManager` indirectly (as you do the `NotificationManager`), by using `Context.getSystemService(Context.ALARM_SERVICE)`.

Setting alarms is easy, like most things in Android. In the next example, you'll create a simple application that sets an alarm when a button is clicked. When the alarm is triggered, it will pass back a simple `Toast` to inform you that the alarm has been fired.

8.6.1. Creating a simple alarm example

In this next example, you'll create an Android project called `SimpleAlarm` that has the package name `com.msi.manning.chapter8.simpleAlarm`, the application name `SimpleAlarm`, and the `Activity` name `GenerateAlarm`. This project uses another open source icon, which you can find at www.manning.com/ableson3/ or in the download for this chapter. Change the name of the icon to `clock`, and add it to the `res/drawable` directory of the project when you create it.

Next, edit the `AndroidManifest.xml` file to have a receiver (you'll create that soon) called `AlarmReceiver`, as shown in the following listing.

Listing 8.9. `AndroidManifest.xml`

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.msi.manning.chapter8.simpleAlarm">

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

    <application android:icon="@drawable/clock">

        <activity android:name=".GenerateAlarm"

            android:label="@string/app_name">

            <intent-filter>

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

                <category android:name="android.intent.category.LAUNCHER" />

            </intent-filter>

        </activity>

        <receiver android:name=".AlarmReceiver" android:process=":remote" />

    </application>

</manifest>
```

Next, edit the `string.xml` file in the `values` directory, and add two new strings:

```
<string name="set_alarm_text">Set Alarm</string>

<string name="alarm_message">Alarm Fired</string>
```

You'll use this as the value of the button in the layout. Next, edit the `main.xml` file to add a new button to the layout:

```
<Button android:id="@+id/set_alarm_button"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="@string/set_alarm_text">

    <requestFocus />

</Button>
```

You're ready to create a new class that will act as the `Receiver` for the notification the alarm will generate, as shown in the following listing. In this case, you'll generate a `Toast`-style notification to let the user know the alarms have been triggered. This class waits for the alarm to broadcast to the `AlarmReceiver` and then generates the `Toast`.

Listing 8.10. AlarmReceiver.java

```
public class AlarmReceiver extends BroadcastReceiver {  
    public void onReceive(Context context, Intent intent) {  
        try {  
            Toast.makeText(context, R.string.app_name,  
↳Toast.LENGTH_SHORT).show(); }  
            catch (Exception r) { Toast.makeText(context, "woops",  
↳Toast.LENGTH_SHORT).show(); }  
    }  
}
```

 **Broadcast Toast when
Intent is received**

Next, edit the `SimpleAlarm` class to create a button widget (as discussed in [chapter 3](#)) that calls the inner class `setAlarm`. In `setAlarm`, we create an `onClick` method that will schedule the alarm, call the `Intent`, and fire off the `Toast`. The following listing shows what the finished class should look like.

Listing 8.11. GenerateAlarm.java

```

public class GenerateAlarm extends Activity {
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        Button button = (Button) findViewById(R.id.set_alarm_button);
        button.setOnClickListener(this.mOneShotListener);
    }

    private OnClickListener mOneShotListener = new OnClickListener() {
        public void onClick(View v) {
            Intent intent = new Intent(GenerateAlarm.this,
            ↪AlarmReceiver.class
            PendingIntent appIntent =
            ↪PendingIntent.getBroadcast(GenerateAlarm.this,
            ↪0, intent, 0);
            long triggerAlarm = System.currentTimeMillis() + 30000;
            AlarmManager am = (AlarmManager)
            ↪getSystemService(Context.ALARM_SERVICE);
            am.set(AlarmManager.RTC_WAKEUP, triggerAlarm, appIntent);
        }
    };
}

```

The code is annotated with three numbered callouts:

- 1**: Points to the line `button.setOnClickListener(this.mOneShotListener);` with the text "Set up Button to call mOneShotListener".
- 2**: Points to the line `am.set(AlarmManager.RTC_WAKEUP, triggerAlarm, appIntent);` with the text "Create Intent to fire when alarm goes off".
- 3**: Points to the line `AlarmManager am = (AlarmManager)` with the text "Create AlarmManager".

As you can see, this class is simple. We create a `Button` to trigger the alarm **1**. Next, we create an inner class for `mOneShotListener`. Then, we create the `Intent` to be

triggered when the alarm goes off **2**. In the next section of code, we use the `Calendar` class to help calculate the number of milliseconds from the time the button is clicked, which we'll use to set the alarm.

Now we've done everything necessary to create and set the alarm. We create

the `AlarmManager` **3** and then call its `set()` method to set the alarm. To see a little more detail of what's going on in the application, look at these lines of code:

```

AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);

am.set(AlarmManager.RTC_WAKEUP, triggerAlarm, appIntent);

```

These lines are where we create and set the alarm by first using `getSystemService()` to create the `AlarmManager`. The first parameter we pass to the `set()` method is `RTC_WAKEUP`, which is an integer representing the alarm type we want to set.

The `AlarmManager` currently supports four alarm types, as shown in [table 8.3](#).

Table 8.3. `AlarmManager` alarm types

Type	Description
ELAPSED_REALTIME	Alarm time in <code>SystemClock.elapsedRealtime()</code> (time since boot, including sleep).
ELAPSED_REALTIME_WAKEUP	Alarm time in <code>SystemClock.elapsedRealtime()</code> (time since boot, including sleep). This will wake up the device when it goes off.
RTC	Alarm time in <code>System.currentTimeMillis()</code> (wall clock time in UTC).
RTC_WAKEUP	Alarm time in <code>System.currentTimeMillis()</code> (wall clock time in UTC). This will wake up the device when it goes off.

You can use multiple types of alarms, depending on your requirements. `RTC_WAKEUP`, for example, sets the alarm time in milliseconds; when the alarm goes off, it'll wake the device from sleep mode for you, as opposed to `RTC`, which won't.

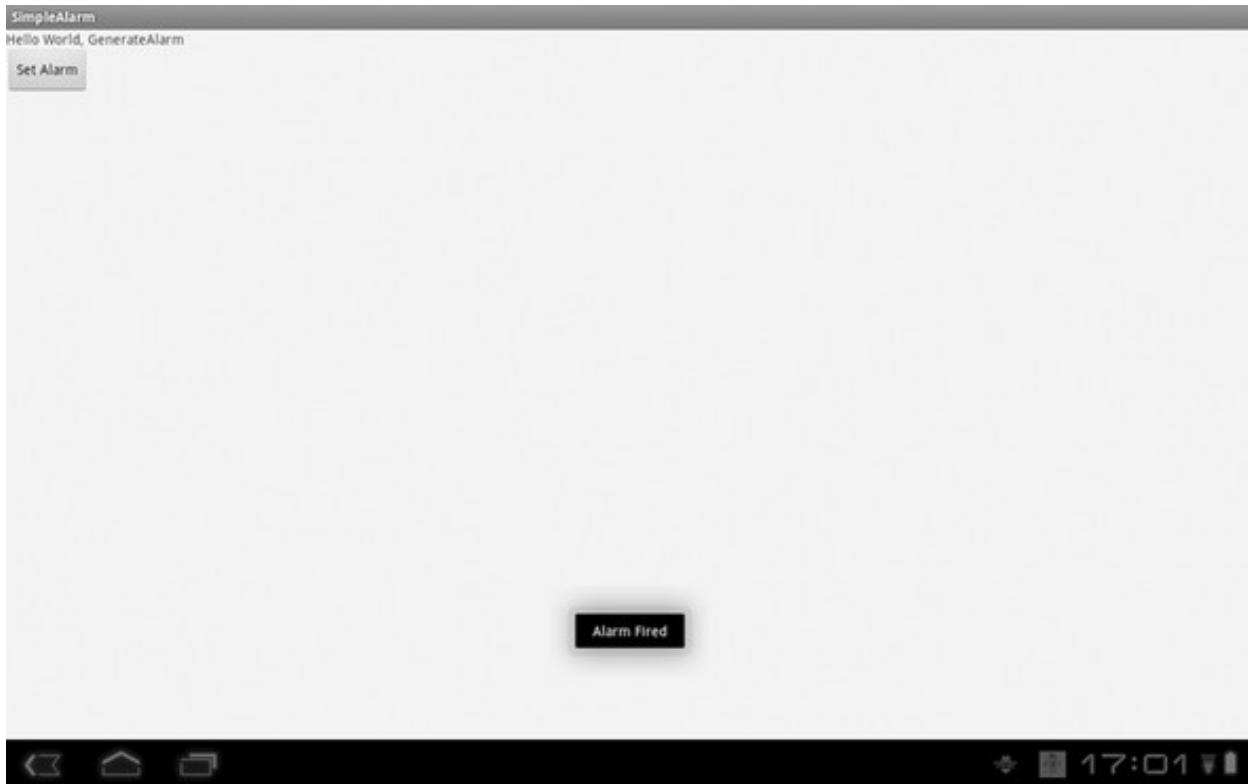
The next parameter we pass to the method is the time, in milliseconds, for when we want the alarm to be triggered. We do this with the following snippet by adding the number of milliseconds to the current time:

```
long triggerAlarm = System.currentTimeMillis() + 30000
```

The last parameter is the `Intent` to which we want to broadcast, which is the `Intent-Receiver`. Now, build the application and run it.

Clicking the Set Alarm button sets the alarm; after 30 seconds, you should see something like [figure 8.6](#), displaying the `Toast` message.

Figure 8.6. After the alarm runs, the application shows a simple `Toast` message.



8.6.2. Using notifications with alarms

Creating an alarm is pretty easy in Android, but what might make more sense would be for that alarm to trigger a notification in the status bar. To do that, you need to add a `NotificationManager` and generate a `Notification`. We've created a new method to add to [listing 8.11](#) called `showNotification()`, which takes four parameters and creates a `Notification`:

```
Intent contentIntent = new Intent(this, SetAlarm.class);

PendingIntent theappIntent = PendingIntent.getBroadcast(SetAlarm.this, 0,
contentIntent, 0);

Notification.Builder builder = new Notification.Builder(this);

builder.setContentTitle ("Attention Please!");

builder.setTicker("Alarm");

builder.setSmallIcon(statusBarIconID);

builder.setContentText("- Message for the User -");

builder.setContentIntent(theappIntent);
```

```
nm.notify(NOTIFICATION_ID, builder.getNotification());
```

Much of this code is similar to the `SimpleNotMessage` code. To add it to your `GenerateAlarm`, edit [listing 8.10](#) to look like [listing 8.12](#); the only other things we've done are to import the `Notification` and `NotificationManager` into the code and add the private variables `nm` and `NOTIFICATION_ID`.

Listing 8.12. `AlarmReceiver.java`

```
public class AlarmReceiver extends BroadcastReceiver {  
    private NotificationManager nm;  
    private int NOTIFICATION_ID;  
  
    public void onReceive(Context context, Intent intent) {  
        this.nm = (NotificationManager)  
            ↪context.getSystemService(Context.NOTIFICATION_SERVICE);  
  
        Intent contentIntent = new Intent(context, AlarmReceiver.class);  
        PendingIntent theappIntent = PendingIntent.getBroadcast(context, 0,  
            ↪contentIntent, 0);  
  
        Notification.Builder builder =  
            ↪new Notification.Builder(context);  
        builder.setContentTitle ("Attention Please!");  
        builder.setTicker("Alarm");  
        builder.setSmallIcon(R.drawable.alarm);  
        builder.setContentText("- Message for the User -");  
        builder.setContentIntent(theappIntent);  
        nm.notify(NOTIFICATION_ID, builder.getNotification());  
        abortBroadcast();  
    }  
}
```

Build
notification

Now edit `GenerateAlarm.java` so it looks like [listing 8.13](#).

Listing 8.13. `GenerateAlarm.java`

```
}  
  
public class GenerateAlarm extends Activity {  
  
    Toast mToast;  
  
    protected void onCreate(Bundle icicle) {  
        super.onCreate(icicle);  
  
        setContentView(R.layout.main);
```

```

        Button button = (Button) findViewById(R.id.set_alarm_button);

        button.setOnClickListener(this.mOneShotListener);

    }

private OnClickListener mOneShotListener = new OnClickListener() {
    public void onClick(View v) {

        Intent intent = new Intent(GenerateAlarm.this, AlarmReceiver.class);

        PendingIntent appIntent = PendingIntent.getBroadcast(GenerateAlarm.this,
0, intent, 0);

        Calendar calendar = Calendar.getInstance();

        calendar.setTimeInMillis(System.currentTimeMillis());

        calendar.add(Calendar.SECOND, 30);

        AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);

        am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(), appIntent);

        NotificationManager nm = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);

        nm.cancel(R.string.app_name);

        Toast.makeText(GenerateAlarm.this, "alarm fired wait 30 seconds",
Toast.LENGTH_SHORT).show();

    }

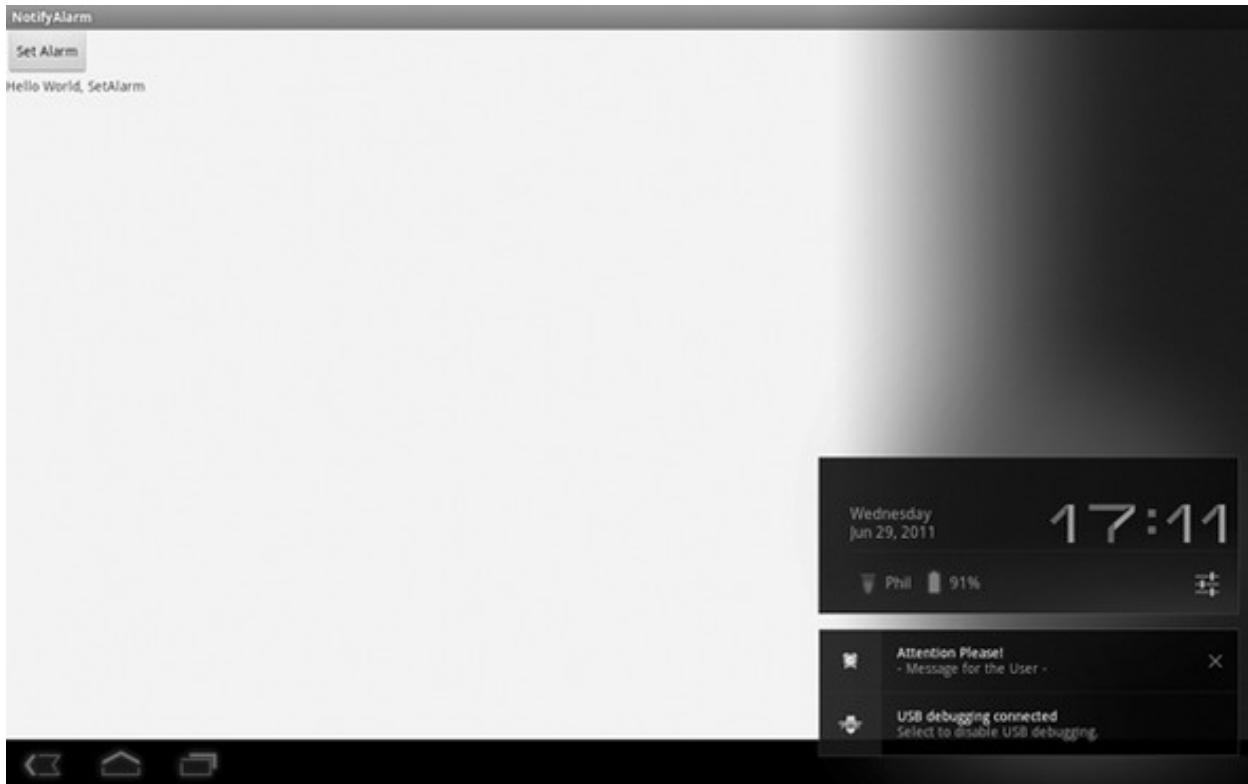
};

}

```

If you run the code and click Set Alarm, you should see the alarm notification in the status bar as shown in [figure 8.7](#). You could easily edit this code to take parameters for time and date, have it show different Intents when the icons are clicked, and so on.

Figure 8.7. Alarm notification shown in the status bar



As you can see from this example, Android alarms and the `AlarmManager` are straightforward, and you should be able to easily integrate them into your applications.

8.7. SUMMARY

In this chapter, we've looked at three separate but related items: `Toast`, `Notification`, and `Alarm`. You learned that for simple, nonpersistent messages, the `Toast` class provides an easy and convenient way to alert the user. We also discussed how to use the `NotificationManager` to generate simple to relatively complex notifications. Then you used the `Notification` class to present a notification to the user by building an example that displays a message in the status bar, vibrates a phone, or even flashes an LED when an SMS messages arrives in the inbox.

We also looked at how to set an alarm to cause an application to start or take some action in the future, including waking the system from sleep mode. Finally, we talked about how to trigger a notification from an alarm. Although the code presented in these examples gives you a taste of what can be done, notifications and alarms both have broad applications limited only by your imagination.

Now that you have an understanding of how to work with the `Notification` and `Alarm` classes, we're going to move on a discussion of graphics and animation. In [chapter 9](#), you'll learn the basic methods of generating graphics in Android,

how to create simple animations, and even how to work with OpenGL to generate stunning 3D graphics.

Chapter 9. Graphics and animation

This chapter covers

- Drawing graphics in Android
- Applying the basics of OpenGL for embedded systems (ES)
- Animating with Android

By now, you should've picked up on the fact that it's much easier to develop Android applications than it is to use other mobile application platforms. This ease of use is especially apparent when you're creating visually appealing UIs and metaphors, but there's a limit to what you can do with typical Android UI elements (such as those we discussed in [chapter 3](#)). In this chapter, we'll look at how to create graphics using Android's Graphics API, discuss how to develop animations, and explore Android's support for the OpenGL standard, as well as introduce you to Android's new cross-platform, high-performance graphics language RenderScript. (To see examples of what you can do with Android's graphics platform, go to www.omnigsoft.com/Android/ADC/readme.html.)

First, we're going to show you how to draw simple shapes using the Android 2D Graphics API, using Java and then XML to describe 2D shapes. Next, we'll look at making simple animations using Java and the Graphics API to move pixels around, and then using XML to perform a frame-by-frame animation. After that we'll examine Android's support of the OpenGL ES API, make a simple shape, and then make a more complex, rotating, three-dimensional shape. Finally we'll introduce RenderScript, a low-level, C-derived, native language that allows developers to take advantage of multicore systems and graphics accelerators to make more performant, visually intensive applications.

If you've ever worked with graphics in Java, you'll likely find the Graphics API and how graphics work in Android familiar. If you've worked with OpenGL, you'll find Android's implementation of OpenGL ES reasonably straightforward. You must remember, though, that cell phones, tablets, and other mobile devices don't have the graphical processing power of a desktop. Regardless of your experience, you'll find the Android Graphics API both powerful and rich, allowing you to accomplish even some of the most complex graphical tasks.

Note

You can find more information on the differences between OpenGL and OpenGL ES to help you determine the level of effort in porting code at the Khronos website. For example, the OpenGL ES 1.5 specification at <http://mng.bz/qapb> provides information on differences between OpenGL and OpenGL ES.

9.1. DRAWING GRAPHICS IN ANDROID

In this section, we'll cover Android's graphical capabilities and show you examples of how to make simple 2D shapes. We'll be applying the `android.graphics` package (see <http://mng.bz/CIFJ>), which provides all the low-level classes you need to create graphics. The graphics package supports such things as bitmaps (which hold pixels), canvases (what your draw calls draw on), primitives (such as rectangles and text), and paints (which you use to add color and styling). Although these aren't the only graphics packages, they're the main ones you'll use in most applications. Generally, you use Java to call the Graphics API to create complex graphics.

To demonstrate the basics of drawing a shape with Java and the Graphics API, let's look at a simple example in the following listing, where we'll draw a rectangle.

Listing 9.1. simpleshape.java

```

package com.msi.manning.chapter9.SimpleShape;
public class SimpleShape extends Activity {
@Override
protected void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(new SimpleView(this));
}
private static class SimpleView extends View {
    private ShapeDrawable mDrawable =
        new ShapeDrawable();
    public SimpleView(Context context) { ← ② Set up View
        super(context);
        setFocusable(true);
        this.mDrawable =
            new ShapeDrawable(new RectShape());
        this.mDrawable.getPaint().setColor(0xFFFF0000);
    }
    @Override
    protected void onDraw(Canvas canvas) {
        int x = 10;
        int y = 10;
        int width = 300;
        int height = 50;
        this.mDrawable.setBounds(x, y, x + width, y + height);
        this.mDrawable.draw(canvas);
        y += height + 5;
    }
}
}

```

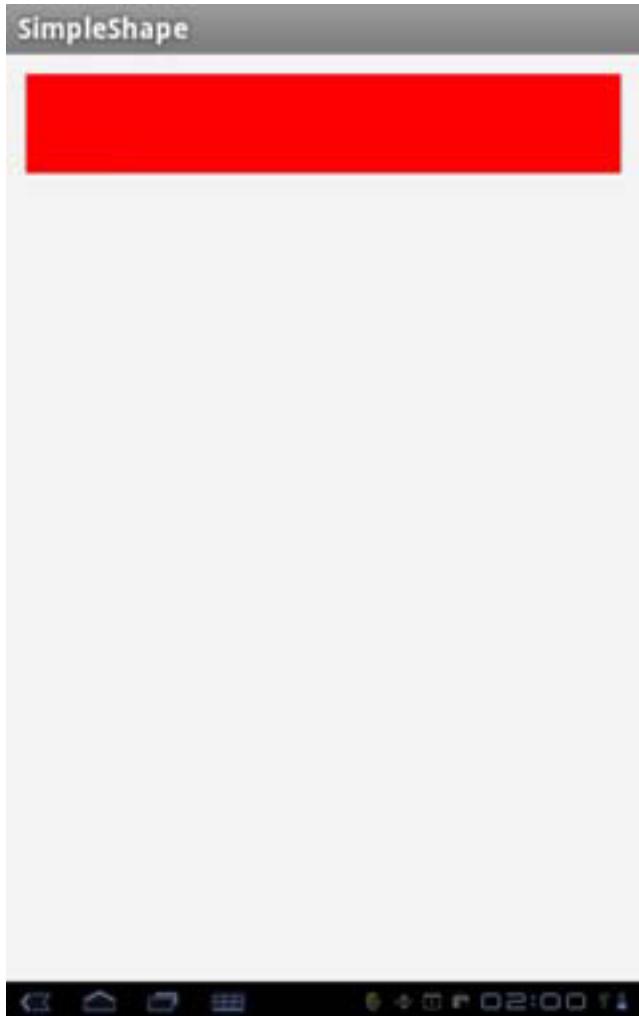
The diagram consists of three numbered callouts with arrows pointing to specific parts of the code. Callout 1, at the top right, points to the line 'private ShapeDrawable mDrawable ='. Callout 2, in the middle right, points to the constructor 'public SimpleView(Context context)'. Callout 3, on the right side, points to the line 'this.mDrawable = new ShapeDrawable(new RectShape());'.

First, we need to import the necessary packages, including graphics. Then we import `ShapeDrawable`, which will support adding shapes to our drawing, and then `shapes`, which supports several generic shapes (including `RectShape`) that we'll use.

Next, we need to create ① and then set up a `View` ②. After this, we create a `new ShapeDrawable` to add our `Drawable` to ③. After we have a `ShapeDrawable`, we can assign shapes to it. In the code, we use the `RectShape`, but we could've used `OvalShape`, `PathShape`, `RectShape`, `RoundRectShape`, or `Shape`. We then use the `onDraw()` method to draw the `Drawable` on the `Canvas`. Finally, we use the `Drawable's setBounds()` method to set the boundary (a rectangle) in which we'll draw our rectangle using the `draw()` method.

When you run [listing 9.1](#), you should see a simple rectangle like the one shown in [figure 9.1](#)(it's red, although you can't see the color in the printed book).

Figure 9.1. A simple rectangle drawn using Android's Graphics API



Another way to do the same thing is through XML. Android allows you to define shapes to draw in an XML resource file.

9.1.1. Drawing with XML

With Android, you can create simple drawings using an XML file approach. You might want to use XML for several reasons. One basic reason is because it's simple to do. Also, it's worth keeping in mind that graphics described by XML can be programmatically changed later, so XML provides a simple way to do initial design that isn't necessarily static.

To create a drawing with XML, create one or more `Drawable` objects, which are defined as XML files in your drawable directory, such as `res/drawable`. The XML to create a simple rectangle looks like this:

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

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">  
    <solid android:color="#FF0000FF"/>  
</shape>
```

With Android XML drawable shapes, the default is a rectangle, but you can choose a different shape by using the `type` tag and selecting the value `oval`, `rectangle`, `line`, or `arc`. To use your XML shape, you need to reference it in a layout, as shown in [listing 9.2](#). The layout resides in `res/layout`.

ARGB color values

Android uses of Alpha, Red, Green, Blue (ARGB) values common in the software industry for defining color values throughout the Android API. In RGB, colors are defined as ints made up of four bytes: red, green, and blue, plus an alpha. Each value can be a number from 0 to 255 that is converted to hexadecimal (hex). The alpha indicates the level of transparency from 0 to 255.

For example, to create a transparent yellow, we might start with an alpha of 50.2% transparency, where the hex value is `0x80`: this is 128, which is 50.2% of 255. To get yellow, we need red plus green. The number 255 in hex for red and green is `FF`. No blue is needed, so its value is `00`. Thus a transparent yellow is `80FFFF00`. This may seem confusing, but numerous ARGB color charts are available that show the hexadecimal values of a multitude of colors.

Listing 9.2. xmllayout.xml

```
<?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">  
  
    <LinearLayout  
        android:orientation="vertical"  
        android:layout_width="fill_parent"  
        android:layout_height="wrap_content">  
  
        <ImageView android:layout_width="fill_parent"  
            android:src="@drawable/redshape" />  
    
```

```
        android:layout_height="50dip"
        android:src="@drawable/simplerectangle" />
    </LinearLayout>
</ScrollView>
```

All you need to do is create a simple `Activity` and place the UI in a `ContentView`, as follows:

```
public class XMLDraw extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.xmldrawable);
    }
}
```

If you run this code, it draws a simple rectangle. You can make more complex drawings or shapes by stacking or ordering XML drawables, and you can include as many shapes as you want or need, depending on space. Let's explore what multiple shapes might look like next.

9.1.2. Exploring XML drawable shapes

One way to draw multiple shapes with XML is to create multiple XML files that represent different shapes. A simple way to do this is to change the `xml drawable.xml` file to look like the following listing, which adds a number of shapes and stacks them vertically.

Listing 9.3. xml drawable.xml

```
<?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">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
```

```

        android:layout_height="wrap_content">

<ImageView android:layout_width="fill_parent"
           android:layout_height="50dip"
           android:src="@drawable/shape_1" />

<ImageView android:layout_width="fill_parent"
           android:layout_height="wrap_content"
           android:src="@drawable/shape_2" />

<ImageView
           android:layout_width="fill_parent"
           android:layout_height="50dip"
           android:src="@drawable/shape_3" />

<ImageView
           android:layout_width="fill_parent"
           android:layout_height="50dip"
           android:src="@drawable/shape_4" />

</LinearLayout>

</ScrollView>

```

Try adding any of the shapes shown in the following code snippets into the res/drawable folder. You can sequentially name the files shape_n.xml, where n is some number. Or you can give the files any acceptable name as long as the XML file defining the shape is referenced in the xmldrawable.xml file.

In the following code, we're creating a rectangle with rounded corners. We've added a tag called padding, which allows us to define padding or space between the object and other objects in the UI:

```

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

<shape xmlns:android="http://schemas.android.com/apk/res/android"
       type="oval" >

    <solid android:color="#00000000"/>

    <padding android:left="10sp" android:top="4sp"
             android:right="10sp" android:bottom="4sp" />

```

```
<stroke android:width="1dp" android:color="#FFFFFF"/>  
</shape>
```

We're also using the `stroke` tag, which allows us to define the style of the line that makes up the border of the oval, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>  
  
<shape xmlns:android="http://schemas.android.com/apk/res/android">  
  
    <solid android:color="#FF0000FF"/>  
  
    <stroke android:width="4dp" android:color="#FFFFFF"  
            android:dashWidth="1dp" android:dashGap="2dp" />  
  
    <padding android:left="7dp" android:top="7dp"  
             android:right="7dp" android:bottom="7dp" />  
  
    <corners android:radius="4dp" />  
  
</shape>
```

The next snippet introduces the `corners` tag, which allows us to make rounded corners with the attribute `android:radius`:

```
<?xml version="1.0" encoding="utf-8"?>  
  
<shape xmlns:android="http://schemas.android.com/apk/res/android"  
      type="oval">  
  
    <gradient android:startColor="#FFFF0000" android:endColor="#80FF00FF"  
              android:angle="270"/>  
  
    <padding android:left="7dp" android:top="7dp"  
             android:right="7dp" android:bottom="7dp" />  
  
    <corners android:radius="8dp" />  
  
</shape>
```

Finally, we create a shape of the type `line` with a `size` tag using the `android:height` attribute, which allows us to describe the number of pixels used on the vertical to size the line:

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

```
type="line" >

<solid android:color="#FFFFFF"/>

<stroke android:width="1dp" android:color="#FFFFFF"
        android:dashWidth="1dp" android:dashGap="2dp" />

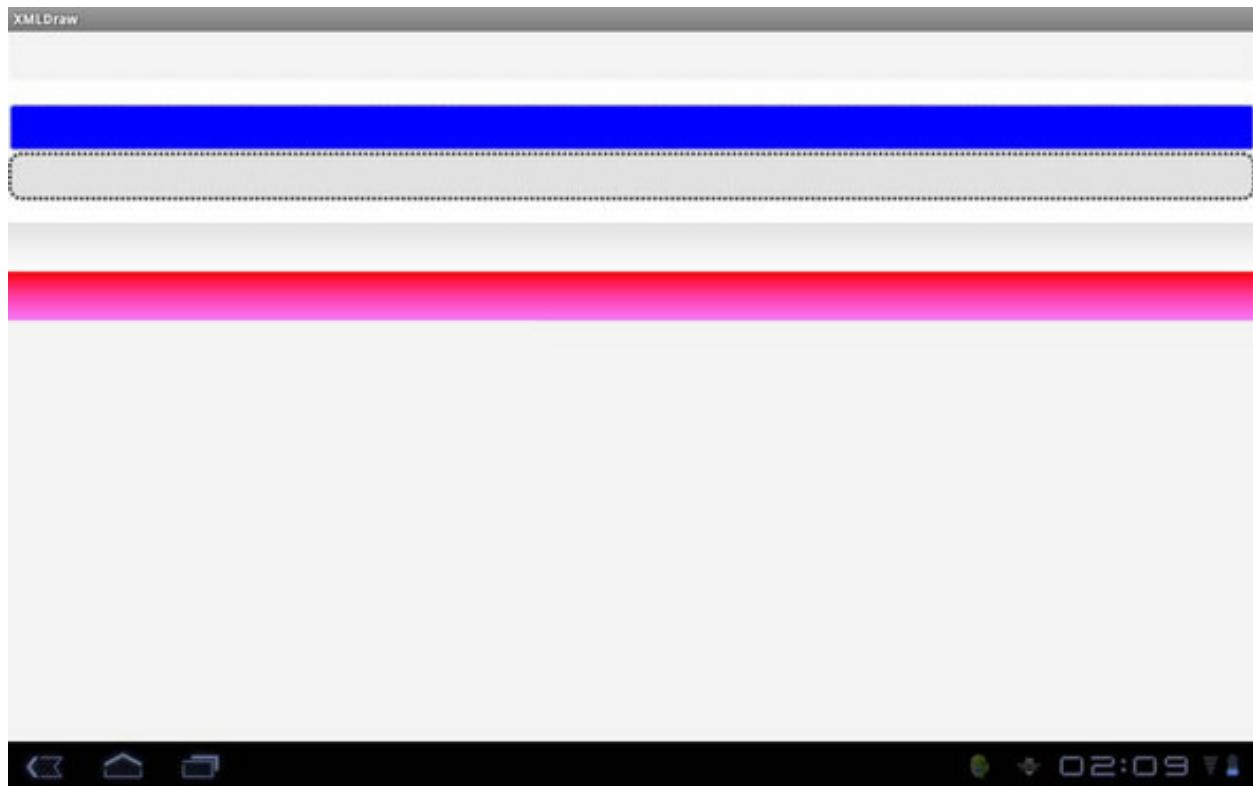
<padding android:left="1dp" android:top="25dp"
          android:right="1dp" android:bottom="25dp" />

<size android:height="23dp" />

</shape>
```

If you run this code, you should see something like [figure 9.2](#).

Figure 9.2. Various shapes drawn using XML



As you can see, Android provides the ability for developers to programmatically draw anything they need. In the next section, we'll look at what you can draw with Android's animation capabilities.

9.2. CREATING ANIMATIONS WITH ANDROID'S GRAPHICS API

If a picture says a thousand words, then an animation must speak volumes. Android supports multiple methods of creating animation, including through XML, as you saw in [chapter 3](#); via Android's XML frame-by-frame animations using the Android Graphics API; and via Android's support for OpenGL ES. In this section, you'll create a simple animation of a bouncing ball using Android's frame-by-frame animation.

9.2.1. Android's frame-by-frame animation

Android allows you to create simple animations by showing a set of images one after another to give the illusion of movement, much like stop-motion film. Android sets each frame image as a drawable resource; the images are then shown one after the other in the background of a `View`. To use this feature, you define a set of resources in an XML file and then call `AnimationDrawable.start()`.

To demonstrate this method for creating an animation, you need to download this project from the Manning website (www.manning.com/ableson3) so you'll have the images. The images for this exercise are six representations of a ball bouncing. Next, create a project called `XMLanimation`, and create a new directory called `/anim` under the `/res` resources directory. Place all the images for this example in `res/drawable`. Then, create an XML file called `Simple_animation.xml` that contains the code shown in the following listing.

Listing 9.4. Simple_animation.xml

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

<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    id="selected" android:oneshot="false">

    <item android:drawable="@drawable/ball1" android:duration="50" />
    <item android:drawable="@drawable/ball2" android:duration="50" />
    <item android:drawable="@drawable/ball3" android:duration="50" />
    <item android:drawable="@drawable/ball4" android:duration="50" />
    <item android:drawable="@drawable/ball5" android:duration="50" />
    <item android:drawable="@drawable/ball6" android:duration="50" />
</animation-list>
```

The XML file defines the list of images to be displayed for the animation. The XML `<animation-list>` tag contains the tags for two attributes: `drawable`, which describes the path to the image, and `duration`, which describes the length of time to show the image, in nanoseconds.

Now, edit the main.xml file to look like the following listing.

Listing 9.5. 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="fill_parent"
    android:layout_height="fill_parent"
    >

    <ImageView android:id="@+id/simple_anim"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:layout_centerHorizontal="true"
        />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Hello World, XMLAnimation"
        />

```

All we've done to the file is added an `ImageView` tag that sets up the layout for the `ImageView`. Finally, create the code to run the animation, as follows.

Listing 9.6. xmlanimation.java

```

public class XMLAnimation extends Activity
{
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        ImageView img =
            (ImageView) findViewById(R.id.simple_anim);
        img.setBackgroundResource(R.anim.simple_animation);
        MyAnimationRoutine mar =
            new MyAnimationRoutine();
        MyAnimationRoutine2 mar2 =
            new MyAnimationRoutine2();
        Timer t = new Timer(false);
        t.schedule(mar, 100);
        Timer t2 = new Timer(false);
        t2.schedule(mar2, 5000);

    }
    class MyAnimationRoutine extends TimerTask {
        @Override
        public void run() {
            ImageView img = (ImageView) findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
                img.getBackground();
            frameAnimation.start();
        }
    }
    class MyAnimationRoutine2 extends TimerTask {
        @Override
        public void run() {
            ImageView img = (ImageView) findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
                img.getBackground();
            frameAnimation.stop();
        }
    }
}

```

Bind resources to ImageView

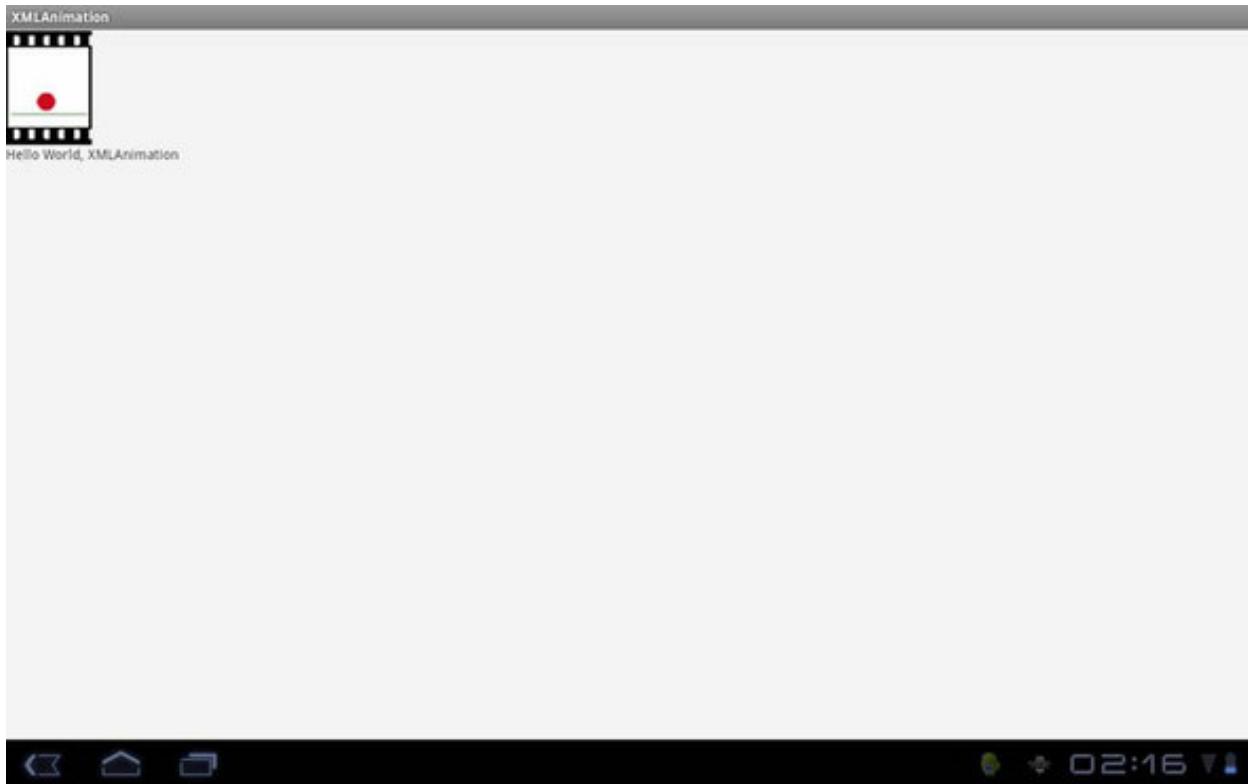
Call subclasses to start and stop animation

1 Allow wait time before starting animation

Listing 9.6 may be slightly confusing because we've used the `TimerTask` classes. Because we can't control the animation from within the `OnCreate()` method, we need to create two such subclasses to call `AnimationDrawable's start()` and `stop()` methods,

respectively. The first subclass, `MyAnimationRoutine`, extends `TimerTask` 1 and calls the `frameAnimation.start()` method for the `AnimationDrawable` bound to the `ImageView` background. If you run the project now, you should see something like figure 9.3.

Figure 9.3. Making a ball bounce using an Android XML animation



As you can see, creating an `Animation` with XML in Android is pretty simple. You can make animations that are reasonably complex, as you would with any stop-motion-type movie; but to create more sophisticated animations programmatically, you need to use Android's 2D and 3D graphics abilities. In the next section, we'll show you how to do just that.

9.2.2. Programmatically creating an animation

In the previous section, you used Android's frame-by-frame animation capabilities to show a series of images in a loop that gives the impression of movement. In this section, you'll programmatically animate a globe so that it moves around the screen.

To create this animation, you'll animate a graphics file (a PNG file) with a ball that appears to be bouncing around inside the Android viewing window. You'll create a `Thread` in which the animation will run and a `Handler` that will help communicate back to the program messages that reflect the changes in the state of the animation. You'll use this same approach in [section 9.3](#) when we talk about OpenGL ES. You'll find that this approach is useful for creating most complex graphics applications and animations.

Creating the Project

This example's animation technique uses an image bound to a sprite. In general, *sprite* refers to a two-dimensional image or animation that is overlaid onto a background or more complex graphical display. For this example, you'll move the sprite around the screen to give the appearance of a bouncing ball. To get started, create a new project called BouncingBall with a `BounceActivity`. You can copy and paste the code in the following listing for the `BounceActivity.java` file.

Listing 9.7. BounceActivity.java

```
public class BounceActivity extends Activity {  
    protected static final int GUIUPDATEIDENTIFIER = 0x101;           ← 1 Create unique identifier  
    Thread myRefreshThread = null;  
    BounceView myBounceView = null;  
    Handler myGUIUpdateHandler = new Handler() {           ← 2 Create handler  
        public void handleMessage(Message msg) {  
            switch (msg.what) {  
                case BounceActivity.GUIUPDATEIDENTIFIER:  
                    myBounceView.invalidate();  
                    break;  
            }  
            super.handleMessage(msg);  
        }  
    };  
  
    @Override  
    public void onCreate(Bundle icicle) {  
        super.onCreate(icicle);  
        this.requestWindowFeature(Window.FEATURE_NO_TITLE);  
        this.myBounceView = new BounceView(this);           ← 3 Create view  
        this.setContentView(this.myBounceView);  
        new Thread(new RefreshRunner()).start();  
    }  
    class RefreshRunner implements Runnable {  
        public void run() {           ← 4 Run animation  
            while (!Thread.currentThread().isInterrupted()) {  
                Message message = new Message();  
                message.what = BounceActivity.GUIUPDATEIDENTIFIER;  
                BounceActivity.this.myGUIUpdateHandler  
.sendMessage(message);  
                try {  
                    Thread.sleep(100);  
                } catch (InterruptedException e) {  
                    Thread.currentThread().interrupt();  
                }  
            }  
        }  
    }  
}
```

First we import the `Handler` and `Message` classes, and then we create a unique identifier to allow us to send a message back to our program to update the `view` in the main thread. We need to send a message telling the main thread to update the `view` each time the child thread has finished drawing the ball. Because different messages can be thrown by the system, we need to guarantee the uniqueness of our message to our handler by creating a unique identifier called `GUIUPDATEIDENTIFIER`. Next, we create the `Handler` that will process our messages to update the main `View`. A `Handler` allows us to send and process `Message` classes and `Runnable` objects associated with a thread's message queue.

Handlers are associated with a single thread and its message queue, but their methods can be called from any thread. Thus we can use the `Handler` to allow objects running in another thread to communicate changes in state back to the thread that spawned them, or vice versa.



Note

For more information about handling long-running requests in your applications, see <http://mng.bz/K0H4>.



We set up a `View` **3** and create the new thread. Finally, we create a `RefreshRunner` inner class implementing `Runnable` that will run unless something interrupts the thread, at which point a message is sent to the `Handler` to call `BounceView`'s `invalidate()` method **4**. The `invalidate()` method invalidates the `View` and forces a refresh.

You've got your new project. Now you need to create the code that will perform the animation and create a `View`.

Making Animation Happen

The example uses an image of a globe, which you can obtain from www.manning.com/ableson3. (Alternatively, you can use any PNG file you'd like.) You'll also have the Android logo as a background; it's included with the source code downloads. Make sure to drop the images into `res/drawable/`.

Next, create a Java file called BounceView, using the code from the following listing.

Listing 9.8. BounceView.java

```
public class BounceView extends View {
    protected Drawable mySprite;
    protected Point mySpritePos = new Point(0,0);
    protected enum HorizontalDirection {LEFT, RIGHT} ;
    protected enum VerticalDirection {UP, DOWN} ;
    protected HorizontalDirection myXDirection =
HorizontalDirection.RIGHT;
    protected VerticalDirection myYDirection = VerticalDirection.UP;
    public BounceView(Context context) {
        super(context);
    this.setBackground(this.getResources().getDrawable(R.drawable.android));
    this.mySprite =
        this.getResources().getDrawable(R.drawable.world); ←
    }
    @Override
    protected void onDraw(Canvas canvas) {
this.mySprite.setBounds(this.mySpritePos.x,
    this.mySpritePos.y,
    this.mySpritePos.x + 50, this.mySpritePos.y + 50);
    if (mySpritePos.x >= this.getWidth() - ←
mySprite.getBounds().width()) {
        this.myXDirection = HorizontalDirection.LEFT;
    } else if (mySpritePos.x <= 0) {
        this.myXDirection = HorizontalDirection.RIGHT;
    }
    if (mySpritePos.y >= this.getHeight() - ←
mySprite.getBounds().height()) {
        this.myYDirection = VerticalDirection.UP;
    } else if (mySpritePos.y <= 0) {
        this.myYDirection = VerticalDirection.DOWN;
    }
    if (this.myXDirection ==
HorizontalDirection.RIGHT) {
        this.mySpritePos.x += 10;
    } else {
        this.mySpritePos.x -= 10;
    }
    if (this.myYDirection ==
        VerticalDirection.DOWN) { ←
        this.mySpritePos.y += 10;
    } else {
        this.mySpritePos.y -= 10;
    }
    this.mySprite.draw(canvas);
}
}
```

1 Get image file and map to sprite

2 Set bounds of globe

3 Move ball left or right, up or down

4 Check if ball is trying to leave screen

In this listing, we do all the real work of animating the image. First, we create a `Drawable` to hold the globe image and a `Point` that we use to position and track the globe as we animate it. Next, we create enumerations (`enums`) to hold directional values for horizontal and vertical directions, which we'll use to keep track of the moving globe. Then we map the globe to the `mSurface` variable and set the Android logo as the background for the animation ①.

Now that we've done the setup work, we create a new `View` and set all the boundaries for the `Drawable` ②. After that, we create simple conditional logic that detects whether the globe is trying to leave the screen; if it starts to leave the screen, we change its direction ③. Then we provide simple conditional logic to keep the ball moving in the same direction if it hasn't encountered the bounds of the `View` ④. Finally, we draw the globe using the `draw()` method.

If you compile and run the project, you should see the globe bouncing around in front of the Android logo, as shown in [figure 9.4](#).

Figure 9.4. Animation of a globe bouncing in front of the Android logo



Although this animation isn't too exciting, you could—with a little extra work—use the key concepts (dealing with boundaries, moving `drawables`, detecting changes, dealing with threads, and so on) to create something like the Google Lunar Lander example game or even a simple version of Asteroids. If you want more graphics power and want to easily work with 3D objects to create things such as games or sophisticated animations, you'll learn how in the next section on OpenGL ES.

9.3. INTRODUCING OPENGL FOR EMBEDDED SYSTEMS

One of the most interesting features of the Android platform is its support of *OpenGL for Embedded Systems* (*OpenGL ES*). OpenGL ES is the embedded systems version of the popular OpenGL standard, which defines a cross-platform and cross-language API for computer graphics. The OpenGL ES API doesn't support the full OpenGL API, and much of the OpenGL API has been stripped out to allow OpenGL ES to run on a variety of mobile phones, PDAs, video game consoles, and other embedded systems. OpenGL ES was originally developed by the Khronos Group, an industry consortium. You can find the most current version of the standard at www.khronos.org/opengles/.

OpenGL ES is a fantastic API for 2D and 3D graphics, especially for graphically intensive applications such as games, graphical simulations, visualizations, and all sorts of animations. Because Android also supports 3D hardware acceleration, developers can make graphically intensive applications that target hardware with 3D accelerators.

Android 2.1 supports the OpenGL ES 1.0 standard, which is almost equivalent to the OpenGL 1.3 standard. If an application can run on a computer using OpenGL 1.3, it should be possible to run it on Android after light modification, but you need to consider the hardware specifications of your Android handset. Although Android offers support for hardware acceleration, some handsets and devices running Android have had performance issues with OpenGL ES in the past. Before you embark on a project using OpenGL, consider the hardware you're targeting and do extensive testing to make sure that you don't overwhelm your hardware with OpenGL graphics.

Because OpenGL and OpenGL ES are such broad topics, with entire books dedicated to them, we'll cover only the basics of working with OpenGL ES and Android. For a much deeper exploration of OpenGL ES, check out the specification and the OpenGL ES tutorial at <http://mng.bz/0tdm>. After reading this section on Android support for OpenGL ES, you should have enough information to follow a more in-depth discussion of OpenGL ES, and you should be able to port your code from other languages (such as the tutorial examples) into the Android framework. If you already know OpenGL or OpenGL ES, then the OpenGL commands will be familiar; concentrate on the specifics of working with OpenGL on Android.

Note

For another good OpenGL resource from Silicon Graphics see www.glprogramming.com/red/index.html.

9.3.1. Creating an OpenGL context

Keeping in mind the comments we made in the introduction to this section, let's apply the basics of OpenGL ES to create an `OpenGLContext` and a `Window` to draw in. Much of this task will seem overly complex compared to Android's Graphics API. The good news is that you have to do this setup work only once.



Note

Much of the material covered here will require further detailed explanation if you aren't already experienced with OpenGL. For more information, we suggest that you refer directly to the documentation from OpenGL at www.opengl.org/.



You'll use the general processes outlined in the following sections to work with OpenGL ES in Android:

1. Create a custom `View` subclass.
2. Get a handle to an `OpenGLContext`, which provides access to Android's OpenGL ES functionality.
3. In the `View`'s `onDraw()` method, use the handle to the GL object and then use its methods to perform any GL functions.

Following these basic steps, first you'll create a class that uses Android to create a blank surface to draw on. In [section 9.3.2](#), you'll use OpenGL ES commands to draw a square and an animated cube on the surface. To start, open a new project called `OpenGLSquare` and create an `Activity` called `OpenGLSquare`, as shown in the following listing.

Listing 9.9. `OpenGLSquare.java`

```
public class SquareActivity extends Activity {  
    @Override  
    public void onCreate(Bundle icicle) {  
        super.onCreate(icicle);  
        setContentView(new DrawingSurfaceView(this));  
    }  
    class DrawingSurfaceView extends SurfaceView implements  
    SurfaceHolder.Callback {  
        public SurfaceHolder mHolder;  
        public DrawingThread mThread;  
        public DrawingSurfaceView(Context c) {  
            super(c);  
            init();  
        }  
        public void init() {  
            mHolder = getHolder();  
            mHolder.addCallback(this);  
            mHolder.setType(SurfaceHolder.SURFACE_TYPE_GPU);  
        }  
        public void surfaceCreated(SurfaceHolder holder) {  
            mThread = new DrawingThread();  
            mThread.start();  
        }  
        public void surfaceDestroyed(SurfaceHolder holder) {  
            mThread.waitForExit();  
            mThread = null;  
        }  
    }  
}
```

1 Handle creation and destruction

2 Do drawing

3 Register as callback

```

public void surfaceChanged(SurfaceHolder holder,
    int format, int w, int h) {
    mThread.onWindowResize(w, h);
}
class DrawingThread extends Thread {
    boolean stop;
    int w;
    int h;
    boolean changed = true;
    DrawingThread() {
        super();
        stop = false;
        w = 0;
        h = 0;
    }
    @Override
    public void run() {
EGL10 egl = (EGL10)EGLContext.getEGL();
        EGLDisplay dpy =
            egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);
        int[] version = new int[2];
        egl.eglInitialize(dpy, version);
        int[] configSpec = {
            EGL10.EGL_RED_SIZE,      5,
            EGL10.EGL_GREEN_SIZE,    6,
            EGL10.EGL_BLUE_SIZE,     5,
            EGL10.EGL_DEPTH_SIZE,   16,
            EGL10.EGL_NONE
        };
        EGLConfig[] configs = new EGLConfig[1];
        int[] num_config = new int[1];
        egl.eglChooseConfig(dpy, configSpec, configs, 1,
num_config);
        EGLConfig config = configs[0];
        EGLContext context = egl.eglCreateContext(dpy,
config, EGL10.EGL_NO_CONTEXT, null);
        EGLSurface surface = null;
        GL10 gl = null;
        while(!stop) {

```

4 Create thread to do drawing

5 Get EGL Instance

6 Specify configuration to use

7 Obtain reference to OpenGL ES context

8 Do drawing

[Listing 9.9](#) generates an empty black screen. Everything in this listing is code you need to draw and manage any OpenGL ES visualization. First, we import all our needed classes. Then we implement an inner class, which will handle everything about managing a surface: creating it, changing it, or deleting it. We extend the class `SurfaceView` and implement the `SurfaceHolder` interface, which allows us to get information back from

Android when the surface changes, such as when someone resizes it 1. With Android, all this has to be done asynchronously; you can't manage surfaces directly.

Next, we create a thread to do the drawing **2** and create an `init()` method that uses the `SurfaceView` class's `getHolder()` method **3** to get access to the `SurfaceView` and add a callback to it via the `addCallBack()` method **4**. Now we can implement `surfaceCreated()`, `surfaceChanged()`, and `surfaceDestroyed()`, which are all methods of the `Callback` class and are fired on the appropriate condition of change in the `Surface`'s state.

When all the `Callback` methods are implemented, we create a thread to do the drawing **5**. Before we can draw anything, though, we need to create an OpenGL ES context **6** and create a handler to the `Surface` **7** so that we can use the OpenGL context's method to act on the surface via the handle **8**. Now we can finally draw something, although in the `drawFrame()` method **9** we aren't doing anything.

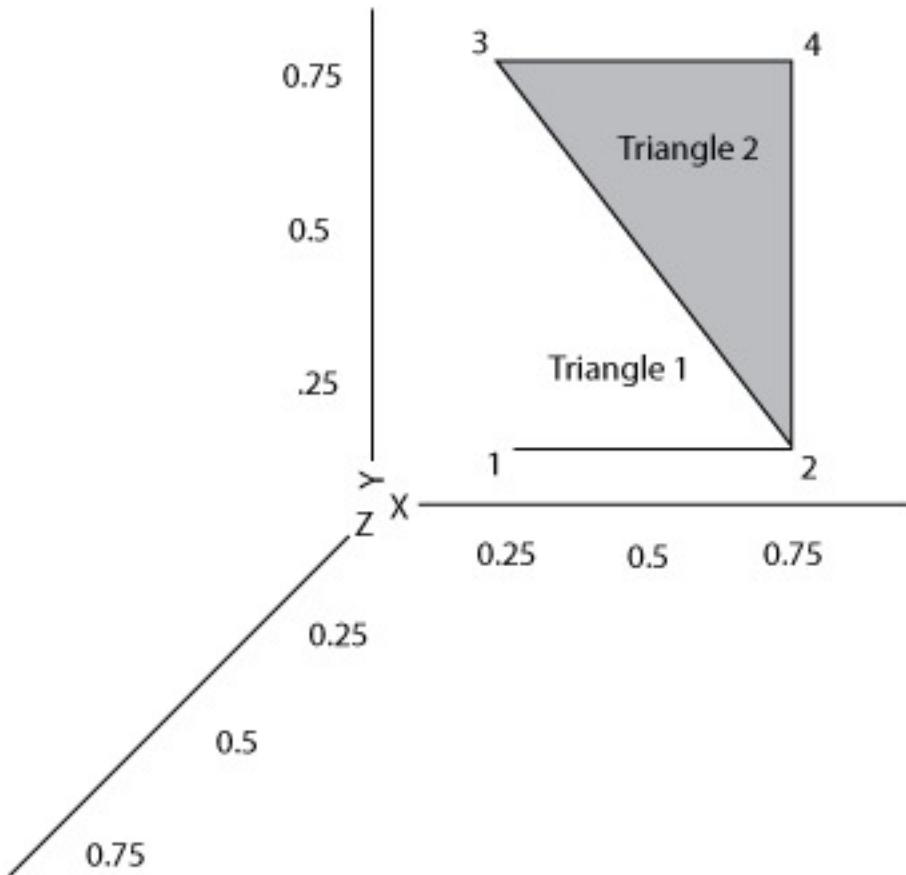
If you were to run the code right now, all you'd get would be an empty window; but what we've generated so far will appear in some form or another in any OpenGL ES application you make on Android. Typically, you'll break up the code so that an `Activity` class starts the code and another class implements the custom `View`. Yet another class may implement your `SurfaceHolder` and `SurfaceHolder.Callback`, providing all the methods for detecting changes to the surface, as well as those for the drawing of your graphics in a thread. Finally, you may have another class for whatever code represents your graphics.

In the next section, we'll look at how to draw a square on the surface and how to create an animated cube.

9.3.2. Drawing a rectangle with OpenGL ES

In the next example, you'll use OpenGL ES to create a simple drawing, a rectangle, using OpenGL primitives, which in OpenGL ES are pixels and triangles. When you draw the square, you'll use a primitive called the `GL_Triangle_Strip`, which takes three vertices (the x, y, and z points in an array of vertices) and draws a triangle. The last two vertices become the first two vertices for the next triangle, with the next vertex in the array being the final point. This process repeats for as many vertices as there are in the array, and it generates something like [figure 9.5](#), where two triangles are drawn.

Figure 9.5. How two triangles are drawn from an array of vertices



OpenGL ES supports a small set of primitives, shown in [table 9.1](#), that allow you to build anything using simple geometric shapes, from a rectangle to 3D models of animated characters.

Table 9.1. OpenGL ES primitives and their descriptions

Primitive flag	Description
GL_LINE_LOOP	Draws a continuous set of lines. After the first vertex, it draws a line between every successive vertex and the vertex before it. Then it connects the start and end vertices.
GL_LINE_STRIP	Draws a continuous set of lines. After the first vertex, it draws a line between every successive vertex and the vertex before it.
GL_LINES	Draws a line for every pair of vertices given.
GL_POINTS	Places a point at each vertex.
GL_TRIANGLE_FAN	After the first two vertices, every successive vertex uses the previous vertex and the first vertex to draw a triangle. This flag is used to draw cone-like shapes.
GL_TRIANGLE_STRIP	After the first two vertices, every successive vertex uses the previous two vertices to draw the next triangle.

Primitive flag	Description
GL_TRIANGLES	For every triplet of vertices, it draws a triangle with corners specified by the coordinates of the vertices.

In the next listing, we use an array of vertices to define a square to paint on our surface. To use the code, insert it directly into the code for [listing 9.9](#), immediately below the commented line // do whatever drawing here.

Listing 9.10. OpenGLSquare.java

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT |  
    GL10.GL_DEPTH_BUFFER_BIT);  
float[] square = new float[] {  
    0.25f, 0.25f, 0.0f,  
    0.75f, 0.25f, 0.0f,  
    0.25f, 0.75f, 0.0f,  
    0.75f, 0.75f, 0.0f };  
FloatBuffer squareBuff;  
ByteBuffer bb =  
    ByteBuffer.allocateDirect(square.length*4);  
    bb.order(ByteOrder.nativeOrder());  
    squareBuff = bb.asFloatBuffer();  
    squareBuff.put(square);  
    squareBuff.position(0);  
    gl.glMatrixMode(GL10.GL_PROJECTION);  
    gl.glLoadIdentity();  
    GLU.gluOrtho2D(gl, 0.0f, 1.2f, 0.0f, 1.0f);  
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, squareBuff);  
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);  
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);  
    gl glColor4f(0,1,1,1);  
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);
```

1 Create float buffer to hold square
2 Set up 2D orthographic viewing region
3 Set current vertices for drawing

This code is dense with OpenGL commands. The first thing we do is clear the screen using `glClear`, which you want to do before every drawing. Then we build the array to represent the set of vertices that make up our square. As we explained, we use the OpenGL primitive `GL_TRIANGLE_STRIP` to create the rectangle shown in [figure 9.5](#), where the first set of three vertices (points 1, 2, and 3) represent the first triangle. The last vertex represents the third vertex (point 4) in the second triangle, which reuses vertices 2 and 3 from the first triangle as its first two to make the triangle described by points 2, 3, and 4. To put it more succinctly, Open GL ES takes one triangle and flips it over on its third side (in this case, the hypotenuse). We then create a buffer to hold that same

square data 1. We also tell the system that we'll be using a `GL_PROJECTION` for our matrix mode, which is a type of matrix transformation that's applied to every point in the matrix stack.

The next things we do are more related to setup. We load the identity matrix and then use the `gluOrtho2D(GL10 gl, float left, float right, float bottom, float top)` command to set the clipping planes that are mapped to the lower-left and upper-right corners of the window .

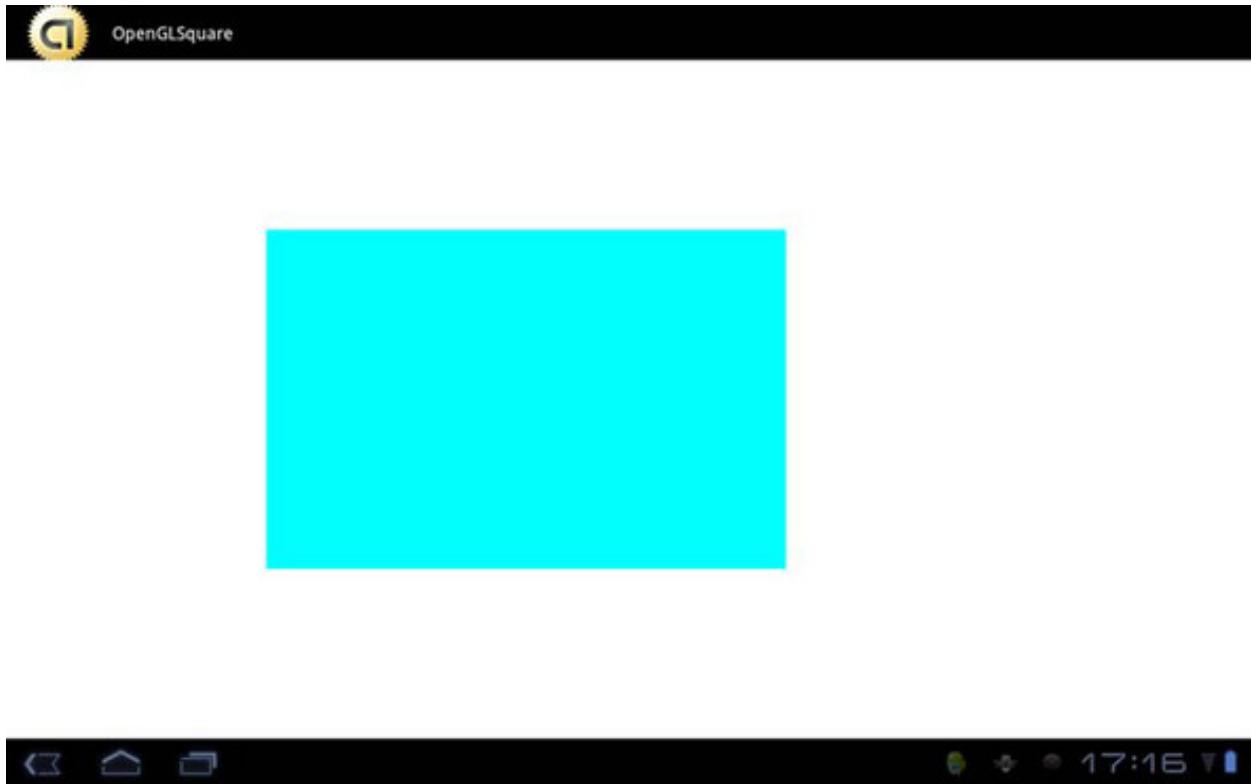
Now we're ready to start drawing the image. First, we use the `glVertexPointer(int size, int type, int stride, pointer to array)` method, which indicates the location of vertices for the triangle strip. The method has four attributes: `size`, `type`, `stride`, and `pointer`. The `size` attribute specifies the number of coordinates per vertex (for example, a 2D shape might ignore the z axis and use only two coordinates per vertex), `type` defines the data type to be used (`GL_BYTE`, `GL_SHORT`, `GL_FLOAT`, and so

on) , `stride` specifies the offset between consecutive vertices (how many unused values exist between the end of the current vertex and the beginning of the next), and `pointer` is a reference to the array. Although most drawing in OpenGL ES is performed by using various forms of arrays such as the vertex array, they're all disabled by default to save system resources. To enable them, we use the OpenGL command `glEnableClientState(array type)`, which accepts an array type; in this case, the type is `GL_VERTEX_ARRAY`.

Finally, we use the `glDrawArrays` function to render our arrays into the OpenGL primitives and create our simple drawing. The `glDrawArrays(mode, first, count)` function has three attributes: `mode` indicates which primitive to render, such as `GL_TRIANGLE_STRIP`; `first` is the starting index into the array, which we set to `0` because we want it to render all the vertices in the array; and `count` specifies the number of indices to be rendered, which in this case is `4`.

If you run the code, you should see a simple blue rectangle on a white surface, as shown in [figure 9.6](#). It isn't particularly exciting, but you'll need most of the code you used for this example for any OpenGL project.

Figure 9.6. A rectangle drawn on the surface using OpenGL ES



There you have it—your first graphic in OpenGL ES. Next, we’re going to do something way more interesting. In the next example, you’ll create a 3D cube with different colors on each side and then rotate it in space.

9.3.3. Three-dimensional shapes and surfaces with OpenGL ES

In this section, you’ll use much of the code from the previous example, but you’ll extend it to create a 3D cube that rotates. We’ll examine how to introduce perspective to your graphics to give the illusion of depth.

Depth works in OpenGL by using a *depth buffer*, which contains a depth value for every pixel, in the range 0 to 1. The value represents the perceived distance between objects and your viewpoint; when two objects’ depth values are compared, the value closer to 0 will appear in front on the screen. To use depth in your program, you need to first enable the depth buffer by passing `GL_DEPTH_TEST` to the `glEnable()` method. Next, you use `glDepthFunc()` to define how values are compared. For this example, you’ll use `GL_LESS`, defined in [table 9.2](#), which tells the system to show objects with a lower depth value in front of other objects.

Table 9.2. Flags for determining how values in the depth buffer are compared

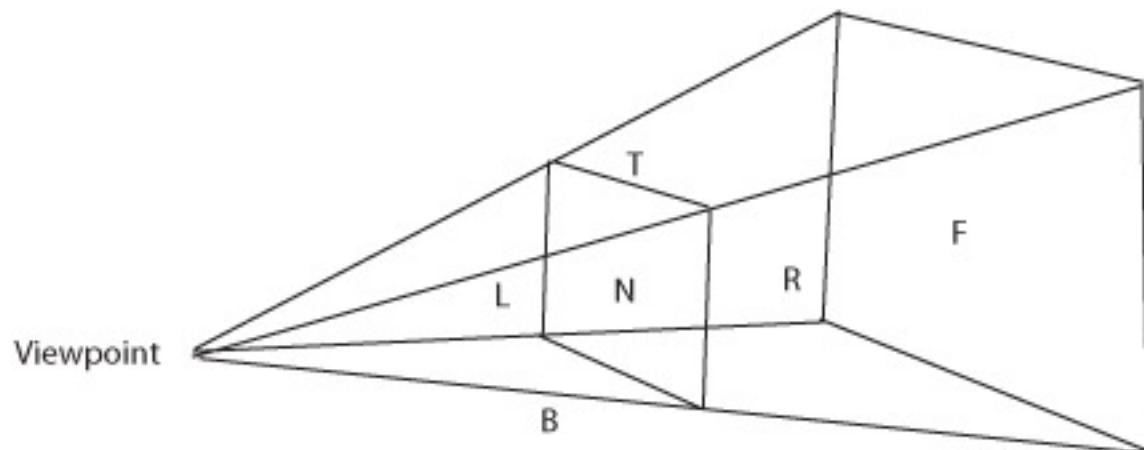
Flag	Description
GL_ALWAYS	Always passes
GL_EQUAL	Passes if the incoming depth value is equal to the stored value
GL_GEQUAL	Passes if the incoming depth value is greater than or equal to the stored value
GL_GREATER	Passes if the incoming depth value is greater than the stored value
GL_LEQUAL	Passes if the incoming depth value is less than or equal to the stored value
GL_LESS	Passes if the incoming depth value is less than the stored value
GL_NEVER	Never passes
GL_NOTEQUAL	Passes if the incoming depth value isn't equal to the stored value

When you draw a primitive, the depth test occurs. If the value passes the test, the incoming color value replaces the current one.

The default value is `GL_LESS`. You want the value to pass the test if the values are equal as well. Objects with the same z value will display, depending on the order in which they were drawn. We pass `GL_LEQUAL` to the function.

One important part of maintaining the illusion of depth is providing perspective. In OpenGL, a typical perspective is represented by a viewpoint with near and far clipping planes and top, bottom, left, and right planes, where objects that are closer to the far plane appear smaller, as in [figure 9.7](#).

Figure 9.7. In OpenGL, a perspective is made up of a viewpoint and near (N), far (F), left (L), right (R), top (T), and bottom (B) clipping planes.



OpenGL ES provides a function called `gluPerspective(GL10 gl, float fovy, float aspect, float zNear, float zFar)` with five parameters (see [table 9.3](#)) that lets you easily create perspective.

Table 9.3. Parameters for the `gluPerspective` function

Parameter Description	
aspect	Aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).
fovy	Field of view angle in the y direction, in degrees.
gl	GL10 interface.
zFar	Distance from the viewer to the far clipping plane. This value is always positive.
zNear	Distance from the viewer to the near clipping plane. This value is always positive.

To demonstrate depth and perspective, you’re going to create a project called `OpenGLCube`. Copy and paste the code from [listing 9.11](#) into `OpenGLCubeActivity`.

Now add two new variables to your code, shown in the following listing, right at the beginning of the `DrawSurfaceView` inner class.

Listing 9.11. OpenGLCubeActivity.java

```
class DrawingSurfaceView extends SurfaceView implements  
SurfaceHolder.Callback {  
  
    public SurfaceHolder mHolder;  
  
    float xrot = 0.0f;  
  
    float yrot = 0.0f;
```

We’ll use the `xrot` and `yrot` variables later in the code to govern the rotation of the cube.

Next, just before the method, add a new method called `makeFloatBuffer()`, as in the following listing.

Listing 9.12. OpenGLCubeActivity.java

```
protected FloatBuffer makeFloatBuffer(float[] arr) {  
  
    ByteBuffer bb = ByteBuffer.allocateDirect(arr.length*4);  
  
    bb.order(ByteOrder.nativeOrder());
```

```
    FloatBuffer fb = bb.asFloatBuffer();  
  
    fb.put(arr);  
  
    fb.position(0);  
  
    return fb;  
  
}
```

This float buffer is the same as the one in [listing 9.11](#), but we've abstracted it from the `drawFrame()` method so we can focus on the code for rendering and animating the cube.

Next, copy and paste the code from the following listing into the `drawFrame()` method. Note that you'll also need to update your `drawFrame()` call in the following way:

```
drawFrame(gl, w, h);
```

Listing 9.13. OpenGLCubeActivity.java

```
private void drawFrame(GL10 gl, int w1, int h1) {  
    float mycube[] = {  
        // FRONT  
        -0.5f, -0.5f,  0.5f,  
        0.5f, -0.5f,  0.5f,  
        -0.5f,  0.5f,  0.5f,  
        0.5f,  0.5f,  0.5f,  
        // BACK  
        -0.5f, -0.5f, -0.5f,  
        -0.5f,  0.5f, -0.5f,  
        0.5f, -0.5f, -0.5f,  
        0.5f,  0.5f, -0.5f,  
        // LEFT  
        -0.5f, -0.5f,  0.5f,  
        -0.5f,  0.5f,  0.5f,  
        -0.5f, -0.5f, -0.5f,  
        -0.5f,  0.5f, -0.5f,  
        // RIGHT  
        0.5f, -0.5f, -0.5f,  
        0.5f,  0.5f, -0.5f,  
        0.5f, -0.5f,  0.5f,  
        0.5f,  0.5f,  0.5f,  
        // TOP  
        -0.5f,  0.5f,  0.5f,  
        0.5f,  0.5f,  0.5f,  
        -0.5f,  0.5f, -0.5f,  
        0.5f,  0.5f, -0.5f,  
        // BOTTOM  
        -0.5f, -0.5f,  0.5f,  
        -0.5f, -0.5f, -0.5f,  
        0.5f, -0.5f,  0.5f,  
        0.5f, -0.5f, -0.5f,
```

```

};

FloatBuffer cubeBuff;
cubeBuff = makeFloatBuffer(mycube);
gl.glEnable(GL10.GL_DEPTH_TEST);
gl.glEnable(GL10.GL_CULL_FACE);
gl.glDepthFunc(GL10.GL_EQUAL);
gl.glClearDepthf(1.0f);
gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
GL10.GL_DEPTH_BUFFER_BIT);
gl.glMatrixMode(GL10.GL_PROJECTION);
gl.glLoadIdentity();
gl.glViewport(0,0,w,h);
GLU.gluPerspective(gl, 45.0f,
((float)w)/h, 1f, 100f);
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glLoadIdentity();
GLU.gluLookAt(gl, 0, 0, 3, 0, 0, 0, 0, 1, 0);
gl.glShadeModel(GL10.GL_SMOOTH);
gl.glVertexPointer(3, GL10.GL_FLOAT, 0, cubeBuff);
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glRotatef(xrot, 1, 0, 0);
gl.glRotatef(yrot, 0, 1, 0);
gl glColor4f(1.0f, 0, 0, 1.0f);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 4, 4);
gl glColor4f(0, 1.0f, 0, 1.0f);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 8, 4);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 12, 4);
gl glColor4f(0, 0, 1.0f, 1.0f);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 16, 4);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 20, 4);
xrot += 1.0f;
yrot += 0.5f;

```

1 Create float buffer for vertices

2 Enable depth test

3 Define perspective

4 Draw six sides in three colors

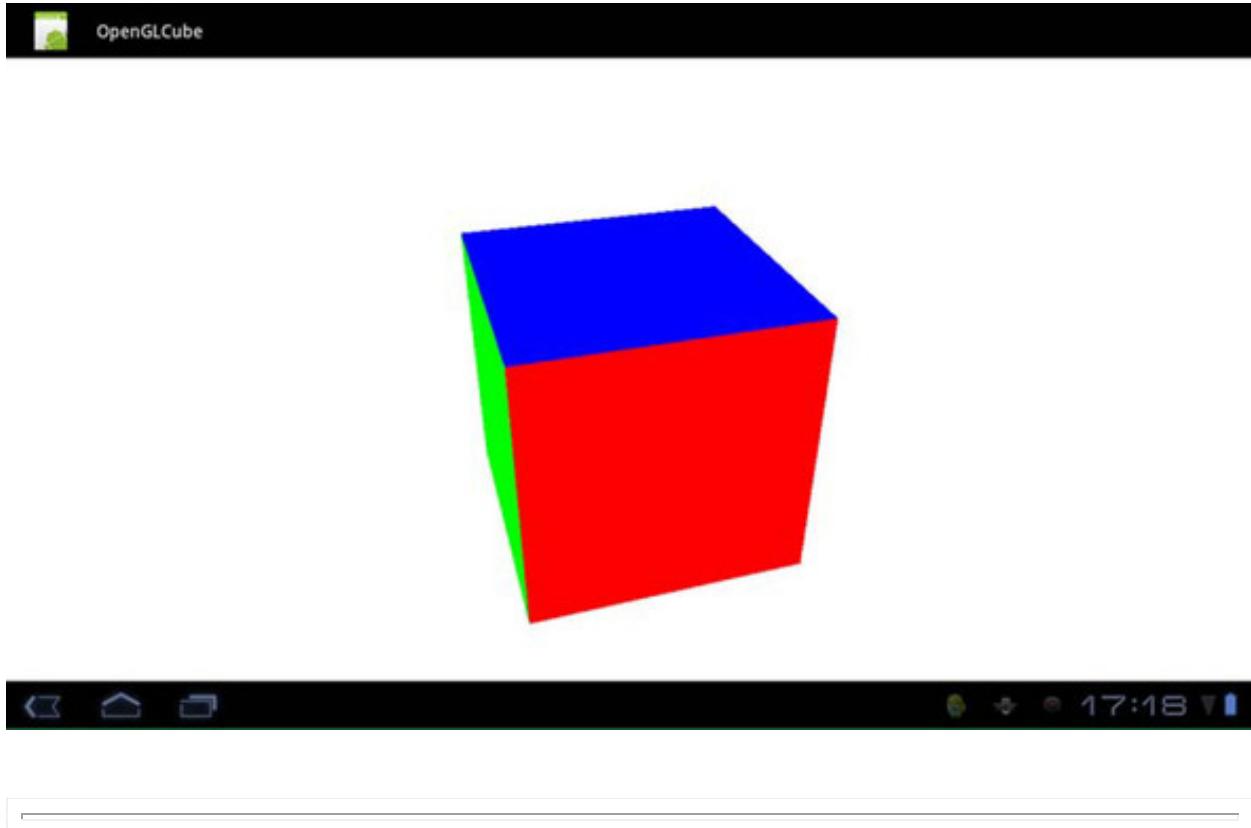
Increment x and y rotations

This listing doesn't contain much new code. First, we describe the vertices for a cube, which is built the same way as the rectangle in [listing 9.10](#) (using triangles). Next, we set up the float buffer for our vertices ① and enable the depth function ② and perspective function ③ to provide a sense of depth. Note that with `gluPerspective` we passed `45.0f` (45 degrees) to give a more natural viewpoint.

Next, we use the `GLU.gluLookAt(GL10 gl, float eyeX, float eyeY, float eyeZ, float centerX, float centerY, float centerZ, float upX, float upY, float upZ)` function to move the position of the view without having to modify the projection matrix directly. When we've established the view position, we turn on smooth shading for the model and rotate the cube around the x and y axes. Then we draw the cube sides and increment the rotation so that on the next iteration of `draw()`, the cube is drawn at a

slightly different angle 4. If you run the code, you should see a rotating 3D cube like the one shown in figure 9.8.

Figure 9.8. A 3D cube rotating in space



Note

You can try experimenting with the `fovy` value to see how changing the angle affects the display of the cube.

You've done a lot in this section, starting with creating an OpenGL ES context in which you can develop your OpenGL ES applications. Next, you learned how to build shapes using OpenGL ES by “triangulation” (creating multiple triangles). Then, you learned how to realize this in three dimensions while incorporating it into a simple example. You accomplished much of this without diving deep into OpenGL ES, which is definitely nontrivial, but the good news is that if you're serious about doing 3D graphics on Android, it's definitely possible.

With the addition of RenderScript, introduced in the next section of this chapter, developers can write code that is designed to use native code on specific hardware, allowing for much better performance of applications that are heavily dependent on processing power (such as Open GL applications). Because Android provides excellent support for OpenGL ES, you can find plenty of tutorials and references on the internet or at your local bookstore.

Now, let's look at how to use RenderScript to develop complex, rich, and high-performance graphical application that let you take advantage of the latest mobile hardware platforms that run multicore processors with dedicated graphics accelerators.

9.4. INTRODUCING RENDERSCRIPT FOR ANDROID

RenderScript is a new API in Android that is focused on helping developers who need extremely high performance for graphics and computationally intensive operations. RenderScript isn't completely new to Android 3.0+; it's been part of earlier versions in 2.0 but not publicly available. As of Android 3, RenderScript has come to the fore as the tool of choice for graphically intensive games and applications such as live wallpapers, the new video carousel, and Google's e-book reader on the Xoom. In this section, we'll look at how RenderScript fits into the Android architecture, how to build a RenderScript application, and when and where to use RenderScript.

RenderScript in many ways is a new paradigm for the Android platform. Although Android uses Java syntax and a virtual machine for developing applications, RenderScript is based on C99, a modern dialect of the C language. Furthermore, RenderScript is compiled down to native code on each device at runtime but is controlled by higher-level APIs running in the Android VM. This allows Android via RenderScript to provide developers a way to develop optimized high-performance code that is cross platform. This may seem extremely attractive, and many developers may be keen to write most of their applications in RenderScript, but RenderScript doesn't replace or subsume development of Android apps in Java. There are both pros and cons to working with RenderScript.

9.4.1. RenderScript advantages and disadvantages

As already discussed, the first advantage of using RenderScript is that it's a lower-level language offering higher performance. Second, it allows Android apps to more easily use multicore CPUs as well as graphical processing units (GPUs). RenderScript, by design, at runtime selects the best-performance approach to running its code. This includes running the code across multiple CPUs; running some simpler tasks on GPUs; or, in some cases where no special hardware is present, running on just one CPU.

RenderScript offers fantastic performance and cross-platform compatibility without the need to target specific devices or create your own complex architectures for cross-platform compatibility. RenderScript is best for two types of applications and only has APIs to support those two types of applications: graphical applications and computationally intensive applications. Many applications that use Android's implementation of OpenGL are good candidates to target for RenderScript.

The first major drawback of RenderScript is that it uses C99. Although there is nothing wrong with C99, it breaks the Java style paradigm that most Android developers are comfortable with. To be truly comfortable developing RenderScript applications, you should also be comfortable with C, a lower-level language when compared to Java.

Second, and perhaps most important, RenderScript applications are inherently more complex and difficult to develop than regular Android applications. In part this is because you're developing in two different languages, Java and C; but in addition, RenderScript by its nature is very hard to debug—at times frustratingly so, unless you have a strong understanding of both your application and the hardware it's running on. For example, if you have a multicore platform with a GPU, your code may be run on either the CPUs or the GPU, reducing your ability to spot issues. Also be aware that most RenderScript applications won't run in the emulator, forcing you to debug on hardware as well.

Finally, you'll find that you have a lot more bugs, because RenderScript is in C, the current Android Development Tools (ADT) application for Eclipses doesn't support the various extensions for it, and RenderScript applications tend to be more complex than regular Android applications. But you shouldn't avoid developing in RenderScript, nor should you overuse it as opposed to the standard Android APIs and Java syntax. Rather, you should look to use RenderScript in applications that are graphically intensive or computationally intensive.

Let's try building a RenderScript application.

9.4.2. Building a RenderScript application

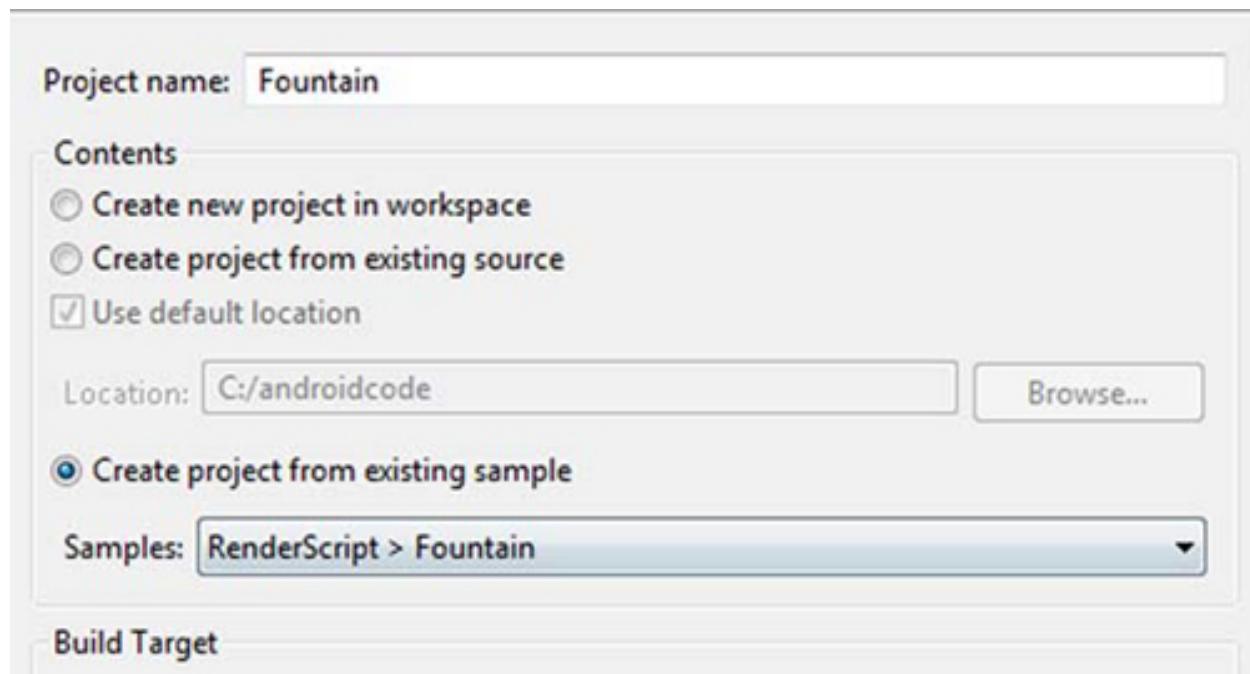
Building a RenderScript application is a bit more complicated than developing a normal Android application. You lay out your application in a similar manner, but keep in mind that you'll be also developing RenderScript files, with the .rs file extension, alongside your .java files. Your normal .java application files then call the RenderScript code as needed; when you build your project, you'll see the .rs files built into bytecode with the same name as the RenderScript file but with the .bc extension under the raw folder. For example, if you had a RenderScript file called Helloworld.rs under src, you'd see a Helloworld.bc file when your application was built.

Note

We won't be covering the C or C99 language; we assume you know C. If you don't know C, you'll need to reference another resource such as Manning's *C# in Depth*, 2nd edition, by John Skeet.

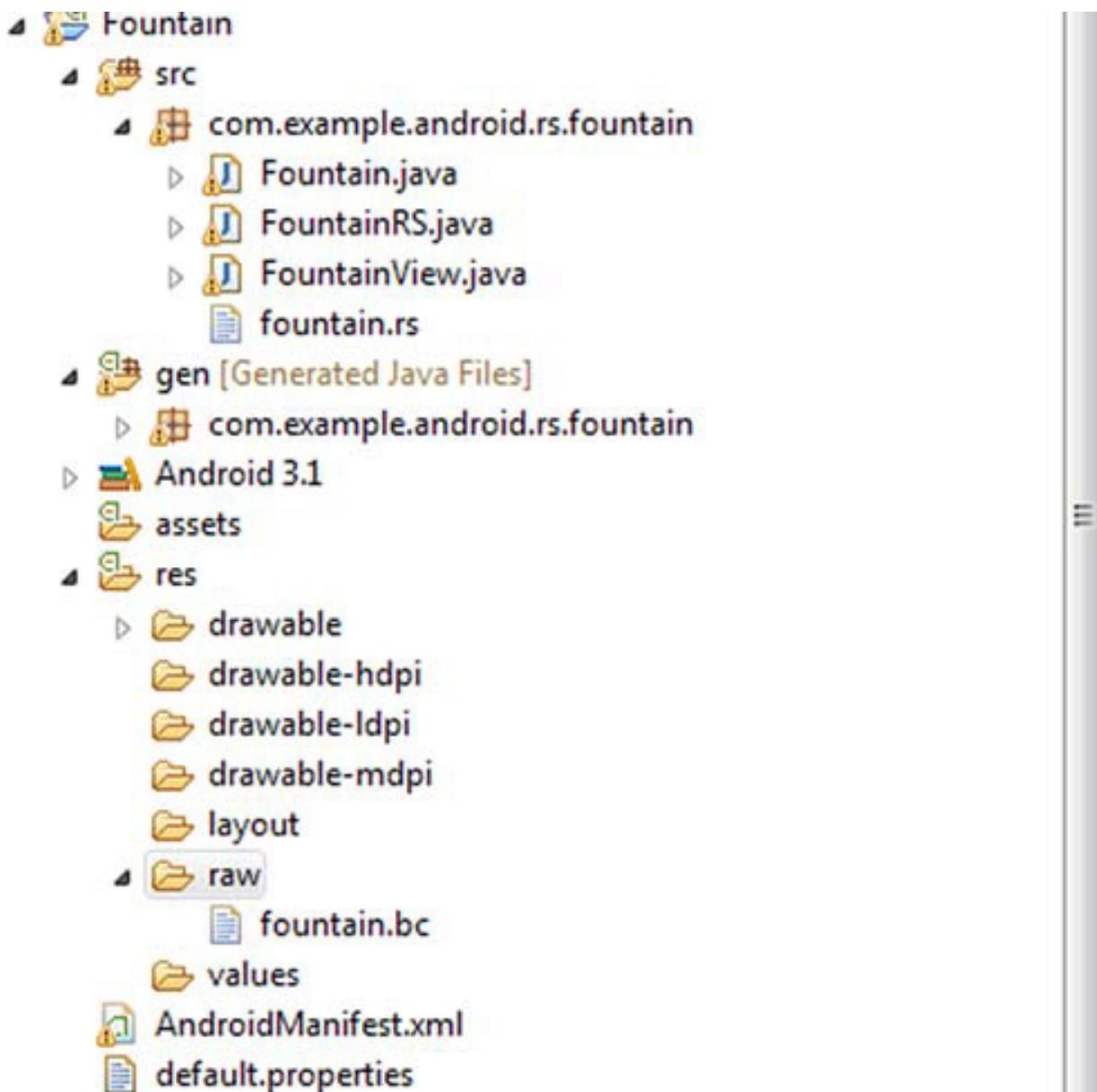
For your RenderScript application, you're going to use the ADT's built-in Android project wizard to create a RenderScript project from built-in sample applications. To do so, first create a new project using the ADT, but instead of selecting Create New Project in Workspace, select Create Project from Existing Sample, as shown in [figure 9.9](#). Make sure you've selected API level of 11 or Android 3.0, and select the sample RenderScript > Fountain from the Samples drop-down list. Click OK.

Figure 9.9. Using the ADT to build a sample RenderScript application



Eclipse now builds the RenderScript application. Expand the application in the Eclipse Package Explorer, as shown in [figure 9.10](#). There are several things to note here before we go over each file. First, note the RenderScript file with the extension .rs. This is a file written in C. This file does all the real graphics work, and the other .java files provide the higher-level calls to APIs to set up a `View`, manage inputs, and the like. This file is compiled when the project is built into bytecode, which you can see when you expand the raw directory.

Figure 9.10. The Fountain project in the Eclipse Package Explorer showing a typical RenderScript application structure



Now that we've touched on the file layout, let's look at the source code. The first file, `Fountain.java`, is trivial: it's the basic Android `Activity` class. As you can see in the following listing, it has an `onCreate()` method that sets the `contentView` to an instance of the `FountainView` class.

Listing 9.14. Basic Android `Activity` class

```
public class Fountain extends Activity {
```

```

private static final String LOG_TAG = "libRS_jni";

private static final boolean DEBUG = false;

private static final boolean LOG_ENABLED = DEBUG ? Config.LOGD : Config.LOGV;

private FountainView mView;

public void onCreate(Bundle icicle) {

    super.onCreate(icicle);

    mView = new FountainView(this);

    setContentView(mView);

}

protected void onResume() {

    Log.e("rs", "onResume");

    super.onResume();

    mView.resume();

}

protected void onPause() {

    Log.e("rs", "onPause");

    super.onPause();

    mView.pause();

}

static void log(String message) {

    if (LOG_ENABLED) {

        Log.v(LOG_TAG, message);

    }

}

```

The FountainView.java file introduces a new type of Android View, the RSSurfaceView, as you can see in the next listing. This class represents the SurfaceView on which your RenderScript code will draw its graphics.

Listing 9.15. RSSurfaceView

```
public class FountainView extends RSSurfaceView {  
    public FountainView(Context context) {  
        super(context);  
    }  
  
    private RenderScriptGL mRS;  
    private FountainRS mRender;  
  
    public void surfaceChanged(SurfaceHolder holder, int format, int w,  
    int h) {  
        super.surfaceChanged(holder, format, w, h);  
        if (mRS == null) {  
            RenderScriptGL.SurfaceConfig sc = new  
            RenderScriptGL.SurfaceConfig();  
            mRS = createRenderScriptGL(sc);  
            mRS.setSurface(holder, w, h);  
            mRender = new FountainRS();  
            mRender.init(mRS, getResources(), w, h);  
        }  
    }  
  
    protected void onDetachedFromWindow() {  
        if (mRS != null) {  
            mRS = null;  
            destroyRenderScriptGL();  
        }  
    }  
}
```

1 Create new
RenderScript

2 Create the
FountainRS class

```

public boolean onTouchEvent(MotionEvent ev) {
    int act = ev.getActionMasked();
    if (act == ev.ACTION_UP) {
        mRender.newTouchPosition(0, 0, 0, ev.getPointerId(0));
        return false;
    } else if (act == MotionEvent.ACTION_POINTER_UP) {
        int pointerIndex = ev.getActionIndex();
        int pointerId = ev.getPointerId(pointerIndex);
        mRender.newTouchPosition(0, 0, 0, pointerId);
    }
    int count = ev.getHistorySize();
    int pcount = ev.getPointerCount();

    for (int p=0; p < pcount; p++) {
        int id = ev.getPointerId(p);
        mRender.newTouchPosition(ev.getX(p),
                               ev.getY(p),
                               ev.getPressure(p),
                               id);

        for (int i=0; i < count; i++) {
            mRender.newTouchPosition(ev.getHistoricalX(p, i),
                                   ev.getHistoricalY(p, i),
                                   ev.getHistoricalPressure(p, i),
                                   id);
        }
    }
    return true;
}
}

```

If you look at the listing, you'll notice in the `surfaceChanged()` method a new RenderScript class as well as a `FountainRS` class ①. The code

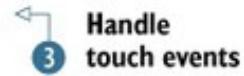
```

RenderScriptGL.SurfaceConfig sc = new RenderScriptGL.SurfaceConfig();

mRS = createRenderScriptGL(sc);

```

is important in that it not only creates a `RenderScriptGL` object that contains the surface our graphics go into, but the `SurfaceConfig` class allows us to set all the major properties for the drawing surface (such as depth). The `FountainRS` class is important in that it acts as a renderer for the `FountainView` ② `SurfaceView` as well as controls the actual RenderScript. One of the other important things this `FountainView` class does is handle touch events with the `onTouchEvent()` method and pass these events to the RenderScript ③.



The next class we'll look at is `FountainRS`, shown in the following listing.

Listing 9.16. FountainRS class

```
public class FountainRS {
    public static final int PART_COUNT = 50000;

    public FountainRS() {
    }

    private Resources mRes;
    private RenderScriptGL mRS;
    private ScriptC_fountain mScript;
    public void init(RenderScriptGL rs, Resources res,
        int width, int height) {
        mRS = rs;
        mRes = res;
        ProgramFragmentFixedFunction.Builder pfb = new
        ProgramFragmentFixedFunction.Builder(rs);
        pfb.setVaryingColor(true);
        rs.bindProgramFragment(pfb.create());
        ScriptField_Point points =
        new ScriptField_Point(mRS, PART_COUNT); ① Bind ScriptC_fountain class
        Mesh.AllocationBuilder smb = new Mesh.AllocationBuilder(mRS);
        smb.addVertexAllocation(points.getAllocation());
        smb.addIndexSetType(Mesh.Primitive.POINT);
        Mesh sm = smb.create();
```

```

        mScript = new ScriptC_fountain(mRS, mRes, R.raw.fountain);
        mScript.set_partMesh(sm);
        mScript.bind_point(points);
        mRS.bindRootScript(mScript);
    }

    boolean holdingColor[] = new boolean[10];
    public void newTouchPosition(float x, float y,
→float pressure, int id) {
        if (id >= holdingColor.length) {
            return;
        }
        int rate = (int)(pressure * pressure * 500.f);
        if (rate > 500) {
            rate = 500;
        }
        if (rate > 0) {
            mScript.invoke_addParticles(rate, x, y, id, !holdingColor[id]);
            holdingColor[id] = true;
        } else {
            holdingColor[id] = false;
        }
    }
}

```

When developing a graphical RenderScript application, you'll have a class called `ClassNameRS` that acts as a communication channel between your RenderScript file and the rest of the Android application. (RenderScript compute projects don't have a file like this.) The `FountainRS` class interacts with the RenderScript code in `fountain.rs` via interfaces exposed by `ScriptC_fountain`, a class generated by the ADT when you build the project and found in the `gen` folder. The `ScriptC_fountain` class binds to the RenderScript bytecode so the `RenderScriptGL` context knows which RenderScript to

bind to ①. This may sound somewhat complicated, and it is, but the ADT or Android tooling manages most of this for you.

Finally, let's look at the C code in `fountain.rs`, shown in [listing 9.17](#). The first thing you'll notice is how simple it is. The code draws a simple cascade of points whose center is the point touched on the screen. It's important to note that all the methods to capture the information about where the user presses are captured, handled, and passed down to this class via the higher-level .java classes already discussed, and that `fountain.rs` is solely focused on drawing.

Listing 9.17. C code in `fountain.rs`

```
#pragma version(1)
#pragma rs java_package_name(com.example.android.rs.fountain)
#pragma stateFragment(parent)
#include "rs_graphics.rsh"

static int newPart = 0;
rs_mesh partMesh;

typedef struct __attribute__((packed, aligned(4))) Point {
    float2 delta;
    float2 position;
    uchar4 color;
} Point_t;
Point_t *point;

int root() {
    float dt = min(rsGetDt(), 0.1f);
    rsgClearColor(0.f, 0.f, 0.f, 1.f);
    const float height = rsgGetHeight();
    const int size = rsAllocationGetDimX(rsGetAllocation(point));
    float dy2 = dt * (10.f);
    Point_t *p = point;
    for (int ct=0; ct < size; ct++) {
        p->delta.y += dy2;
        p->position += p->delta;
        if ((p->position.y > height) && (p->delta.y > 0)) {
            p->delta.y *= -0.3f;
        }
        p++;
    }
}
```

Required pragma
directives class

```

    rsgDrawMesh(partMesh);
    return 1;
}

static float4 partColor[10];
void addParticles(int rate, float x, float y, int index, bool newColor)
{
    if (newColor) {
        partColor[index].x = rsRand(0.5f, 1.0f);
        partColor[index].y = rsRand(1.0f);
        partColor[index].z = rsRand(1.0f);
    }
    float rMax = ((float)rate) * 0.02f;
    int size = rsAllocationGetDimX(rsGetAllocation(point));
    uchar4 c = rsPackColorTo8888(partColor[index]);

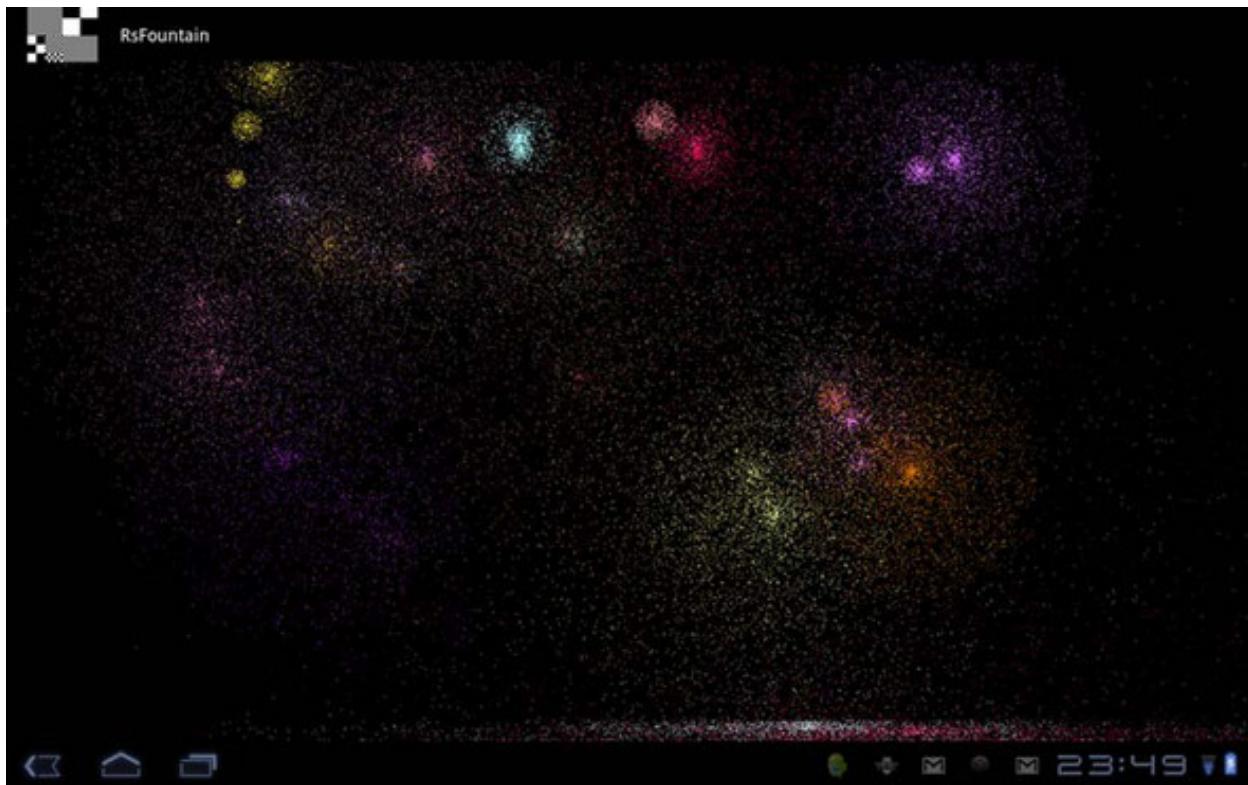
    Point_t * np = &point[newPart];
    float2 p = {x, y};
    while (rate--) {
        float angle = rsRand(3.14f * 2.f);
        float len = rsRand(rMax);
        np->delta.x = len * sin(angle);
        np->delta.y = len * cos(angle);
        np->position = p;
        np->color = c;
        newPart++;
        np++;
        if (newPart >= size) {
            newPart = 0;
            np = &point[newPart];
        }
    }
}

```

The first thing to note is the inclusion of two pragmas that must be part of any RenderScript file, which provide the version and package name. Also note the use of two functions familiar to C developers, `init()` and `root()`. The `init()` function provides a mechanism for setting up variables or constants before anything else is executed in the class. The `root()` method is of course the main root function of the class; for graphics applications, RenderScript will expect to render the frame to be displayed in this method. Other than that, the C code is relatively straightforward.

If you run this application and then touch the screen, you should see a burst of color and cascading dots that fall to the bottom of the screen as shown in [figure 9.11](#). Although you could have done the same thing with Android's 2-D API, and it would have been much easier to code, the RenderScript application is extremely fast with no discernable lag on a Motorola Xoom.

Figure 9.11. Example of the Fountain project running on the Xoom



We can't go into RenderScript in depth in this book—it warrants its own chapter—but we've touched on the main points. You now know the basics of how to build your own RenderScript graphical applications.

9.5. SUMMARY

In this chapter, we've lightly touched on a number of topics related to Android's powerful graphics features. First, we looked at how both Java and XML can be used with the Android Graphics API to describe simple shapes. Next, we examined how to use Android's frame-by-frame XML to create an animation. You also learned how to use more standard pixel manipulation to provide the illusion of movement through Java and the Graphics API. Finally, we delved into Android's support of OpenGL ES. We looked at how to create an OpenGL ES context, and then we built a shape in that context as well as a 3D animated cube. Finally, we took a high-level look at a RenderScript application and discussed how the RenderScript system works inside Android.

Graphics and visualizations are large and complex topics that can easily fill a book. But because Android uses open and well-defined standards and supports an excellent API for graphics, it should be easy for you to use Android's documentation, API, and other

resources, such as Manning's *Java 3D Programming* by Daniel Selman, to develop anything from a new drawing program to complex games.

In the next chapter, we'll move from graphics to working with multimedia. We'll explore working with audio and video to lay the groundwork for making rich multimedia applications.

Chapter 10. Multimedia

This chapter covers

- Playing audio and video
- Controlling the camera
- Recording audio
- Recording video

Today, people use cell phones for almost everything but phone calls, from instant messaging to surfing the web to listening to music and even to watching live streaming TV. Nowadays, a cell phone needs to support multimedia to be considered a usable device. In this chapter, we're going to look at how you can use Android to play audio files, watch video, take pictures, and even record sound and video.

As of Android 2.0, Google decided to phase out the OpenCORE system for Android's multimedia needs and move to a new multimedia system called Stagefright. As of Android 2.3, Stagefright has subsumed OpenCORE and become its replacement. That being said, most of Android's interaction with media is abstracted through the MediaPlayer API, hiding the specific implementation of Stagefright, versus OpenCORE in older versions of Android. What this means to you is that by considering which core media formats you wish to support and by carefully developing your application, it's possible to create applications that will work on Android 2.3 and up as well as older versions of Android that use OpenCORE.

In this chapter, we'll be looking at Stagefright's multimedia architecture and features. Moving on from architecture, we'll explore how to use Stagefright via Android's MediaPlayer API.

10.1. INTRODUCTION TO MULTIMEDIA AND STAGEFRIGHT

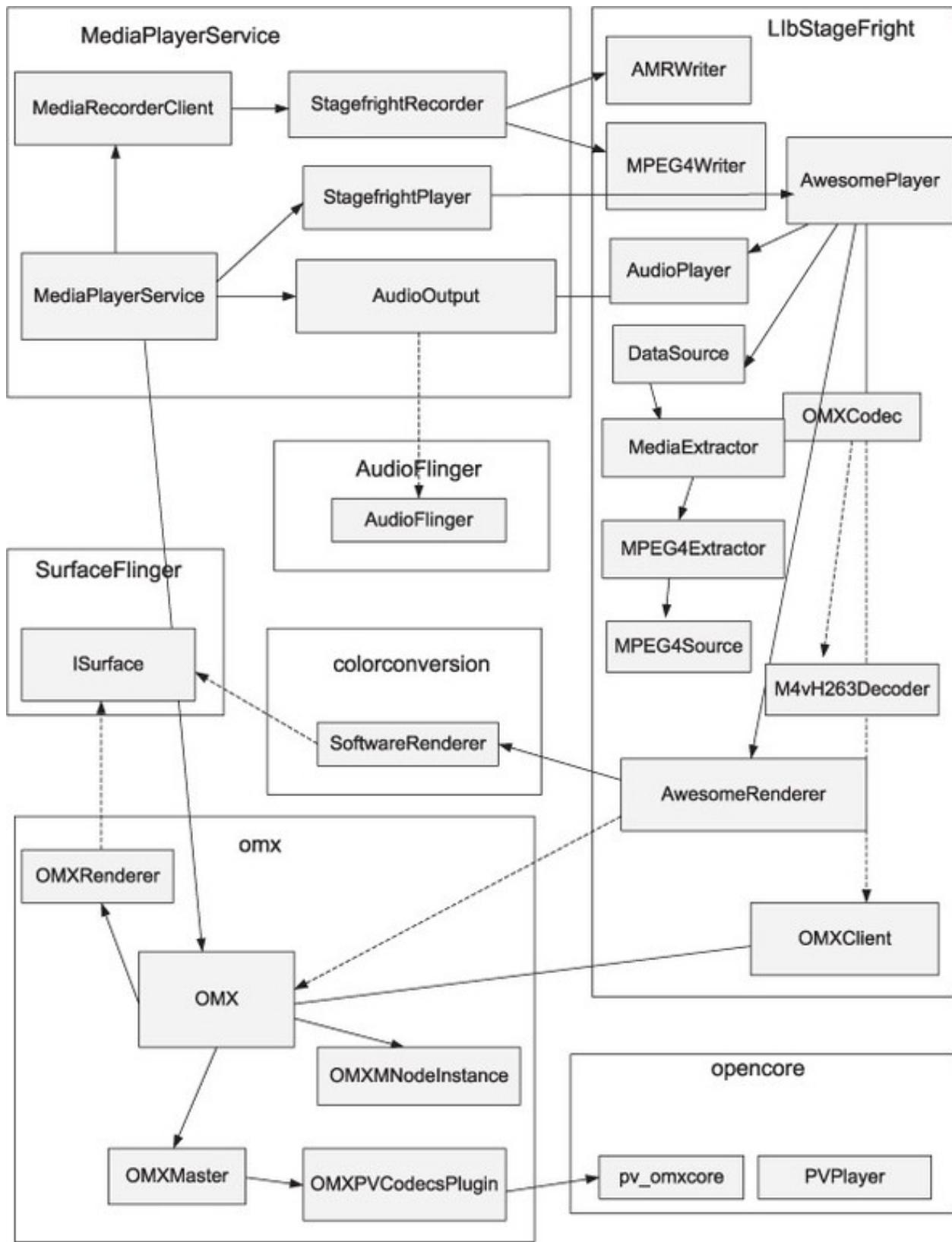
Because the foundation of Android's multimedia platform is Google's new media platform Stagefright, we're going to review Stagefright's architecture and services. Stagefright, as of now, supports the following core media files, services, and features:

- Interfaces for third-party and hardware media codecs, input and output devices, and content policies

- Media playback, streaming, downloading, and progressive playback, including third-Generation Partnership Program (3GPP), Moving Picture Experts Group 4 (MPEG-4), Advanced Audio Coding (AAC), and Moving Picture Experts Group (MPEG) Audio Layer 3 (MP3) containers
- Network protocols including RTSP (TRP, SDP), HTTP progressive streaming, and HTTP live streaming.
- Video and image encoders and decoders, including MPEG-4, International Telecommunication Union H.263 video standard (H.263), Advanced Video Coding (AVC H.264), and the Joint Photographic Experts Group (JPEG)
- Speech codecs, including Adaptive Multi-Rate audio codecs AMR-NB and AMR-WB
- Audio codecs, including MP3, AAC, and AAC+, and more
- Media recording, including 3GPP, VP8, MPEG-4, and JPEG
- Video telephony based on the 3GPP video conferencing standard 3GPP2-M

Stagefright provides all this functionality in a well-laid-out set of services, shown in [figure 10.1](#).

Figure 10.1. Stagefright services



Note

Different API versions, such as 3.0 on the Xoom, may not support all the listed media formats. To check which formats are supported, see <http://developer.android.com/guide/appendix/media-formats.html>.

10.1.1. Stagefright overview

Stagefright has a much simpler internal implementation than OpenCORE. In [figure 10.1](#), you can see a rough outline of how the MediaPlayer works with Stagefright internally. Essentially Stagefright works as follows:

- `MediaExtractor` retrieves track data and corresponding metadata from the file system or HTTP stream.
- `MediaPlayer` is responsible for playing audio as well as managing timing for A/V synchronization for audio.
- Depending on which codec is picked, a local or remote render is created for video play. The system clock is used as the time base for video-only playback.
- `AwesomePlayer` works as the engine to coordinate the preceding classes. It's integrated via `StagefrightPlayer` in the Android `MediaPlayerService`.
- OpenCORE is still partially present for the Ocean Matrix (OMX) video standard for decoding. There are two OMX plugins currently, although this may change in future versions of Android.

In the next section, we'll dive in and use the Android API, and thus Stagefright, to play audio files.

10.2. PLAYING AUDIO

Probably the most basic need for multimedia on a cell phone is the ability to play audio files, whether new ringtones, MP3s, or quick audio notes. Android's `MediaPlayer` is easy to use. At a high level, all you need to do to play an MP3 file is follow these steps:

1. Put the MP3 in the res/raw directory in a project (note that you can also use a URI to access files on the network or via the internet).
2. Create a new instance of the `MediaPlayer`, and reference the MP3 by calling `MediaPlayer.create()`.
3. Call the `MediaPlayer` methods `prepare()` and `start()`.

Let's work through an example to demonstrate how simple this task is. First, create a new project called MediaPlayerExample, with an Activity called MediaPlayerActivity. Now, create a new folder under res/ called raw; you'll store your MP3s in this folder. For this example, we'll use a ringtone for the game Halo 3, which you can download from the Android in Action Google code site at <http://code.google.com/p/android-in-action/>, or you can use your own MP3. Download the Halo 3 theme song and any other MP3s you fancy, and put them in the raw directory. Next, create a simple Button for the music player, as shown in the following listing.

Listing 10.1. main.xml for MediaPlayer example

```
<?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:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Simple Media Player"
        />

    <Button android:id="@+id/playsong"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Halo 3 Theme Song"
        />
</LinearLayout>
```

Next, fill out the MediaPlayerActivity class, as shown in the following listing.

Listing 10.2. MediaPlayerActivity.java

```

public class MediaPlayerActivity extends Activity {
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        Button mybutton = (Button) findViewById(R.id.playsong);
        mybutton.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                MediaPlayer mp =
                    MediaPlayer.create(MediaPlayerActivity.this,
                        R.raw.halotheme);
                mp.start();
                mp.setOnCompletionListener(new OnCompletionListener() {
                    public void onCompletion(MediaPlayer arg0) {
                    }
                });
            }
        });
    }
}

```

Set view and button to play MP3

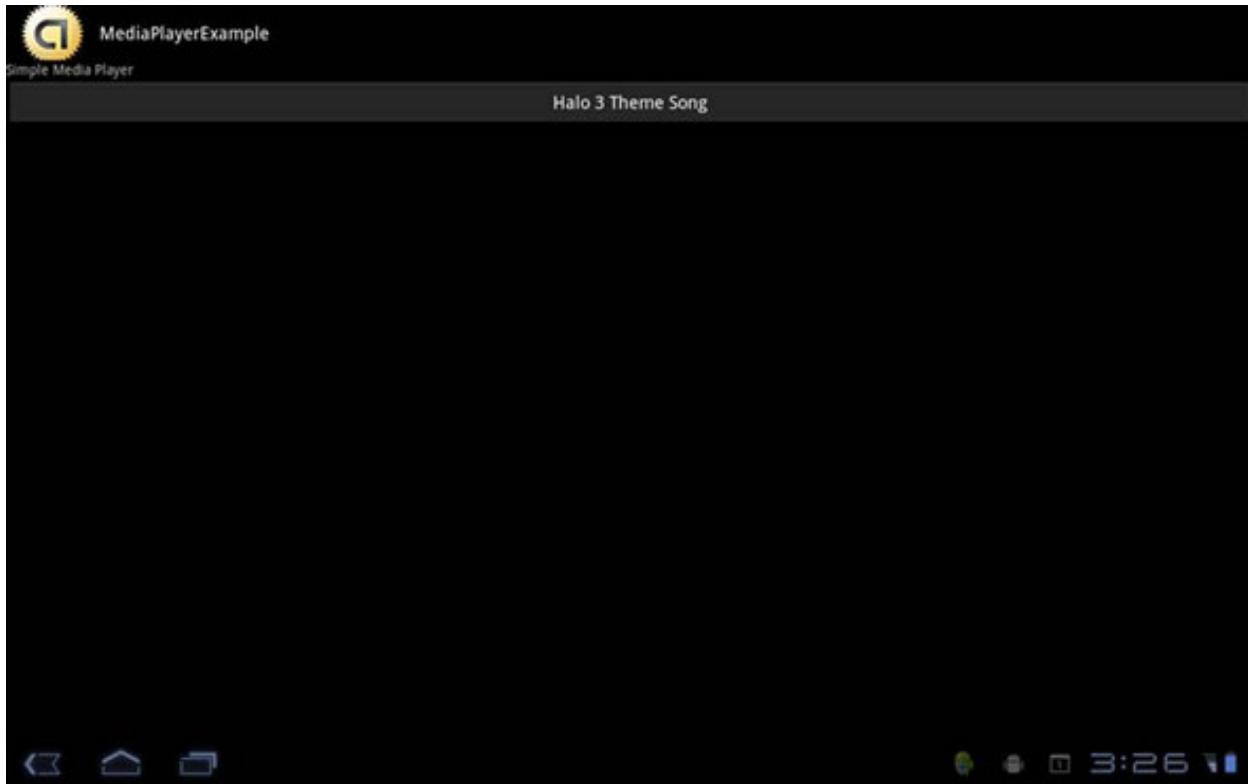
Get context and play MP3

As you can see, playing back an MP3 is easy. In [listing 10.2](#), all we do is use the `View` that we created in [listing 10.1](#) and map the resource ID, `playsong`, to `mybutton`, which we then bind to `setOnClickListener()`. Inside the listener, we create

the `MediaPlayer` instance using the `create(Context context, int resourceId)` method, which takes our context and a resource ID for the MP3. Finally, we set the `setOnCompletionListener`, which will perform some task on completion. For the moment, we do nothing, but you may want to change a button's state or provide a notification to a user that the song is over, or ask if the user would like to play another song. If you want to do any of these things, you'll use this method.

If you compile the application and run it, you should see something like [figure 10.2](#). Click the button, and you should hear the Halo 3 song played back on your device's speakers.

Figure 10.2. Media player example



Now that we've looked at how to play an audio file, let's see how you can play a video file.

10.3. PLAYING VIDEO

Playing a video is slightly more complicated than playing audio with the `MediaPlayer` API, in part because you have to provide a view surface for your video to play on. Android has a `VideoView` widget that handles that task for you; you can use it in any layout manager. Android also provides a number of display options, including scaling and tinting. Let's get started with playing video by creating a new project called Simple Video Player. Next, create a layout, as shown in the following listing.



Note

Currently the emulator has some issues playing video content on certain computers and operating systems. Don't be surprised if your audio or video playback is choppy.



Listing 10.3. main.xml UI for Simple Video Player

```
<?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"
>
    <VideoView android:id="@+id/video"
        android:layout_width="320px"
        android:layout_height="240px"
    />
</LinearLayout>
```



1 Add
VideoView
widget

All we've done in this listing is add the `VideoView` widget 1. It provides a UI widget with Stop, Play, Advance, Rewind, and other buttons, making it unnecessary to add your own. Next, you need to write a class to play the video, as shown in the following listing.

Listing 10.4. SimpleVideo.java

```
public class SimpleVideo extends Activity {
    private VideoView myVideo;
    private MediaController mc;
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        setContentView(R.layout.main);
        myVideo = (VideoView) findViewById(R.id.video);
        File pathToTest= new File
            (Environment.getExternalStorageDirectory(),"test.mp4");
        mc = new MediaController(this);
        mc.setMediaPlayer(myVideo);
        myVideo.setMediaController(mc);
        myVideo.requestFocus();
    }
}
```



1 Create
translucent
window

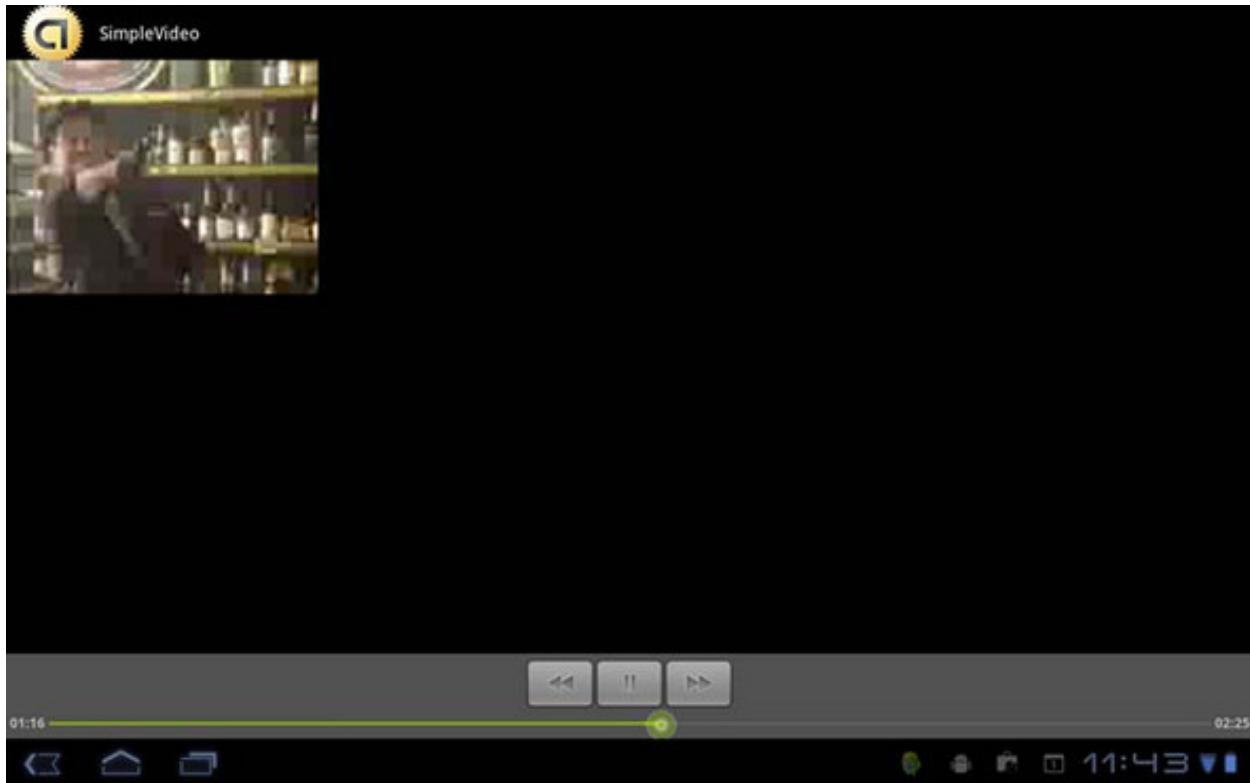
In this listing, we first create a translucent window, which is necessary for

the `SurfaceView` 1. Next, we reference the `VideoView` as a container for playing the video and use its `setVideoPath()` method to have it look at an SD card (using the approved Environment API for this purpose) for our test MP4. Finally, we set up the `MediaController` and use the `setMediaController()` method to perform a callback to the `VideoView` to notify it when our video is finished playing.

Before you can run this application, you'll need to either use the ADB to push a video onto your Android 3.0 device or create one that you'll reference in

the `setVideoPath()` method. You can download this project from the book's source code repository, which includes a number of short, license-free videos you can use for testing. When the videos are on your device's SD card, run the application and touch the screen where the movie will play in the upper-left corner. Doing so will cause the controls to appear. Push Play, and you should see something like [figure 10.3](#).

Figure 10.3. Video and associated player controls in `VideoView`



As you can see, the `VideoView` and `MediaPlayer` classes simplify working with video files. Something you'll need to pay attention to when working with video files is that the emulator and physical devices will react differently with very large media files.

Now that you've seen how simple it is to play media using Android's `MediaPlayer` API, let's look at how you can use a phone's built-in camera or microphone to capture images or audio.

10.4. CAPTURING MEDIA

Using your cell phone to take pictures, record memos, film short videos, and so on, are features that are expected of any such device. In this section, we'll look at how to capture media from the microphone and camera, and also how to write these files to the SD card.

To get started, let's examine how to use the Android `Camera` class to capture images and save them to a file.

10.4.1. Understanding the camera

An important feature of modern cell phones is their ability to take pictures or video using a built-in camera. Some phones even support using the camera's microphone to capture audio. Android, of course, supports all three features and provides a variety of ways to interact with the camera. In this section, we'll look at how to interact with the camera and take photographs.

You'll be creating a new project called SimpleCamera to demonstrate how to connect to a phone's camera to capture images. For this project, you'll use the `Camera` class (<http://mng.bz/E244>) to tie the emulator's (or phone's) camera to a `view`. Most of the code that you'll create for this project deals with showing the input from the camera, but the main work for taking a picture is done by a single method called `takePicture(Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.PictureCallback jpeg)`, which has three callbacks that allow you to control how a picture is taken.

Before we get any further into the `Camera` class and how to use the camera, let's create the project. You'll be creating two classes; because the main class is long, we'll break it into two sections. When you create the project, you'll need to add the `CAMERA` and `WRITE_EXTERNAL_STORAGE` permissions to the manifest, like this:

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

Both permissions are essentially self explanatory.

Note

The Android emulator doesn't allow you to connect to camera devices, such as a webcam, on your computer; all your pictures will display a chessboard. You can connect to a web camera and get live images and video, but doing so requires some hacking. You can find an excellent example of how to do this at Tom Gibara's website, where he has an open source project for obtaining live images from a webcam: www.tomgibara.com/android/camera-source. It's possible that in later versions of the SDK, the emulator will support connections to cameras on the hardware the emulator is running on.

Now, create the example's layout, as shown in the following listing.

Listing 10.5. Main.xml for SimpleCamera

```
<?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">

    <SurfaceView android:id="@+id/surface"
        android:layout_width="fill_parent" android:layout_height="10dip"
        android:layout_weight="1">

    </SurfaceView>

    <Button android:id="@+id/pictureButton" android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text= "Take Picture"
        android:enabled="true" />

</LinearLayout>
```

The next listing shows the first part of CameraExample.java.

Listing 10.6. CameraExample.java

```
public class SimpleCamera extends Activity implements
    SurfaceHolder.Callback {

    private Camera camera;
    private boolean isPreviewRunning = false;
    private SimpleDateFormat timeStampFormat =
        new SimpleDateFormat("yyyyMMddHHmmssSS");
    private static final String TAG = "camera";
    private SurfaceView surfaceView;
    private SurfaceHolder surfaceHolder;
    private Uri targetResource = Media.EXTERNAL_CONTENT_URI;

    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        Log.e(getClass().getSimpleName(), "onCreate");
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        setContentView(R.layout.main);
        this.surfaceView = (SurfaceView) findViewById(R.id.surface);
        this.surfaceHolder = this.surfaceView.getHolder();
        this.surfaceHolder.addCallback(this);
        this.surfaceHolder.setType(SurfaceHolder.
    SURFACE_TYPE_PUSH_BUFFERS);

        Button takePicture = (Button) findViewById(R.id.pictureButton);
        takePicture.setOnClickListener(new OnClickListener() {
            public void onClick(View view) {
                try {
                    takePicture();
                } catch (Exception e) {
                    Log.e(TAG, e.toString());
                    e.printStackTrace();
                }
            }
        });
    }
}
```

```

        }

    public boolean onCreateOptionsMenu(android.view.Menu menu) {
        MenuItem item = menu.add(0, 0, 0, "View Pictures"); ←
        item.setOnMenuItemClickListener(new
    ↵MenuItem.OnMenuItemClickListener() {

        public boolean onMenuItemClick(MenuItem item) {
            Intent intent = new Intent(Intent.ACTION_VIEW,
    ↵SimpleCamera.this.targetResource);
            startActivity(intent);
            return true;
        }
    });
    return true;
}

protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
}

Camera.PictureCallback mPictureCallbackRaw =
    ↵new Camera.PictureCallback() { ←
        2 Create
        PictureCallback

        public void onPictureTaken(byte[] data, Camera c) {
            SimpleCamera.this.camera.startPreview();
        }
    };

Camera.ShutterCallback mShutterCallback =
    ↵new Camera.ShutterCallback() { ←
        3 Create
        ShutterCallback

        public void onShutter() {
        }
    };
}

```

This listing is straightforward. First, we set variables for managing a `SurfaceView` and then set up the `View`. Next, we create a menu and menu option that will float over the surface when the user clicks the Menu button on the phone while the application is

running ①. Doing so will open Android's picture browser and let the user view the photos on the camera. Next, we create the first `PictureCallback`, which is called when a picture is first taken ②. This first callback captures the `PictureCallback`'s only method, `onPictureTaken(byte[] data, Camera camera)`, to grab the raw image data directly from the camera. Next, we create a `ShutterCallback`, which can be used with its `onShutter()` method, to play a sound; here we don't call the `onShutter()` method ③.

We'll continue with `CameraExample.java` in the next listing.

Listing 10.7. CameraExample.java, continued

```
public boolean takePicture() {
    ImageCaptureCallback camDemo = null;

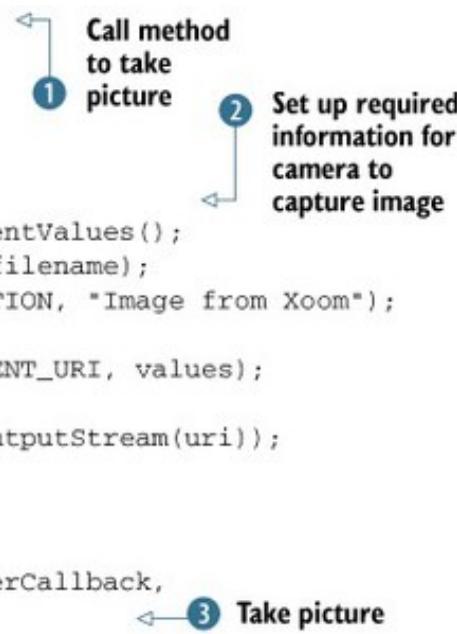
    try {
        String filename =
this.timeStampFormat.format(new Date());
        ContentValues values = new ContentValues();
        values.put(MediaColumns.TITLE, filename);
        values.put(ImageColumns.DESCRIPTION, "Image from Xoom");
        Uri uri =
getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, values);
        camDemo = new
ImageCaptureCallback(getContentResolver().openOutputStream(uri));
    } catch (Exception ex) {

    }

    this.camera.takePicture(this.mShutterCallback,
this.mPictureCallbackRaw, camDemo);           ← 3 Take picture
    return true;
}

protected void onResume() {
    Log.e(getClass().getSimpleName(), "onResume");
    super.onResume();
}

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



```

protected void onStop() {
    super.onStop();
}

public void surfaceChanged(SurfaceHolder holder, int format,
int w, int h) {
    if (this.isPreviewRunning) {
        this.camera.stopPreview();
    }
    Camera.Parameters p = this.camera.getParameters();
    p.setPreviewSize(w, h);
    this.camera.setParameters(p);
    try {
        this.camera.setPreviewDisplay(holder);
    } catch (IOException e) {

        e.printStackTrace();
    }
    this.camera.startPreview();
    this.isPreviewRunning = true;
}

public void surfaceCreated(SurfaceHolder holder) {
    this.camera = Camera.open();
}

public void surfaceDestroyed(SurfaceHolder holder) {
    this.camera.stopPreview();
    this.isPreviewRunning = false;
    this.camera.release();
}
)

```

This listing is more complicated than [listing 10.5](#). Much of the code is about managing the surface for the camera preview. The first line is the start of an implementation of the

method `takePicture()` ①, which checks to see whether the Take Picture button was clicked. If it was, we set up the creation of a file; and by using the `ImageCaptureCallback` (which we'll define in [listing 10.7](#)), we create

an `OutputStream` to which we write our image data ②, including not only the image but the filename and other metadata. Next, we call the method `takePicture()` and pass to it the three callbacks `mShutterCallback`, `mPictureCallbackRaw`, and `camDemo.mPictureCallbackRaw` is the raw image, and `camDemo` writes the image to a file on the SD card ③, as you can see in the following listing.

Listing 10.8. ImageCaptureCallback.java

```
public class ImageCaptureCallback implements PictureCallback {
```

```

private OutputStream filoutputStream;

public ImageCaptureCallback(OutputStream filoutputStream) {
    this.filoutputStream = filoutputStream;
}

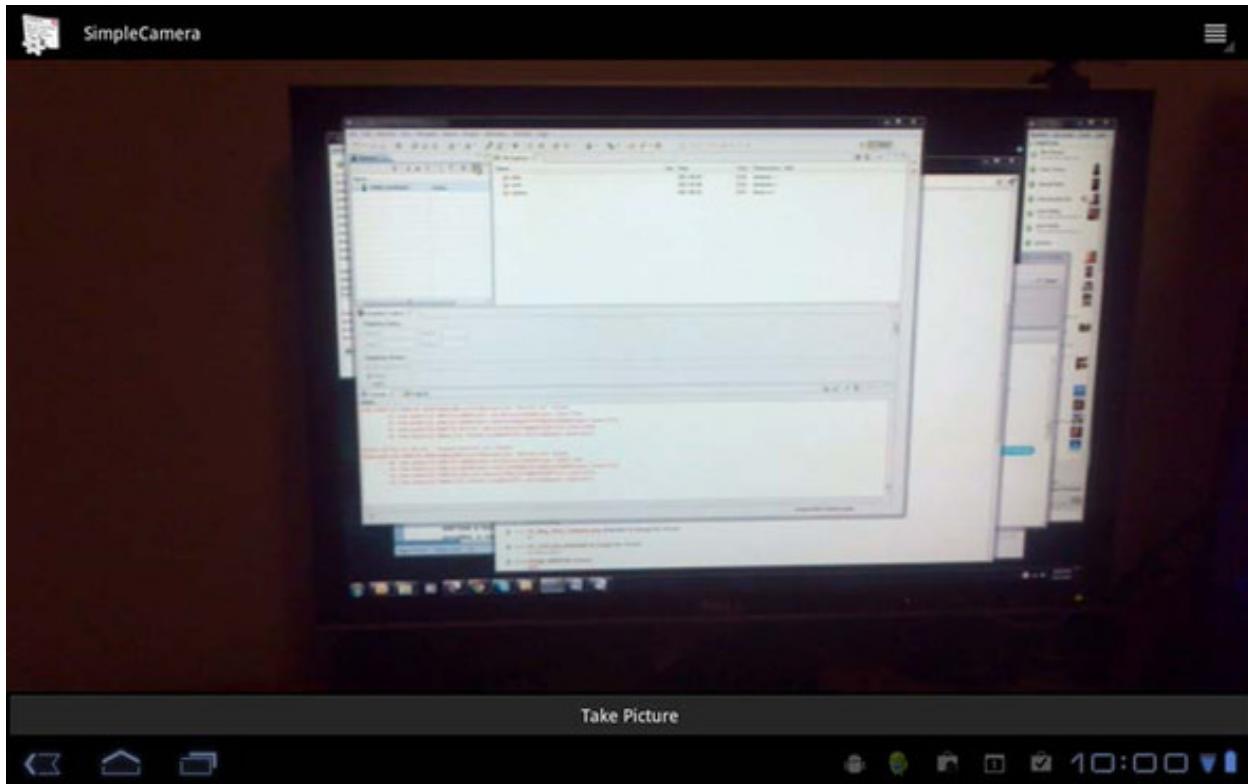
public void onPictureTaken(byte[] data, Camera camera) {
    try {
        this.filoutputStream.write(data);
        this.filoutputStream.flush();
        this.filoutputStream.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

```

Here, the class implements the `PictureCallback` interface and provides two methods. The constructor creates a stream to write data to, and the second method, `onPictureTaken()`, takes binary data and writes to the SD card as a JPEG.

If you build this project and attempt to run it on a 3.0 emulator, it's likely that it won't work; as noted earlier, the emulator in 3.0 has issues with trying to detect a camera device. Probably the best to test camera support on an emulator is to create an input to the emulator that simulates a camera. One tool to do this is NyARToolkit, which you can get at <http://mng.bz/gdBh>. Because simulating a camera in Android isn't an optimal way to test your code, it's by far preferred to use a real device such as the Motorola Xoom. If you run this project in a Xoom, you should see something like [figure 10.4](#).

Figure 10.4. Using the SimpleCamera application on a Xoom



Now that you've seen how the `Camera` class works in Android, let's look at how to capture or record audio from a camera's microphone. In the next section, we'll explore the `MediaRecorder` class, and you'll write recordings to an SD card.

10.4.2. Capturing audio

Now we'll look at using the onboard microphone to record audio. In this section, you'll use the Android `MediaRecorder` example from the Google Android Developers list, which you can find at <http://code.google.com/p/unlocking-android/>. The code shown in this section has been updated slightly.

Note

Audio capture requires a physical device running Android, because it's not currently supported in the Android emulator.

In general, recording audio or video follows the same process in Android:

1. Create an instance of `android.media.MediaRecorder`.
2. Create an instance of `android.content.ContentValues`, and add properties such as `TITLE`, `TIMESTAMP`, and the all-important `MIME_TYPE`.
3. Create a file path for the data to go to, using `android.content.ContentResolver`.
4. To set a preview display on a view surface, use `MediaRecorder.setPreviewDisplay()`.
5. Set the source for audio, using `MediaRecorder.setAudioSource()`.
6. Set the output file format, using `MediaRecorder.setOutputFormat()`.
7. Set your encoding for audio, using `MediaRecorder.setAudioEncoder()`.
8. Use `prepare()` and `start()` to prepare and start your recordings.
9. Use `stop()` and `release()` to gracefully stop and clean up your recording process.

Although recording media isn't especially complex, you may notice that it's more involved than playing it. To understand how to use the `MediaRecorder` class, we'll look at an application. To begin, create a new project called `SoundRecordingDemo`. Next, edit the `AndroidManifest.xml` file and add the following:

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

This code will allow the application to record the audio files and play them. Next, create the class shown in the following listing.

Listing 10.9. SoundRecordingdemo.java

```
public class SoundRecordingDemo extends Activity {

    MediaRecorder mRecorder;
    File mSampleFile = null;
    static final String SAMPLE_PREFIX = "recording";
    static final String SAMPLE_EXTENSION = ".3gpp";
    private static final String OUTPUT_FILE = "/sdcard/audiooutput.3gpp";
    private static final String TAG = "SoundRecordingDemo";

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.mRecorder = new MediaRecorder();

        Button startRecording = (Button) findViewById(R.id.startrecording);
        Button stopRecording = (Button) findViewById(R.id.stoprecording);

        startRecording.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                startRecording();
            }
        });

        stopRecording.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                stopRecording();
                addToDB();
            }
        });
    }
}
```

```
protected void addToDB() {
    ContentValues values = new ContentValues(3);
    long current = System.currentTimeMillis();

    values.put(MediaColumns.TITLE, "test_audio");
    values.put(MediaColumns.DATE_ADDED, (int) (current / 1000));
    values.put(MediaColumns.MIME_TYPE, "audio/3gpp");
    values.put(MediaColumns.DATA, OUTPUT_FILE);
    ContentResolver contentResolver = getContentResolver();

    Uri base = MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
    Uri newUri = contentResolver.insert(base, values);

    sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE,
        newUri)); }

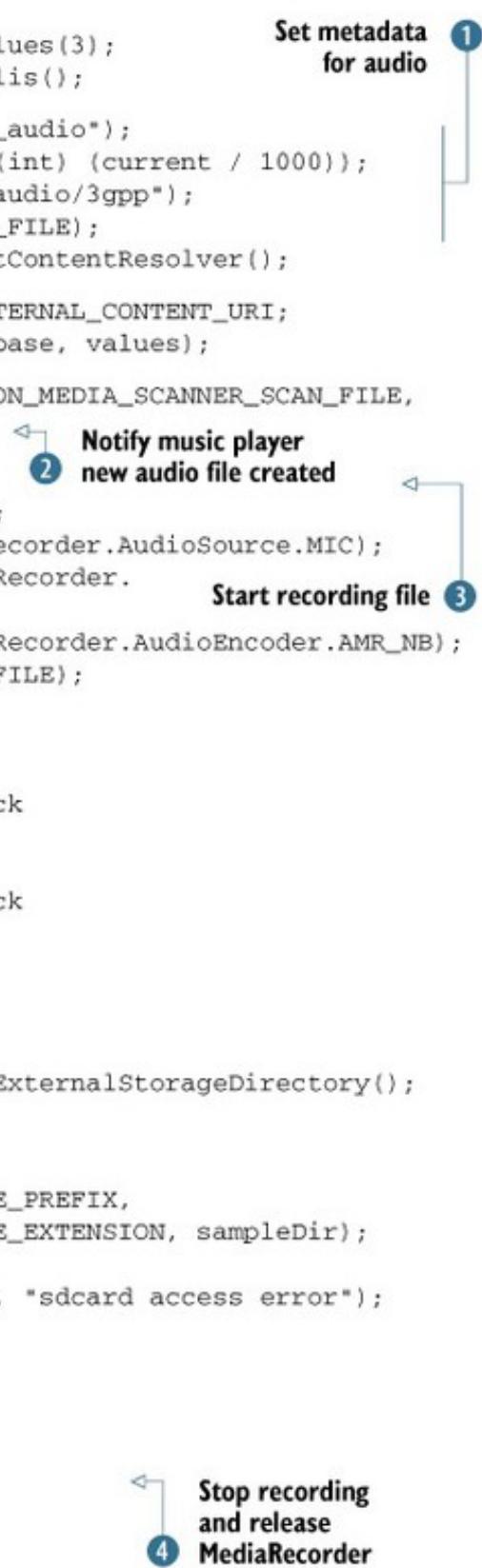
protected void startRecording() {
    this.mRecorder = new MediaRecorder();
    this.mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    this.mRecorder.setOutputFormat(MediaRecorder.
OutputFormat.THREE_GPP);
    this.mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    this.mRecorder.setOutputFile(OUTPUT_FILE);
    try {
        this.mRecorder.prepare();
    } catch (IllegalStateException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    } catch (IOException e1) {
        // TODO Auto-generated catch block
        e1.printStackTrace();
    }
    this.mRecorder.start();

    if (this.mSampleFile == null) {
        File sampleDir = Environment.getExternalStorageDirectory();

        try {
            this.mSampleFile =
File.createTempFile(SoundRecordingDemo.SAMPLE_PREFIX,
                SoundRecordingDemo.SAMPLE_EXTENSION, sampleDir);
        } catch (IOException e) {
            Log.e(SoundRecordingDemo.TAG, "sdcard access error");
            return;
        }
    }
}

protected void stopRecording() {
    this.mRecorder.stop();
    this.mRecorder.release();
}

})
```



Set metadata for audio 1

Notify music player
new audio file created 2

Start recording file 3

Stop recording and release MediaRecorder 4

The first part of the code creates the buttons and button listeners to start and stop the recording; reference main.xml by downloading the code. The first part of the listing that you need to pay attention to is the `addToDB()` method. In this method, we set all the

metadata for the audio file we plan to save, including the title, date, and type of file 1. Next, we call the `Intent ACTION_MEDIA_SCANNER_SCAN_FILE` to notify applications such as Android's Music Player that a new audio file has been created 2. Calling this `Intent` allows us to use the Music Player to look for new files in a playlist and play the files.

Next, we create the `startRecording()` method, which creates a new `MediaRecorder` 3. As in the steps in the beginning of this section, we set an audio source, which is the microphone; set an output format as `THREE_GPP`; set the audio encoder type to `AMR_NB`; and then set the output file path to write the file. Next, we use the methods `prepare()` and `start()` to enable audio recording.

Finally, we create the `stopRecording()` method to stop the `MediaRecorder` from saving audio 4 by using the methods `stop()` and `release()`.

If you build this application and run the emulator with the SD card image from the previous section, you should be able to launch the application from Eclipse and click the Start Recording button. After a few seconds, click the Stop Recording button and open the DDMS; you should be able to navigate to the `sdcard` folder and see your recordings, as shown in [figure 10.5](#). Alternately you can use your device's media player, file browser, or the like to navigate to that file and play it.

Figure 10.5. An example of audio files being saved to the SD card image in the file explorer

Name	Size	Date	Time	Permissions	Info
▶ Download		2011-04-08	01:05	drwxrwxr-x	
▶ MoboTap		2011-04-01	15:09	drwxrwxr-x	
▶ Movies		2011-02-24	19:44	drwxrwxr-x	
▶ Music		2011-02-04	13:45	drwxrwxr-x	
▶ Notifications		2011-02-24	19:44	drwxrwxr-x	
▶ Pictures		2011-02-24	19:44	drwxrwxr-x	
▶ Podcasts		2011-02-24	19:44	drwxrwxr-x	
▶ Ringtones		2011-02-24	19:44	drwxrwxr-x	
▶ airportmania-hd.bin	11680...	2011-04-13	19:38	-rw-rw-r--	
▶ apps2SD		2011-04-02	20:31	drwxrwxr-x	
▶ audiooutput.3gpp	6523	2011-04-21	00:32	-rw-rw-r--	
▶ documents		2011-04-01	16:57	drwxrwxr-x	

If music is playing on your computer's audio system, the Android emulator will pick it up and record it directly from the audio buffer (it's not recording from a microphone). You can then easily test whether it recorded sound by opening the Android Music Player

and selecting Playlists > Recently Added. It should play your recorded file, and you should be able to hear anything that was playing on your computer at the time.

As of version 1.5, Android supported the recording of video, although many developers found it difficult and some vendors implemented their own customer solutions to support video recording. With the releases of Android 2.0 to Android 3.1, video has become far easier to work with, both for playing as well as recording. You'll see how much easier in the next section about using the `MediaRecorder` class to write a simple application for recording video.

10.4.3. Recording video

Video recording on Android is no more difficult than recording audio, with the exception that you have a few different fields. But there's one important difference—unlike with recording audio data, Android requires you to first preview a video feed before you can record it by passing it a surface object, much as we did with the camera application earlier in this chapter. It's worth repeating this point because when Android started supporting video recording, many developers found themselves unable to record video: you must always provide a surface object. This may be awkward for some applications, but it's currently required in Android up to 2.2 and up.

Also, as with recording audio, you have to provide several permissions to Android so you can record video. The new one is `RECORD_VIDEO`, which lets you use the camera to record video. The other permissions are `CAMERA`, `RECORD_AUDIO`, and `WRITE_EXTERNAL_STORAGE`, as shown in the following listing. Go ahead and set up a new project called VideoCam, and use the permissions in this `AndroidManifest.xml` file.

Listing 10.10. `AndroidManifest.xml`

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

<manifest xmlns:android="http://schemas.android.com/apk/res/android"

    package="com.msi.manning.chapter10.VideoCam"

    android:versionCode="1"

    android:versionName="1.0">

    <application android:icon="@drawable/icon"

        android:label="@string/app_name">

        <activity android:name=".VideoCam"

            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>

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

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

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

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

<uses-feature android:name="android.hardware.camera" />

</manifest>

```

One interesting thing that is worth pointing out about the manifest file for this project is the `uses-feature` statement:

```
<uses-feature android:name="android.hardware.camera" />
```

This statement is needed for the application to run, but in general you would use this statement to tell external entities what software and/or hardware the application depends on. This is useful for informing users that your application will only run on devices that have specific hardware, such as a camera or a 3G radio. To read more, see <http://mng.bz/PdE4>.

Now that you've defined the manifest, you need to create a simple layout that has a preview area and some buttons to start, stop, pause, and play your video recording. The layout is shown in the following listing.

Listing 10.11. main.xml for VideoCam

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

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:orientation="vertical"

    android:layout_width="fill_parent"

    android:layout_height="fill_parent">

<RelativeLayout android:layout_width="fill_parent"

    android:layout_height="wrap_content"

    android:id="@+id/relativeVideoLayoutView"

    android:layout_centerInParent="true">

    <VideoView android:id="@+id/videoView" android:layout_width="176px"

        android:layout_height="144px"

        android:layout_centerInParent="true"/>

</RelativeLayout>

<LinearLayout

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:orientation="horizontal"

    android:layout_centerHorizontal="true"

    android:layout_below="@+id/relativeVideoLayoutView">

    <ImageButton android:id="@+id/playRecordingBtn"

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:background="@drawable/play"

        />

    <ImageButton android:id="@+id/bgnBtn"

        android:layout_width="wrap_content"

        android:layout_height="wrap_content"

        android:background="@drawable/record"

        android:enabled="false"

        />

</LinearLayout>
```

```
</RelativeLayout>
```

Note

You'll need to download this code from <https://code.google.com/p/android-in-action/> to get the open source icons that we use for the buttons, or you can use your own.

Video recording follows a set of steps that are similar to those for audio recording:

1. Create an instance of `android.media.MediaRecorder`.
2. Set up a `VideoView`.
3. To set a preview display on a `View` surface, use `MediaRecorder.setPreviewDisplay()`.
4. Set the source for audio using `MediaRecorder.setAudioSource()`.
5. Set the source for video using `MediaRecorder.setVideoSource()`.
6. Set the encoding for audio using `MediaRecorder.setAudioEncoder()`.
7. Set the encoding for video using `MediaRecorder.setVideoEncoder()`.
8. Set the output file format using `MediaRecorder.setOutputFormat()`.
9. Set the video size using `setVideoSize()`. (At the time this book was written, there was a bug in `setVideoSize()` that limited it to 320 by 240.)
10. Set the video frame rate, using `setVideoFrameRate()`.
11. Use `prepare()` and `start()` to prepare and start your recordings.
12. Use `stop()` and `release()` to gracefully stop and clean up the recording process.

As you can see, using video is similar to using audio. Let's finish the example by using the code in the following listing.

Listing 10.12. VideoCam.java

```
public class VideoCam extends Activity implements SurfaceHolder.Callback {

    private MediaRecorder recorder = null;

    private static final String OUTPUT_FILE =
        "/sdcard/uatestvideo.mp4";

    private static final String TAG = "RecordVideo";

    private VideoView videoView = null;

    private ImageButton startBtn = null;

    private ImageButton playRecordingBtn = null;
```

```
private Boolean playing = false;

private Boolean recording = false;

public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);

    startBtn = (ImageButton) findViewById(R.id.bgnBtn);

    playRecordingBtn = (ImageButton)

        findViewById(R.id.playRecordingBtn);

    videoView = (VideoView)this.findViewById(R.id.videoView);

    final SurfaceHolder holder = videoView.getHolder();

    holder.addCallback(this);

    holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);

    startBtn.setOnClickListener(new OnClickListener() {

        public void onClick(View view) {

            if(!VideoCam.this.recording & !VideoCam.this.playing)

            {

                try

                {

                    beginRecording(holder);

                    playing=false;

                    recording=true;

                    startBtn.setBackgroundResource(R.drawable.stop);

                } catch (Exception e) {

                    Log.e(TAG, e.toString());

                    e.printStackTrace();

                }

            }

            else if(VideoCam.this.recording)
```

```

    {

        try
        {

            stopRecording();

            playing = false;

            recording= false;

            startBtn.setBackgroundResource(R.drawable.play);

        }catch (Exception e) {

            Log.e(TAG, e.toString());

            e.printStackTrace();

        }

    }

}

}};

playRecordingBtn.setOnClickListener(new OnClickListener() {

    public void onClick(View view)
    {

        if(!VideoCam.this.playing & !VideoCam.this.recording)

        {

            try
            {

                playRecording();

                VideoCam.this.playing=true;

                VideoCam.this.recording=false;

                playRecordingBtn.setBackgroundResource

(R.drawable.stop);

            } catch (Exception e) {

                Log.e(TAG, e.toString());

```

```
        e.printStackTrace();

    }

}

else if(VideoCam.this.playing)

{

    try

    {

        stopPlayingRecording();

        VideoCam.this.playing = false;

        VideoCam.this.recording= false;

        playRecordingBtn.setBackgroundResource

(R.drawable.play);

    }catch (Exception e) {

        Log.e(TAG, e.toString());

        e.printStackTrace();

    }

}

}

});;

}

public void surfaceCreated(SurfaceHolder holder) {

    startBtn.setEnabled(true);

}

public void surfaceDestroyed(SurfaceHolder holder) {

}

public void surfaceChanged(SurfaceHolder holder, int format, int width,

    int height) {

    Log.v(TAG, "Width x Height = " + width + "x" + height);

}
```

```
private void playRecording() {  
    MediaController mc = new MediaController(this);  
    videoView.setMediaController(mc);  
    videoView.setVideoPath(OUTPUT_FILE);  
    videoView.start();  
}  
  
private void stopPlayingRecording() {  
    videoView.stopPlayback();  
}  
  
private void stopRecording() throws Exception {  
    if (recorder != null) {  
        recorder.stop();  
    }  
}  
  
protected void onDestroy() {  
    super.onDestroy();  
    if (recorder != null) {  
        recorder.release();  
    }  
}  
  
private void beginRecording(SurfaceHolder holder) throws Exception {  
    if(recorder!=null)  
    {  
        recorder.stop();  
        recorder.release();  
    }  
    File outFile = new File(OUTPUT_FILE);  
    if(outFile.exists())  
    {
```

```

        outFile.delete();

    }

    try {

        recorder = new MediaRecorder();

        recorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);

        recorder.setAudioSource(MediaRecorder.AudioSource.MIC);

        recorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);

        recorder.setVideoSize(320, 240);

        recorder.setVideoFrameRate(15);

        recorder.setVideoEncoder(MediaRecorder.VideoEncoder.MPEG_4_SP);

        recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);

        recorder.setMaxDuration(20000);

        recorder.setPreviewDisplay(holder.getSurface());

        recorder.setOutputFile(OUTPUT_FILE);

        recorder.prepare();

        recorder.start();

    }

    catch(Exception e) {

        Log.e(TAG, e.toString());

        e.printStackTrace();

    }

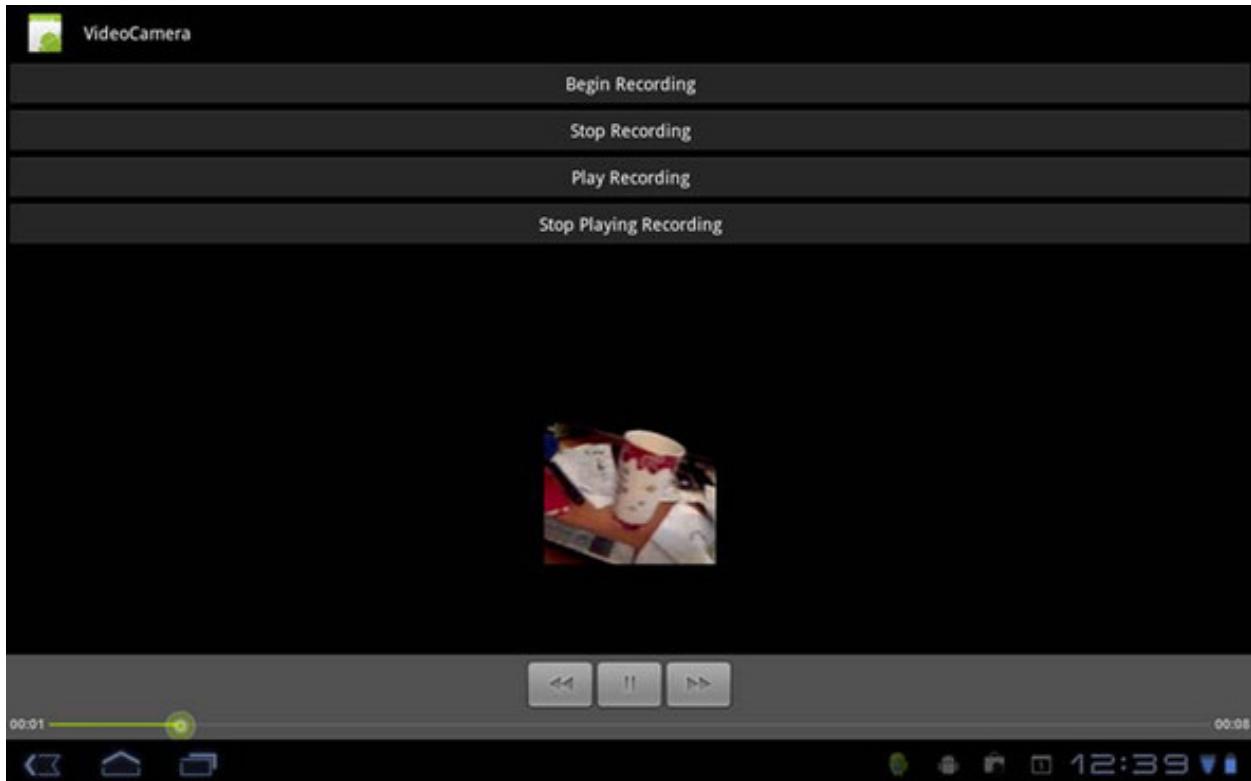
}

```

Because much of this listing is similar to other code in this chapter, we won't describe everything that's happening. If you look quickly at the code, you'll note that it's relatively simple. The first thing we do, besides setting some fields, is set up our surface to support the camera preview, much as we did in the camera application earlier in this chapter. The next part of the code that's important is the `beginRecording()` method. First, this method checks to make sure that everything is ready to record a video file by making sure that the camera is free and that it can record the output file. Then, the method closely follows the preceding processes to set up the camera for recording before calling `prepare()` and then `start()`.

Unfortunately, as we noted with the camera project, there's no easy way to test your application in the emulator. For this example, we've pushed the application to a cell phone to test the camera, and we used the DDMS to note the file that was recorded and to play it back. You can see an example of the output, captured with the DDMS from a Motorola Xoom, in [figure 10.6](#).

Figure 10.6. Photograph of the VideoCam application running on a Xoom



Support for Multiple Cameras

As of Android 2.3, Android supports multiple cameras. For example, the Motorola Xoom has a front-facing camera and a back-facing camera to support applications like video conferencing. For this reason, Android now supports multiple new API calls for the `Camera` class to help developers support multiple cameras.

The `Camera.CameraInfo` class stores the device's camera orientation. Currently it supports `CAMERA_FACING_BACK` and `CAMERA_FACING_FRONT`. You can use this class to essentially auto-discover which camera is which, as in the following code snippet:

```
numberOfCameras = Camera.getNumberOfCameras();  
  
CameraInfo cameraInfo = new CameraInfo();  
  
for (int i = 0; i < numberOfCameras; i++) {
```

```

        Camera.getCameraInfo(i, cameraInfo);

        if (cameraInfo.facing == CameraInfo.CAMERA_FACING_BACK) {

            defaultCameraId = i;

        }

    }

```

Other new camera-related methods

include `getNumberOfCameras()` and `getCameraInfo()`. If you look at the previous code snippet, you can see how these methods can be used to query an application for the number of cameras available and find the camera an application needs.

Another new method, `get()`, allows you to programmatically retrieve information about a specific camera as a `CamcorderProfile`. This lets your applications be more flexible because they can get information about a device's capabilities, and you have to write less code targeting certain hardware platforms. Android 3.0 includes a number of other new methods related to cameras, so be sure to review the Camera API at <http://mng.bz/v1sw>. That being said, if you wish to work with cameras, you'll most likely have to work directly with the hardware on which you wish to run your applications.

Debugging Video Apps

Without a device to test on, you'll have major difficulties debugging your video applications. This is especially true with the Android SDK emulator for Xoom-like tablets, which is difficult to use due to its extremely poor performance. If you decide to develop a video application, we strongly suggest that you not only obtain an Android device to test on, but also test every physical device that you hope your application will run on. Although Android applications that record data from sensors can be difficult to work with on the emulator, they're relatively straightforward to code—but you need to use a physical Android device to test.

10.5. SUMMARY

In this chapter, we looked at how the Android SDK supports multimedia and how you can play, save, and record video and audio. We also discussed various features the Android `MediaPlayer` offers developers, from a built-in video player to wide support for formats, encodings, and standards.

We explained how to interact with other hardware devices attached to the phone, such as a microphone and camera. You used the `MediaRecorder` application to record audio and video and then save it to the SD card.

The most consistent characteristic of multimedia programming with Android is that things are changing and maturing! Multimedia support has moved from OpenCORE to Stagefright as of Android 3.0. Writing multimedia applications requires you to conduct a bit more work directly on the hardware you wish an application to work on, because the emulated environments don't adequately replicate the hardware capabilities of the handsets. Despite this potential speed bump in the development process, Android currently offers everything you need to create rich and compelling media applications.

In the next chapter, you'll learn how to use Android's location services to interact with GPS and maps. By mixing in what you've learned in this chapter, you'll be able to create your own GPS application that not only provides voice direction, but also responds to voice commands.

Chapter 11. Location, location, location

This chapter covers

- Working with `LocationProvider` and `LocationManager`
- Testing location in the emulator
- Receiving location alerts with `LocationListener`
- Drawing with `MapActivity` and `MapView`
- Looking up addresses with the `Geocoder`

Accurate location awareness makes a mobile device more powerful. Combining location awareness with network data can change the world—and Android shines here. Other platforms have gained similar abilities in recent years, but Android excels with its easy-to-use and popular location API framework based on Google Maps.

From direct network queries to triangulation with cell towers and even satellite positioning via GPS, an Android-powered device has access to different types of `LocationProvider` classes that allow access to location data. Various providers supply a mix of location-related metrics, including latitude and longitude, speed, bearing, and altitude.

Developers generally prefer to work with GPS because of its accuracy and power. But some devices may not have a GPS receiver, and even GPS-enabled devices can't access satellite data when inside a large building or otherwise obstructed from receiving the signal. In those instances the Android platform provides a graceful and automatic fallback to query other providers when your first choice fails. You can examine provider availability and hook into one or another using the `LocationManager` class.

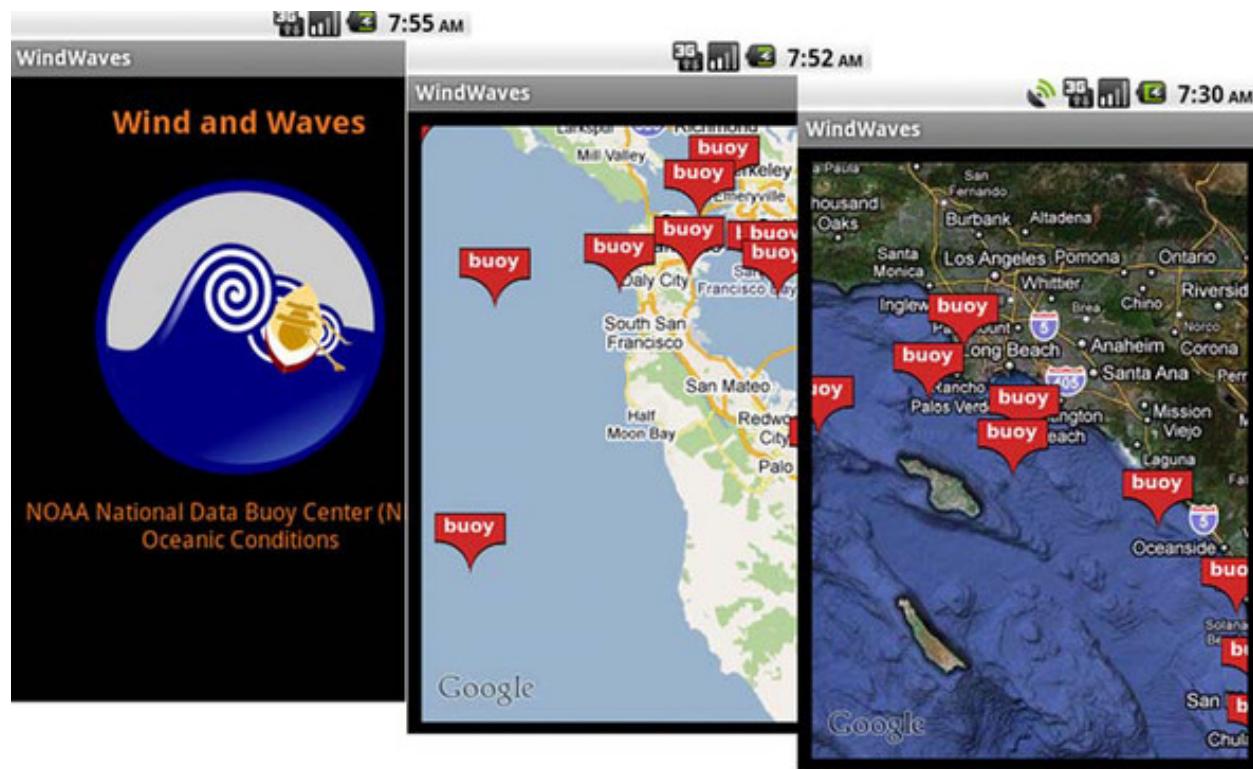
Location awareness¹ opens up a new world of possibilities for application development. In this chapter, you'll build an application that combines location awareness with data from the U.S. National Oceanic and Atmospheric Administration (NOAA) to produce an interesting and useful mashup.

¹ For more about location, check out *Location-Aware Applications* by Richard Ferraro and Murat Aktihanoglu, published by Manning in July 2011: www.manning.com/ferraro.

Specifically, you'll connect to the National Data Buoy Center (NDBC) to retrieve data from buoys and ships located around the coastline in North America. Thanks to the NOAA-NDBC system, which polls sensors on buoys and makes that data available in RSS feeds, you can retrieve data for the vicinity, based on the current location, and

display condition information such as wind speed, wave height, and temperature. Although we won't cover non-location-related details in this chapter, such as using HTTP to pull the RSS feed data, the full source code for the application is available with the code download for this chapter. Our Wind and Waves application has several main screens, including an Android `MapActivity` with a `MapView`. These components are used for displaying and manipulating map information, as shown in [figure 11.1](#).

Figure 11.1. Screens from the Wind and Waves location-aware application



Accessing buoy data, which is important mainly for marine use cases, has a somewhat limited audience. But the principles shown in this app demonstrate the range of Android's location-related capabilities and should inspire you to develop your own unique application.

In addition to displaying data based on the current location, you'll use this application to create several `LocationListener` instances that receive updates when the user's location changes. When the position changes, the device will inform your application, and you'll update your `MapView` using an `Overlay`—an object that allows you to draw on top of the map.

Beyond the buoy application requirements, you'll also write a few samples for working with the `Geocoder` class. This class allows you to map between a `GeoPoint` (latitude and longitude) and a place (city or postal code) or address. This utility doesn't help much on the high seas but does benefit many other apps.

Before writing the sample apps, you'll start by using the device's built-in mapping application and simulating your position within the Android emulator. This approach will allow you to mock your location for the emulator. After we've covered all of the emulator location-related options, we'll move on to building Wind and Waves.

11.1. SIMULATING YOUR LOCATION WITHIN THE EMULATOR

For any location-aware application, you'll start by working with the provided SDK and the emulator. Within the emulator, you'll set and update your current location. From there you'll want to progress to supplying a range of locations and times to simulate movement over a geographic area.

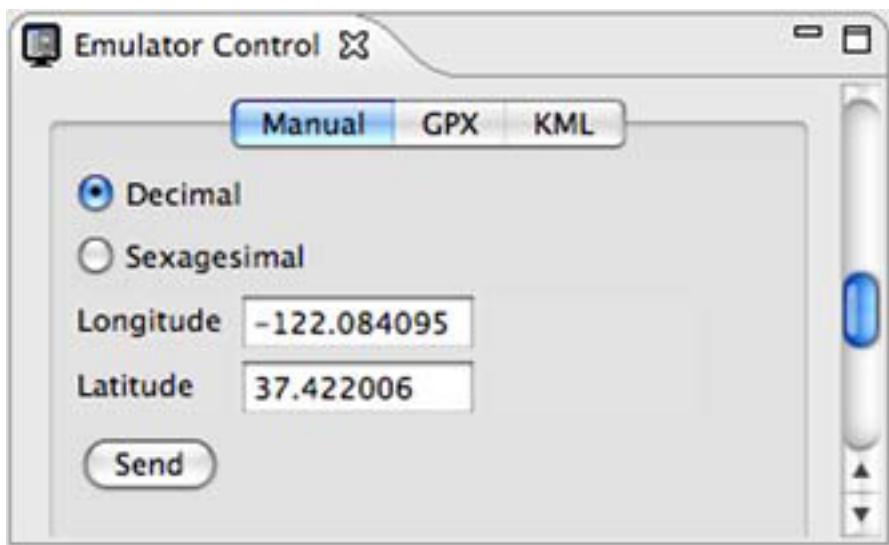
You can accomplish these tasks in several ways, either by using the DDMS tool or by using the command line from the shell. To get started quickly, let's first send in direct coordinates through the DDMS tool.

11.1.1. Sending in your coordinates with the DDMS tool

You can access the DDMS tool in two ways, either launched on its own from the SDK tools subdirectory or as the Emulator Control view within the Eclipse IDE. You need to have Eclipse and the Android Eclipse plug-in to use DDMS within Eclipse; see [chapter 2](#) and [appendix A](#) for more details about getting the SDK and plug-in set up.

With the DDMS tool you can send direct latitude and longitude coordinates manually from the Emulator Control > Location Controls form. This is shown in [figure 11.2](#). Note that *Longitude* is the first field, which is the standard around the world, but backward from how latitude and longitude are generally expressed in the United States.

Figure 11.2. Using the DDMS tool to send direct latitude and longitude coordinates to the emulator as a mock location



If you launch the built-in Maps application from Android's main menu and send in a location with the DDMS tool, you can then use the menu to select My Location, and the map will animate to the location you've specified—anywhere on Earth.

Note

Both the Google Maps application and the mapping APIs are part of the optional Google APIs. As such, not all Android phones support these features. Check your target devices to ensure that they provide this support. For development, you'll need to install an Android Virtual Device² (AVD) that supports the Google APIs.

² For more on Android, maps and Android Virtual Devices, try here: <http://developer.appcelerator.com/doc/mobile/android-maps>.

Try this a few times to become comfortable with setting locations; for example, send the decimal coordinates in [table 11.1](#) one by one, and in between browse around the map. When you supply coordinates to the emulator, you'll need to use the decimal form.

Table 11.1. Example coordinates for the emulator to set using the DDMS tool

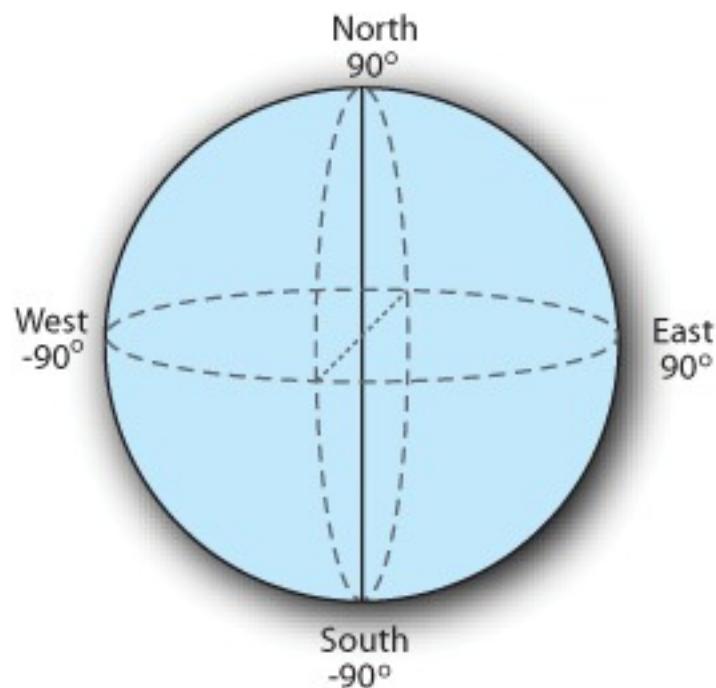
Description	Latitude degrees	Longitude degrees	Latitude decimal	Longitude decimal
Golden Gate Bridge, California	37°49' N	122°29' W	37.82	-122.48

Description	Latitude degrees	Longitude degrees	Latitude decimal	Longitude decimal
Mount Everest, Nepal	27°59' N	86°56' E	27.98	86.93
Ayer's Rock, Australia	25°23' S	131°05' E	-25.38	131.08
North Pole	90°00' N		90.00	
South Pole	90°00' S		-90.00	

Although the DDMS tool requires the decimal format, latitude and longitude are more commonly expressed on maps and other tools as degrees, minutes, and seconds. Degrees ($^{\circ}$) represent points on the surface of the globe as measured from either the equator (for latitude) or the prime meridian (for longitude). Each degree is further subdivided into 60 smaller sections, called minutes ('), and each minute also has 60 seconds ("). If necessary, seconds can be divided into tenths of a second or smaller fractions.

When representing latitude and longitude on a computer, the degrees are usually converted into decimal form with positive representing north and east and negative representing south and west, as shown in [figure 11.3](#).

Figure 11.3. Latitude and longitude spherical diagram, showing positive north and east and negative south and west



If you live in the southern and eastern hemispheres, such as in Buenos Aires, Argentina, which is 34°60' S, 58°40' W in the degree form, the decimal form is negative for both

latitude and longitude, -34.60, -58.40. If you haven't used latitude and longitude much, the different forms can be confusing at first, but they quickly become clear.

Once you've mastered setting a fixed position, you can move on to supplying a set of coordinates that the emulator will use to simulate a range of movement.



Note

You can also send direct coordinates from within the emulator console. If you telnet localhost 5554 (adjust the port where necessary) or adb shell, you'll connect to the default emulator's console. From there you can use the `geo fix` command to send longitude, latitude, and optional altitude; for example, `geo fix -21.55 64.1`. Keep in mind that the Android tools require longitude in the first parameter.



11.1.2. The GPS Exchange Format

The DDMS tool supports two formats for supplying a range of location data in file form to the emulator. The GPS Exchange Format (GPX) allows a more expressive form when working with Android.

GPX is an XML schema that allows you to store waypoints, tracks, and routes. Many handheld GPS devices support this format. The following listing shows the basics of the format in a portion of a sample GPX file.

Listing 11.1. A sample GPX file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1"
    version="1.1"
    creator="Charlie Collins - Hand Rolled"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation=
        "http://www.topografix.com/GPX/1/1/gpx.xsd">
    <metadata>
        <name>Sample Coastal California Waypoints</name>
        <desc>Test waypoints for use with Android</desc>
        <time>2008-11-25T06:52:56Z</time>
        <bounds minlat="25.00" maxlat="75.00"
            minlon="100.00" maxlon="-150.00" />
    </metadata>
    <wpt lat="41.85" lon="-124.38">
        <ele>0</ele>
        <name>Station 46027</name>
        <desc>Off the coast of Lake Earl</desc>
    </wpt>
    <wpt lat="41.74" lon="-124.18">
        <ele>0</ele>
        <name>Station CECC1</name>
        <desc>Crescent City</desc>
    </wpt>
```

1 Define root gpx element

2 Include metadata stanza

3 Supply waypoint element

```

<wpt lat="38.95" lon="-123.74">
    <ele>0</ele>
    <name>Station PTAC1</name>
    <desc>Point Arena Lighthouse</desc>
</wpt>
    . . . remainder of wpts omitted for brevity
<trk>
    <name>Example Track</name>
    <desc>A fine track with trkpts.</desc>
    <trkseg>
        <trkpt lat="41.85" lon="-124.38">
            <ele>0</ele>
            <time>2008-10-15T06:00:00Z</time>
        </trkpt>
        <trkpt lat="41.74" lon="-124.18">
            <ele>0</ele>
            <time>2008-10-15T06:01:00Z</time>
        </trkpt>
        <trkpt lat="38.95" lon="-123.74">
            <ele>0</ele>
            <time>2008-10-15T06:02:00Z</time>
        </trkpt>
        . . . remainder of trkpts omitted for brevity
    </trkseg>
</trk>
</gpx>

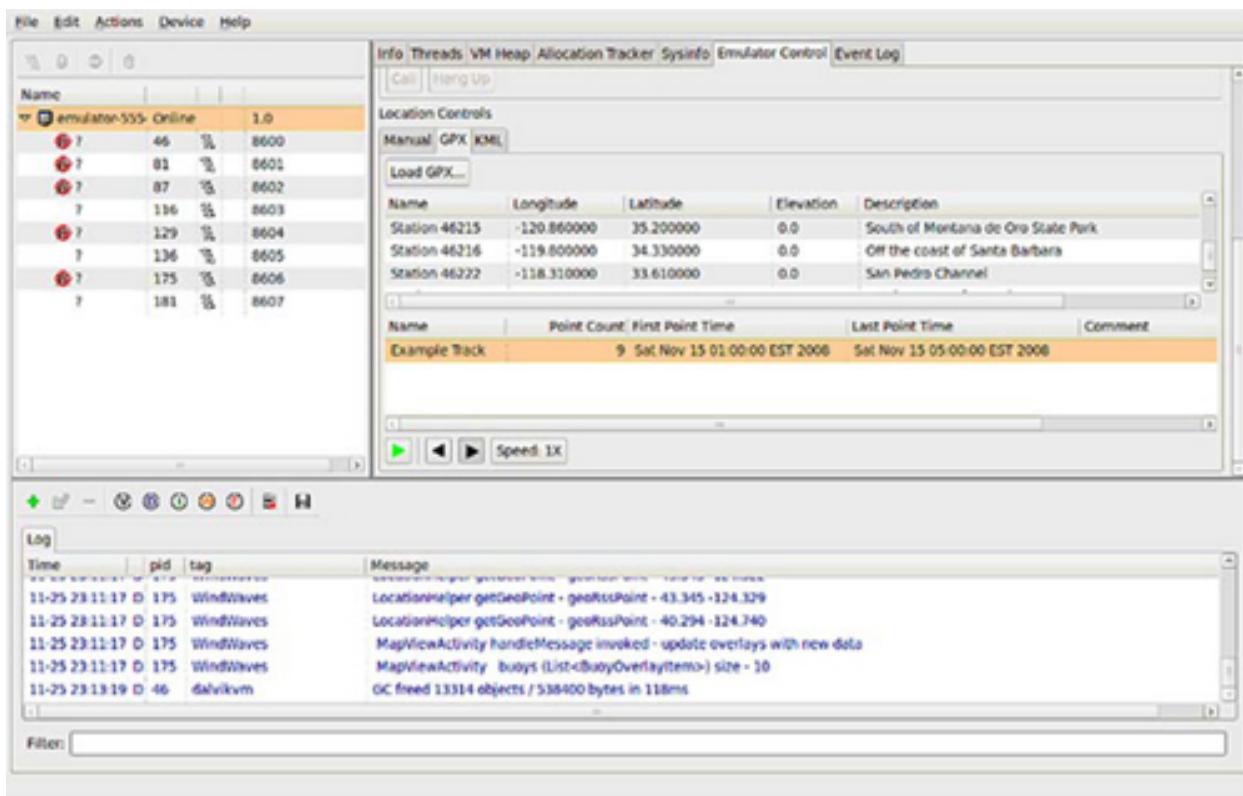
```

- 4 Supply track element
- 5 Use track segment
- 6 Provide specific point

A GPX file requires the correct XML namespace in the root `gpx` element ①. Within its body, the file includes metadata ② and individual waypoints ③. *Waypoints* are named locations at a particular latitude and longitude. Along with individual waypoints, a GPX file supports related route information in the form of *tracks* ④, which can be subdivided further into *track segments* ⑤. Each track segment is made up of *track points*. Finally, each track point ⑥ contains a waypoint with an additional point-in-time property.

When working with a GPX file in the DDMS tool, you can use two different modes, as [figure 11.4](#) reveals. The top half of the GPX box lists individual waypoints; when you click one, that individual location is sent to the emulator. In the bottom half of the GPX box, all the tracks are displayed. Tracks can be “played” forward and backward to simulate movement. As the track reaches each track point, based on the time it defines, it sends those coordinates to the emulator. You can modify the speed for this playback via the Speed button.

Figure 11.4. Using the DDMS tool with a GPX file to send mock location information



GPX is simple and extremely useful when working with mock location information for your Android applications, but it's not the only file format supported. The DDMS tool also supports a format called KML.

11.1.3. The Google Earth Keyhole Markup Language

The second format that the Android DDMS tool supports for sending a range of mock location information to the emulator is the *Keyhole Markup Language (KML)*. KML was originally a proprietary format created by a company named Keyhole. After Google acquired Keyhole, it submitted KML to the Open Geospatial Consortium (OGC), which accepted KML as an international standard.

OGC KML pursues the following goal:

That there be one international standard language for expressing geographic annotation and visualization on existing or future web-based online and mobile maps (2d) and earth browsers (3d).

The following listing shows a sample KML file for sending location data to the Android emulator. This file uses the same coastal location data as you saw with the previous GPX example.

Listing 11.2. A sample KML file

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
    <Placemark>
        <name>Station 46027</name>
        <description>Off the coast of Lake Earl</description>
        <Point>
            <coordinates>-124.38,41.85,0</coordinates>
        </Point>
    </Placemark>
    <Placemark>
        <name>Station 46020</name>
        <description>Outside the Golden Gate</description>
        <Point>
            <coordinates>-122.83,37.75,0</coordinates>
        </Point>
    </Placemark>
    <Placemark>
        <name>Station 46222</name>
        <description>San Pedro Channel</description>
        <Point>
            <coordinates>-118.31,33.61,0</coordinates>
        </Point>
    </Placemark>
</kml>
```

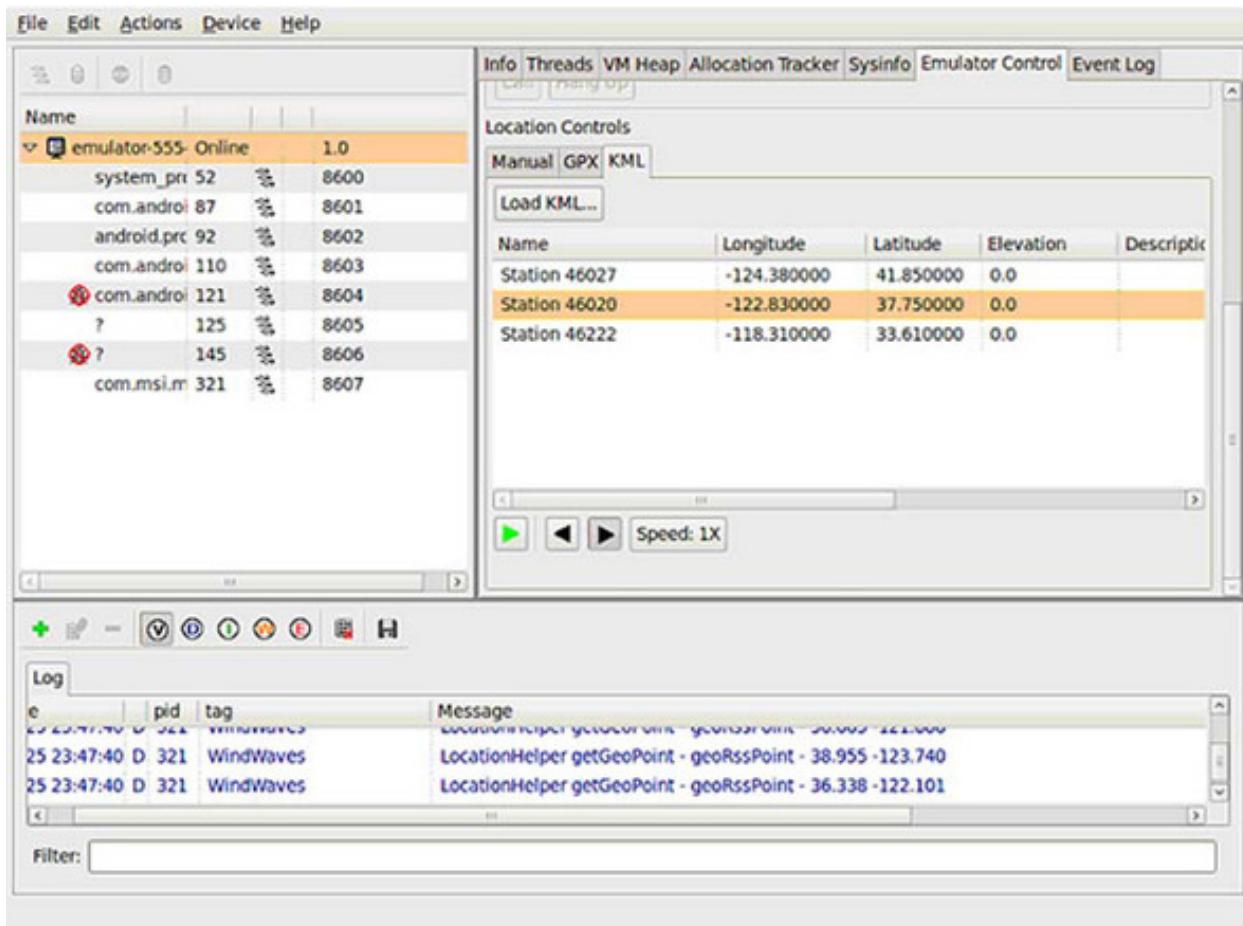
The diagram illustrates the structure of the KML file with three numbered callouts:

- 1 Capture information with Placemark**: Points to the first `<Placemark>` element.
- 2 Provide Point**: Points to the `<Point>` element nested within the first `<Placemark>`.
- 3 Supply coordinates for Point**: Points to the `<coordinates>` element within the `<Point>` element.

KML uses a `kml` root element requiring the correct namespace declaration. KML supports many more elements and attributes than the DDMS tool handles. DDMS only checks your KML files for `Placemark` elements ①, which contain `Point` child elements ②, which in turn supply coordinates ③.

Figure 11.5 shows an example of using a KML file with the DDMS tool.

Figure 11.5. Using the DDMS tool with a KML file to send mock location information



KML³ is flexible and expressive, but it has drawbacks when used with the Android emulator. As we've noted, the DDMS parser looks for the `coordinate` elements in the file and sends the latitude, longitude, and elevation for each in a sequence, one `Placemark` per second. Timing and other advanced features of KML aren't yet supported by DDMS. Because of this, we find it more valuable at present to use GPX as a debugging and testing format, because it supports detailed timing.

³ For more details on KML, go to <http://code.google.com/apis/kml/documentation/>.

KML is still important; it's an international standard and will continue to gain traction. Also, KML is an important format for other Google applications, so you may encounter it more frequently in other contexts than GPX. For example, you could create a KML route using Google Earth, and then later use it in your emulator to simulate movement.

Now that you know how to send mock location information to the emulator in various formats, you can step out of the built-in Maps application and start creating your own programs that rely on location.

11.2. USING LOCATIONMANAGER AND LOCATIONPROVIDER

When building location-aware applications on the Android platform, you'll most often use several key classes. A `LocationProvider` provides location data using several metrics, and you can access providers through a `LocationManager`.

`LocationManager` allows you to attach a `LocationListener` that receives updates when the device location changes. `LocationManager` also can directly fire an `Intent` based on the proximity to a specified latitude and longitude. You can always retrieve the last-known `Location` directly from the manager.

The `Location` class is a Java bean that represents all the location data available from a particular snapshot in time. Depending on the provider used to populate it, a `Location` may or may not have all the possible data present; for example, it might not include speed or altitude.

To get your Wind and Waves sample application started and to grasp the related concepts, you first need to master the `LocationManager`.

11.2.1. Accessing location data with LocationManager

`LocationManager` lets you retrieve location-related data on Android. Before you can check which providers are available or query the last-known `Location`, you need to acquire the manager from the system service. The following listing demonstrates this task and includes a portion of the `MapViewActivity` that will drive our Wind and Waves application.

Listing 11.3. Start of MapViewActivity

```
public class MapViewActivity extends MapActivity {  
    private static final int MENU_SET_SATELLITE = 1;  
    private static final int MENU_SET_MAP = 2;  
    private static final int MENU_BUOYS_FROM_MAP_CENTER = 3;  
    private static final int MENU_BACK_TO_LAST_LOCATION = 4;  
    . . . Handler and LocationListeners omitted here for brevity - shown in  
        later listings  
    private MapController mapController;  
    private LocationManager locationManager;  
    private LocationProvider locationProvider;  
    private MapView mapView;  
    private ViewGroup zoom;  
    private Overlay buoyOverlay;  
    private ProgressDialog progressDialog;  
    private Drawable defaultMarker;  
    private ArrayList<BuoyOverlayItem> buoys;  
    @Override  
    public void onCreate(Bundle icicle) {  
        super.onCreate(icicle);  
        setContentView(R.layout.mapview_activity);  
        mapView = (MapView) this.findViewById(R.id.map_view);  
        zoom = (ViewGroup) findViewById(R.id.zoom);  
        zoom.addView(this.mapView.getZoomControls());  
        defaultMarker =  
            getResources().getDrawable(R.drawable.redpin);  
        defaultMarker.setBounds(0, 0,  
            defaultMarker.getIntrinsicWidth(),  
            defaultMarker.getIntrinsicHeight());  
        buoys = new ArrayList<BuoyOverlayItem>();  
    }  
}
```

1 Extend
MapActivity

2 Define
LocationManager

3 Define
LocationProvider

```

@Override
public void onStart() {
    super.onStart();
    locationManager = (LocationManager)
        getSystemService(Context.LOCATION_SERVICE);
    locationProvider =
        locationManager.getProvider(
            LocationManager.GPS_PROVIDER);
    // LocationListeners omitted here for brevity
    GeoPoint lastKnownPoint = this.getLastKnownPoint();
    mapController = this.mapView.getController();
    mapController.setZoom(10);
    mapController.animateTo(lastKnownPoint);
    getBuoyData(lastKnownPoint);
}

. . . onResume and onPause omitted for brevity
. . . other portions of MapViewActivity are included
     in later listings in this chapter
private GeoPoint getLastKnownPoint() {
    GeoPoint lastKnownPoint = null;
    Location lastKnownLocation =
        locationManager.getLastKnownLocation(
            LocationManager.GPS_PROVIDER);
    if (lastKnownLocation != null) {
        lastKnownPoint = LocationHelper.getGeoPoint(lastKnownLocation);
    } else {
        lastKnownPoint = LocationHelper.GOLDEN_GATE;
    }
    return lastKnownPoint;
}

```

Our custom `MapViewActivity` extends `MapActivity` ①. We'll focus on the `MapActivity` in more detail in [section 11.3](#), but for now, recognize that this is a special kind of `Activity`. Within the class, you declare member variables

for `LocationManager` ② and `LocationProvider` ③.

To acquire the `LocationManager`, you use the `Activity getSystemService (String name)` method ④. Once you have the `LocationManager`, you assign

the `LocationProvider` you want to use with the manager's `getProvider()` method ⑤. In this case, use the GPS provider. We'll talk more about the `LocationProvider` class in the next section.

Once you have the manager and provider in place, you implement the `onCreate()` method of your `Activity` to instantiate a `MapController` and set the

initial state for the screen 6. Section 11.3 covers `MapController` and the `MapView` it manipulates.

Along with helping you set up the provider you need, `LocationManager` supplies quick access to the last-known `Location` 7. Use this method if you need a quick fix on the last location, as opposed to the more involved techniques for registering for periodic location updates with a listener; we'll cover that topic in [section 11.2.3](#).

Besides the features shown in this listing, `LocationManager` allows you to directly register for proximity alerts. For example, your app could show a custom message if you pass within a quarter-mile of a store that has a special sale. If you need to fire an `Intent` based on proximity to a defined location, call the `addProximityAlert()` method. This method lets you set the target location with latitude and longitude, and also lets you specify a radius and a `PendingIntent`. If the device comes within the range, the `PendingIntent` is fired. To stop receiving these messages, call `removeProximityAlert()`.

Getting back to the main purpose for which you'll use the `LocationManager` with Wind and Waves, we'll next look more closely at the GPS `LocationProvider`.

11.2.2. Using a LocationProvider

`LocationProvider` helps define the capabilities of a given provider implementation. Each implementation responsible for returning location information may be available on different devices and in different circumstances.

Available provider implementations depend on the hardware capabilities of the device, such as the presence of a GPS receiver. They also depend on the situation: even if the device has a GPS receiver, can it currently receive data from satellites, or is the user somewhere inaccessible such as an elevator or a tunnel?

At runtime, you'll query for the list of providers available and use the most suitable one. You may select multiple providers to fall back on if your first choice isn't available or enabled. Developers generally prefer using the `LocationManager.GPS_PROVIDER` provider, which uses the GPS receiver. You'll use this provider for Wind and Waves because of its accuracy and its support in the emulator. Keep in mind that a real device will normally offer multiple providers, including the `LocationManager.NETWORK_PROVIDER`, which uses cell tower and Wi-Fi access points to determine location data. To piggyback on other applications requesting location, use `LocationManager.PASSIVE_PROVIDER`.

In [listing 11.3](#), we showed how you can obtain the GPS provider directly using the `getProvider(String name)` method. [Table 11.2](#) provides alternatives to this approach of directly accessing a particular provider.

Table 11.2. Methods for obtaining a `LocationProvider` reference

LocationProvider code snippet	Description
<code>List<String> providers = locationManager.getAllProviders();</code>	Get all of the providers registered on the device.
<code>List<String> enabledProviders = locationManager.getAllProviders(true);</code>	Get all of the currently enabled providers.
<code>locationProvider = locationManager.getProviders(true).get(0);</code>	A shortcut to get the first enabled provider, regardless of type.
<code>locationProvider = locationManager.getBestProvider(myCriteria, true);</code>	An example of getting a LocationProvider using a particular Criteria argument. You can create a Criteria instance and specify whether bearing, altitude, cost, and other metrics are required.

Different providers may support different location-related metrics and have different costs or capabilities. The `Criteria` class helps define what each provider instance can handle. Available metrics are latitude and longitude, speed, bearing, altitude, cost, and power requirements.

Remember to set the appropriate Android permissions. Your manifest needs to include location-related permissions for the providers you want to use. The following listing shows the Wind and Waves manifest XML file, which includes both `COARSE-` and `FINE-` grained location-related permissions.

Listing 11.4. A manifest file showing COARSE and FINE location-related permissions

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.windwaves">
    <application android:icon="@drawable/wave_45"
        android:label="@string/app_name"
        android:theme="@android:style/Theme.Black">
        <activity android:name="StartActivity"
            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="MapViewActivity" />
        <activity android:name="BuoyDetailActivity" />
        <uses-library android:name="com.google.android.maps" />
    </application>
    <uses-permission
        android:name=
            "android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission
        android:name=
            "android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission
        android:name="android.permission.INTERNET" />
</manifest>

```

Include both

the `ACCESS_COARSE_LOCATION` ① and `ACCESS_FINE_LOCATION` ② permissions in your manifest. The `COARSE` permission corresponds to the `LocationManager.NETWORK_PROVIDER` provider for cell and Wi-Fi based data, and the `FINE` permission corresponds to the `LocationManager.GPS_PROVIDER` provider. You don't use the network provider in Wind and Waves, but this permission would allow you to enhance the app to fall back to the network provider if the GPS provider becomes unavailable or disabled.

Once you understand the basics of `LocationManager` and `LocationProvider`, you can unleash the real power and register for periodic location updates in your application with the `LocationListener` class.

11.2.3. Receiving location updates with LocationListener

You can keep abreast of the device location by creating a `LocationListener` implementation and registering it to receive updates. `LocationListener` lets you filter for many types of location events based on a

flexible and powerful set of properties. You implement the interface and register your instance to receive location data callbacks.

[Listing 11.5](#) demonstrates those principles as you create several `LocationListener` implementations for the Wind and Waves `MapViewActivity` and then register those listeners using the `LocationManager` and `LocationProvider`. This listing helps complete the initial code from [listing 11.3](#).

Listing 11.5. Creation of `LocationListener` implementations in `MapViewActivity`

```
 . . . start of class in Listing 11.3
private final LocationListener locationListenerGetBuoyData =
    new LocationListener() {
        public void onLocationChanged(
            final Location loc) {
            int lat = (int) (loc.getLatitude()
                * LocationHelper.MILLION);
            int lon = (int) (loc.getLongitude()
                * LocationHelper.MILLION);
            GeoPoint geoPoint = new GeoPoint(lat, lon);
            getBuoyData(geoPoint);
        }
        public void onProviderDisabled(String s) {
        }
        public void onProviderEnabled(String s) {
        }
        public void onStatusChanged(String s,
            int i, Bundle b) {
        }
    };
private final LocationListener locationListenerRecenterMap =
    new LocationListener() {
        public void onLocationChanged(final Location loc) {
            int lat = (int) (loc.getLatitude()
                * LocationHelper.MILLION);
            int lon = (int) (loc.getLongitude()
                * LocationHelper.MILLION);
            GeoPoint geoPoint = new GeoPoint(lat, lon);
            mapController.animateTo(geoPoint);
        }
    }
```

The diagram illustrates the flow of logic in Listing 11.5. It starts with the creation of an anonymous `LocationListener` (Step 1). This listener implements the `onLocationChanged` method (Step 2). Inside this method, it retrieves the latitude and longitude from the `Location` object (Step 3). It then creates a `GeoPoint` using these coordinates (Step 4). Finally, it calls the `getBuoyData` method with this `GeoPoint` (Step 5). The `onLocationChanged` method also handles provider changes and status changes. Additionally, there is another `LocationListener` for recentering the map, which follows a similar pattern of implementation and data processing (Step 6).

```

public void onProviderDisabled(String s) {
}
public void onProviderEnabled(String s) {
}
public void onStatusChanged(String s,
    int i, Bundle b) {
}
};

@Override
public void onStart() {
    super.onStart();
    locationManager =
        (LocationManager)
            getSystemService(Context.LOCATION_SERVICE);
    locationProvider =
        locationManager.getProvider(LocationManager.GPS_PROVIDER);
    if (locationProvider != null) {
        locationManager.requestLocationUpdates(
            locationProvider.getName(), 3000, 185000,
            locationListenerGetBuoyData);
        locationManager.requestLocationUpdates(
            locationProvider.getName(), 3000, 1000,
            locationListenerRecenterMap);
    } else {
        Toast.makeText(this, "Wind and Waves cannot continue,"
            + " the GPS location provider is not available"
            + " at this time.", Toast.LENGTH_SHORT).show();
        finish();
    }
    . . . remainder of repeated code omitted (see listing 11.3)
}

```

Methods intentionally left blank

7 Register locationListener-GetBuoyData

8 Register locationListener-RecenterMap

You'll usually find it practical to use an anonymous inner class **1** to implement the `LocationListener` interface. For this `MapViewActivity`, we create two `LocationListener` implementations so we can later register them using different settings.

The first listener, `locationListenerGetBuoyData`, implements the `onLocationChanged` method **2**. In that method we get the latitude and longitude from the `Location`s sent in the callback **3**. We then use the data to create a `GeoPoint` **4** after multiplying the latitude and longitude by 1 million (1e6). You need to multiply by a million because `GeoPoint` requires microdegrees for coordinates. A separate class, `LocationHelper`, defines this constant and provides other location utilities; you can view this class in the code download for this chapter.

After we have the data, we update the map **5** using a helper method that resets a map `Overlay`; you'll see this method's implementation in the next section. In the second listener `locationListenerRecenterMap`, we perform the different task of centering the map **6**.

The need for two separate listeners becomes clear when you see how listeners are registered with the `requestLocationUpdates()` method of the `LocationManager` class. We register the first listener, `locationListenerGetBuoyData`, to fire only when the new

device location has moved a long way from the previous one **7**. The defined distance is 185,000 meters. (We chose this number to stay just under 100 nautical miles, which is the radius you'll use to pull buoy data for your map; you don't need to redraw the buoy data on the map if the user moves less than 100 nautical miles.) We register the second listener, `locationListenerRecenterMap`, to fire more frequently; the map view recenters

if the user moves more than 1,000 meters **8**. Using separate listeners like this allows you to fine-tune the event processing, rather than having to build in your own logic to do different things based on different values with one listener.

Keep in mind that your registration of `LocationListener` instances could become even more robust by implementing the `onProviderEnabled()` and `onProviderDisabled()` methods. Using those methods and different providers, you could provide useful messages to the user and also provide a graceful fallback through a set of providers; for example, if GPS becomes disabled, you could try the network provider instead.

Note

You should use the `time` parameter to the `requestLocationUpdates()` method carefully. Requesting location updates too frequently (less than 60,000 ms per the documentation) can wear down the battery and make the application too jittery. In this sample, you use an extremely low value (3,000 ms) for debugging purposes. Long-lived or always-running code shouldn't use a value lower than the recommended 60,000 ms in production code.

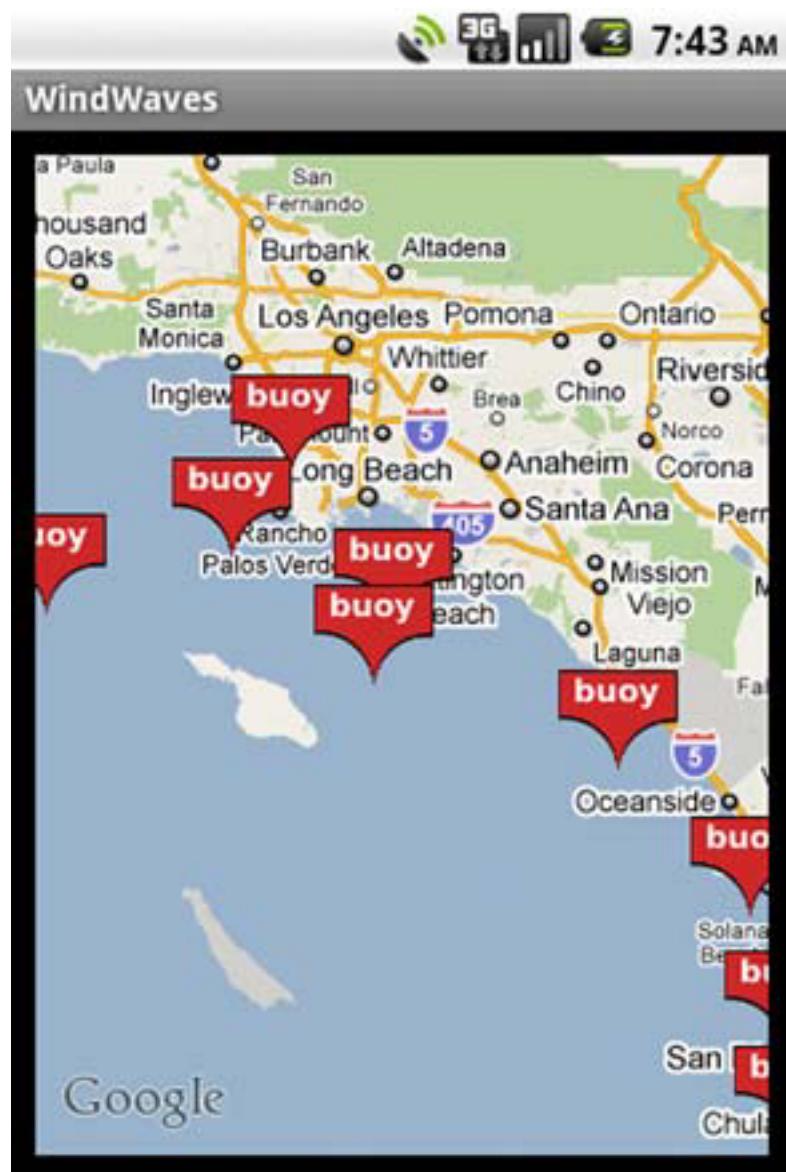
With `LocationManager`, `LocationProvider`, and `LocationListener` instances in place, we can address the `MapActivity` and `MapView` in more detail.

11.3. WORKING WITH MAPS

In the previous sections, you wrote the start of the `MapViewActivity` for the Wind and Waves application. We covered the supporting classes and showed you how to register to receive location updates. With that structure in place, let's now focus on the actual map details.

The `MapViewActivity` screen will look like [figure 11.6](#), where several map `Overlay` classes display on top of a `MapView` within a `MapActivity`.

Figure 11.6. The `MapViewActivity` from the Wind and Waves application shows a `MapActivity` with a `MapView`.



To use the `com.google.android.maps` package on the Android platform and support all the features related to a `MapView`, you must use a `MapActivity`.

11.3.1. Extending MapActivity

A `MapActivity` defines a gateway to the Android Google Maps-like API package and other useful map-related utilities. It handles several details behind creating and using a `MapView` so you don't have to worry about them.

The `MapView`, covered in the next section, offers the most important features. But a `MapActivity` provides essential support for the `MapView`. It manages all the network- and filesystem-intensive setup and teardown tasks needed for supporting the map. For example, the `MapActivity onResume()` method automatically sets up network threads for various map-related tasks and caches map section tile data on the filesystem, and the `onPause()` method cleans up these resources. Without this class, all these details would require extra housekeeping that any `Activity` wishing to include a `MapView` would have to repeat each time.

Your code won't do much with `MapActivity`. Extend this class (as in [listing 11.3](#)), making sure to use only one instance per process, and include a special manifest element to enable the `com.google.android.maps` package. You may have noticed the `uses-library` element in the Wind and Waves manifest in [listing 11.4](#):

```
<uses-library android:name="com.google.android.maps" />
```

The `com.google.android.maps` package, where `MapActivity`, `MapView`, `MapController`, and other related classes such as `GeoPoint` and `Overlay` reside, isn't a standard package in the Android library. This manifest element pulls in support for the Google `maps` package.

Once you include the `uses-library` element and write a basic `Activity` that extends `MapActivity`, you can start writing the main app features with a `MapView` and related `Overlay` classes.

11.3.2. Using a MapView

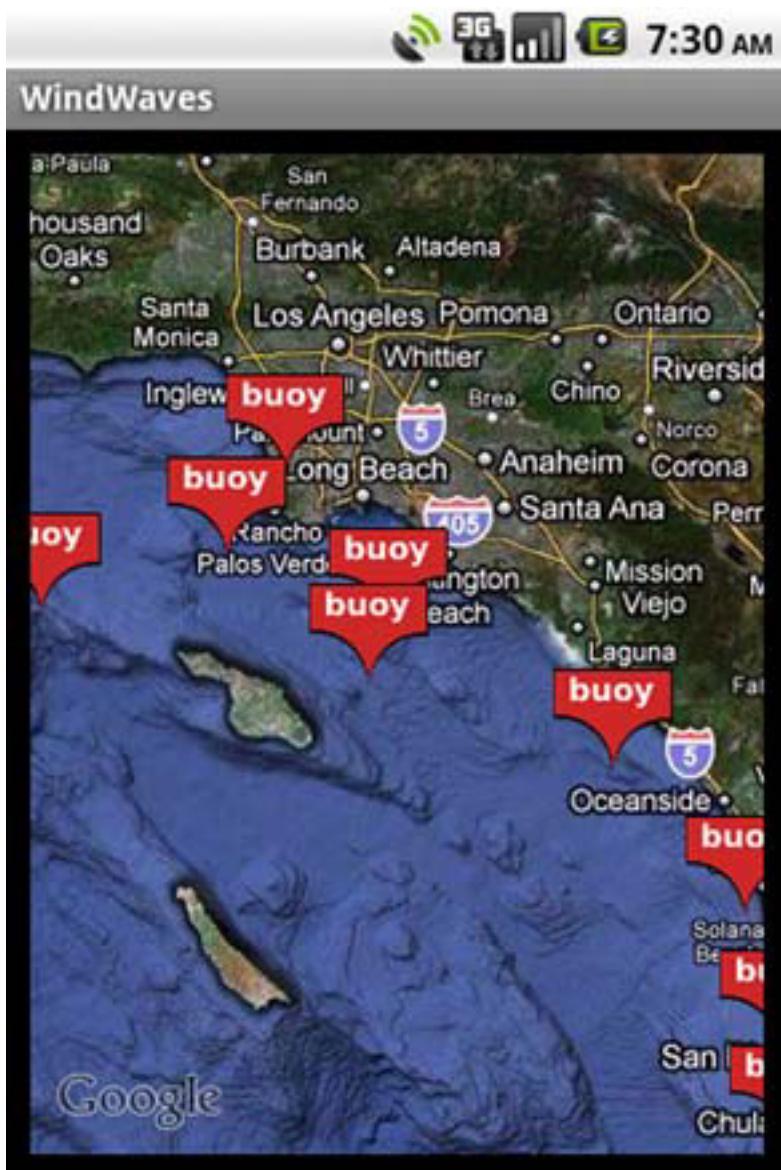
Android offers `MapView`⁴ as a limited version of the Google Maps API in the form of a `View` for your Android application. A `MapView` displays tiles of a map, which it obtains over the network as the map moves and zooms, much like the web version of Google Maps.

⁴ Take a look at this `MapView` tutorial for more information: <http://developer.android.com/guide/tutorials/views/hello-mapview.html>.

Android supports many of the concepts from the standard Google Maps API through the `MapView`. For instance, `MapView` supports a plain map mode, a satellite mode, a street-view mode, and a traffic mode. When you want to write something on top of the map, draw a straight line between two points, drop a “pushpin” marker, or display full-sized images, you use an `Overlay`.

You can see examples of several of these concepts in [figure 11.6](#), which shows `MapViewActivity` screenshots for the Wind and Waves application. [Figure 11.7](#) shows that same `MapViewActivity` again after switching into satellite mode.

Figure 11.7. The `MapViewActivity` from the Wind and Waves application using satellite mode



You've already seen the `MapView` we'll use for the Wind and Waves application declared and instantiated in [listing 11.3](#). Now we'll discuss using this class inside your `Activity` to control, position, zoom, populate, and overlay your map.

Before you can use a map at all, you have to request a Google Maps API key and declare it in your layout file. This listing shows the `MapActivity` layout file you'll use with a special `android:apiKey` attribute.

Listing 11.6. A `MapView` layout file including the Google Maps API key

```
<?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"
    android:gravity="center_horizontal" android:padding="10px">
    <com.google.android.maps.MapView
        android:id="@+id/map_view"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:enabled="true"
        android:clickable="true"
        android:apiKey=
            "051Sygx-ttd-J5GXfsIB-dlpNtggca4I4DMyVqQ" />
</RelativeLayout>
```

1 Define MapView element in XML
2 Include apiKey attribute

You can declare a `MapView` in XML just like other `View` components 1. In order to use the Google Maps network resources, a `MapView` requires an API key 2. You can obtain a map key from the Google Maps Android key registration web page at <http://code.google.com/android/maps-api-signup.html>.

Before you register for a key, you need to look up the MD5 fingerprint of the certificate that signs your application. This sounds tricky, but it's really simple. When using the Android emulator, the SDK always uses a Debug Certificate. To get the MD5 fingerprint for this certificate on Mac and Linux, you can use the following command:

```
cd ~/.android
keytool -list -keystore ./debug.keystore -storepass android -keypass android
```

On Windows, adjust for the user's home directory slash directions, such as

```
cd c:\Users\Chris\.android
keytool -list -keystore debug.keystore -storepass android -keypass android
```

Getting a key for a production application follows the same process, but you need to use the actual certificate your APK file is signed with instead of the debug.keystore file. The Android documentation provides additional information about obtaining a key at <http://code.google.com/android/add-ons/google-apis/mapkey.html>. For more information about digital signatures, keys, and signing in general, see [appendix B](#).

Caution

Android requires you to declare the map API key in the layout file. With the key in the layout file, you must remember to update the key between debug and production modes. Additionally, if you debug on different development machines, you must switch keys by hand.

Once you write a `MapActivity` with a `MapView` and create your view in the layout file, complete with map API key, you can make full use of the map. Several of the previous listings use the `MapView` from the Wind and Waves application. In the next listing, we repeat a few of the map-related lines of code we've already shown and add related items to consolidate all the map-related concepts in one listing.

Listing 11.7. Portions of code that demonstrate working with maps

```
... from onCreate
mapView = (MapView)
    findViewById(R.id.map_view);
mapView.
    setBuiltInZoomControls(true);
...
from onStart
mapController = mapView.getController();
mapController.setZoom(10);
mapController.
    animateTo(lastKnownPoint);
...
from onMenuItemSelected
case MapViewActivity.MENU_SET_MAP:
    mapView.setSatellite(false);
    break;
case MapViewActivity.MENU_SET_SATELLITE:
    mapView.setSatellite(true);
    break;
case MapViewActivity.MENU_BUOYS_FROM_MAP_CENTER:
    getBuoyData(mapView.getMapCenter());
    break;
```

The diagram consists of three numbered callouts pointing to specific sections of the code:

- 1 Inflate MapView from layout**: Points to the line `mapView = (MapView) findViewById(R.id.map_view);`.
- 2 Animate to given GeoPoint**: Points to the line `mapController.animateTo(lastKnownPoint);`.
- 3 Set map satellite mode**: Points to the line `mapView.setSatellite(true);`.

We declare the `MapView` in XML and inflate it just like other `View` components ①. Because it's a `ViewGroup`, we can also combine and attach other elements to it. We tell the `MapView` to display its built-in zoom controls so the user can zoom in and out.

Next we get a `MapController` from the `MapView`. The controller allows us to programmatically zoom and move the map. When starting, we use the controller to set the initial zoom level and animate to a specified `GeoPoint` ②. When the user selects a view mode from the menu, we set the mode of the map from plain to satellite or back again ③. Along with manipulating the map itself, you can retrieve data from it, such as the coordinates of the map center.

Besides manipulating the map and getting data from it, you can draw items on top of the map using `Overlay` instances.

11.3.3. Placing data on a map with an Overlay

The small buoy icons for the Wind and Waves application that we've used in several figures up to this point draw on the screen at specified coordinates using an `Overlay`.

`Overlay` describes an item to draw on the map. You can define your own `Overlay` by extending this class or `MyLocationOverlay`. The `MyLocationOverlay` class lets you display a user's current location with a compass, and it has other useful features such as a `LocationListener` for convenient access to position updates.

Besides showing the user's location, you'll often place multiple marker items on the map. Users generally expect to see markers as pushpins. You'll create buoy markers for the location of every buoy using data you get back from the NDBC feeds. Android provides built-in support for this with the `ItemizedOverlay` base class and the `OverlayItem`.

`OverlayItem`, a simple bean, includes a title, a text snippet, a drawable marker, coordinates defined in a `GeoPoint`, and a few other properties. The following listing shows the buoy data-related `BuoyOverlayItem` class for Wind and Waves.

Listing 11.8. The `OverlayItem` subclass `BuoyOverlayItem`

```
public class BuoyOverlayItem extends OverlayItem {  
  
    public final GeoPoint point;  
  
    public final BuoyData buoyData;  
  
    public BuoyOverlayItem(GeoPoint point, BuoyData buoyData) {
```

```

        super(point, buoyData.title, buoyData.dateString);

        this.point = point;

        this.buoyData = buoyData;

    }

}

```

We extend `OverlayItem` to include all the necessary properties of an item to draw on the map. In the constructor we call the superclass constructor with the location, the title, and a brief snippet, and we assign additional elements our subclass instance variables. In this case, we add a `BuoyData` member, which is another bean with name, water temperature, wave height, and other properties.

After you prepare the individual item class, you need a class that extends `ItemizedOverlay` and uses a `Collection` of the items to display them on the map one by one. The following listing, the `BuoyItemizedOverlay` class, shows how this works.

Listing 11.9. The `BuoyItemizedOverlay` class

```

public class BuoyItemizedOverlay
    extends ItemizedOverlay<BuoyOverlayItem> {
    private final List<BuoyOverlayItem> items;
    private final Context context;
    public BuoyItemizedOverlay(List<BuoyOverlayItem> items,
        Drawable defaultMarker, Context context) {
        super(defaultMarker);
        this.items = items;
        this.context = context;
        populate();
    }
    @Override
    public BuoyOverlayItem createItem(int i) {
        return items.get(i);
    }
    @Override
    protected boolean onTap(int i) {
        final BuoyData bd = items.get(i).buoyData;
        LayoutInflater inflater = LayoutInflater.from(context);
        View bView = inflater.inflate(R.layout.buoy_selected, null);
        TextView title = (TextView) bView.findViewById(R.id.buoy_title);
        . . . rest of view inflation omitted for brevity
        new AlertDialog.Builder(context)

```

The diagram illustrates the five steps required to implement the `BuoyItemizedOverlay` class:

- 1 Extend `ItemizedOverlay`**: An arrow points from the class definition to the `extends ItemizedOverlay<BuoyOverlayItem>` line.
- 2 Include `Collection of OverlayItem`**: An arrow points from the `items` field declaration to the `List<BuoyOverlayItem> items;` line.
- 3 Provide `drawable marker`**: An arrow points from the `super(defaultMarker);` call to the `defaultMarker` parameter.
- 4 Override `createItem`**: An arrow points from the `@Override` annotation to the `createItem` method.
- 5 Get data to `display`**: An arrow points from the `onTap` method to the code that inflates the view and handles the alert dialog.

```

        .setView(bView)
        .setPositiveButton("More Detail",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface di, int what) {
                    Intent intent =
                        new Intent(context, BuoyDetailActivity.class);
                    BuoyDetailActivity.buoyData = bd;
                    context.startActivity(intent);
                }
            })
        .setNegativeButton("Cancel",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface di, int what) {
                    di.dismiss();
                }
            })
        .show();
    return true;
}
@Override
public int size() {
    return items.size();
}
@Override
public void draw(Canvas canvas, MapView mapView, boolean b) {
    super.draw(canvas, mapView, false);
}
}

```

6 Override
size method

7 Customized
drawing

The `BuoyItemizedOverlay` class extends `ItemizedOverlay` 1 and includes a `Collection` of `BuoyOverlayItem` elements 2. In the constructor, we pass the `Drawable` marker to the parent class 3. This marker draws on the screen in the overlay to represent each point on the map.

`ItemizedOverlay` takes care of many of the details you'd otherwise have to implement yourself if you made your own `Overlay` with multiple points drawn on it. This includes drawing items, handling focus, and processing basic events. An `ItemizedOverlay` will invoke the `onCreate()` method 4 for every element in the `Collection` of items it holds. `ItemizedOverlay` also supports facilities such as `onTap` 5, where you can react when the user selects a particular overlay item. In this code, we inflate some views and display an `AlertDialog` with information about the respective buoy when a `BuoyOverlayItem` is tapped. From the alert, the user can navigate to more detailed information if desired.

The `size()` method tells `ItemizedOverlay` how many elements it needs to process **6**, and even though we aren't doing anything special with it in this case, there are also

7 methods such as `onDraw()` that you can customize to draw something beyond the standard pushpin.

When working with a `MapView`, you create the `Overlay` instances you need and then add them on top of the map. *Wind and Waves* uses a separate `Thread` to retrieve the buoy data in the `MapViewActivity`. You can view the data-retrieval code in the code download for this chapter. After downloading the buoy data, you send a `Message` to a `Handler` that adds the `BuoyItemizedOverlay` to the `MapView`. The following listing shows these details.

Listing 11.10. The Handler Wind and Waves uses to add overlays to the MapView

```
private final Handler handler = new Handler() {  
  
    public void handleMessage(final Message msg) {  
  
        progressDialog.dismiss();  
  
        if (mapView.getOverlays().contains(buoyOverlay)) {  
  
            mapView.getOverlays().remove(buoyOverlay);  
  
        }  
  
        buoyOverlay = new BuoyItemizedOverlay(buoys,  
  
                                              defaultMarker,  
  
                                              MapViewActivity.this);  
  
        mapView.getOverlays().add(buoyOverlay);  
  
    }  
  
};
```

A `MapView` contains a `Collection` of `Overlay` elements. We use the `remove()` method to clean up any existing `BuoyOverlayItem` class before we create and add a new one. This way, we reset the data instead of adding more items on top of each other.

The built-in `Overlay` subclasses perfectly handle our requirements.

The `ItemizedOverlay` and `OverlayItem` classes have allowed us to complete the *Wind and Waves* application without having to make our own `Overlay` subclasses directly. If you need to, Android lets you go to that level and implement your own `draw()`, `tap()`, `touch()`, and other methods within your custom `Overlay`.

With this sample application now complete and providing you with buoy data using a `MapActivity` and `MapView`, we need to address one final maps-related concept that you haven't yet encountered—geocoding.

11.4. CONVERTING PLACES AND ADDRESSES WITH GEOCODER

The Android documentation describes *geocoding* as converting a “street address or other description of a location” into latitude and longitude coordinates. *Reverse geocoding* is the opposite—converting latitude and longitude into an address. To accomplish this, the `Geocoder` class makes a network call to a web service.

You won't use geocoding in Wind and Waves because the ocean doesn't contain cities, addresses, and so on. Nevertheless, geocoding provides invaluable tools when working with coordinates and maps. To demonstrate the concepts surrounding geocoding, this listing includes a new single `Activity` application, `GeocoderExample`.

Listing 11.11. A `Geocoder` example

```
... Class declaration and Instance variables omitted for brevity
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    input = (EditText) findViewById(R.id.input);
    output = (TextView) findViewById(R.id.output);
    button = (Button) findViewById(R.id.geocode_button);
    isAddress = (CheckBox)
        findViewById(R.id.checkbox_address);
    button.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            output.setText(performGeocode(
                input.getText().toString(),
                isAddress.isChecked()));
        }
    });
}
```

```

private String performGeocode(String in, boolean isAddr) {
    String result = "Unable to Geocode - " + in;
    if (input != null) {
        Geocoder geocoder = new Geocoder(this);
        if (isAddr) {
            try {
                List<Address> addresses =
                    geocoder.getFromLocationName(in, 1);
                if (addresses != null) {
                    result = addresses.get(0).toString();
                }
            } catch (IOException e) {
                Log.e("GeocodExample", "Error", e);
            }
        } else {
            try {
                String[] coords = in.split(",");
                if ((coords != null) && (coords.length == 2)) {
                    List<Address> addresses =
                        geocoder.getFromLocation(
                            Double.parseDouble(coords[0]),
                            Double.parseDouble(coords[1]),
                            1);
                    result = addresses.get(0).toString();
                }
            } catch (IOException e) {
                Log.e("GeocodExample", "Error", e);
            }
        }
    }
    return result;
}

```

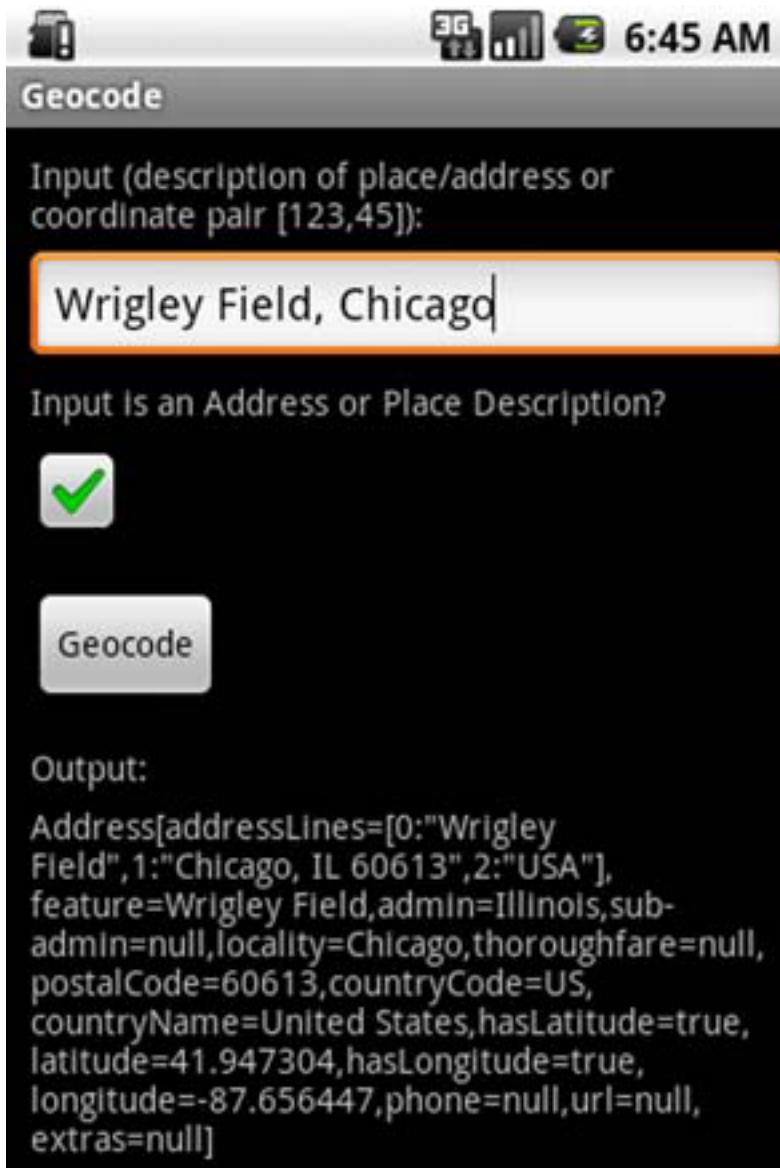
The code is annotated with three numbered callouts:

- 1** Instantiate Geocoder with Context: Points to the line `Geocoder geocoder = new Geocoder(this);`.
- 2** Get Address from location name: Points to the block of code under the condition `if (isAddr)`.
- 3** Get Address from coordinates: Points to the block of code under the condition `else`.

You create a `Geocoder` by constructing it with the `Context` of your application **1**. You then use a `Geocoder` to either convert `String` instances that represent place names into `Address` objects with the `getFromLocationName()` method **2** or convert latitude and longitude coordinates into `Address` objects with the `getFromLocation()` method **3**.

Figure 11.8 shows our `GeocoderExample` in use. In this case, we've converted a `String` describing Wrigley Field in Chicago into an `Address` object containing latitude and longitude coordinates.

Figure 11.8. `Geocoder` example turning a `String` into an `Address` object that provides latitude and longitude coordinates



Geocoder provides many useful features. For instance, if you have data that includes address string portions, or only place descriptions, you can easily convert them into latitude and longitude numbers for use with `GeoPoint` and `Overlay` to place them on the user's map.

Caution

As of this writing, the AVD for API level 8 (the OS 2.2 emulator) doesn't properly support the geocoder. Attempts to look up an address will result in a "Service not Available" exception. Geocoding does work properly on OS 2.2 devices. To work around

this problem during development, you can use API level 7 for building and testing your app on the emulator.



Geocoding concludes our look at the powerful location- and mapping-related components of the Android platform.

11.5. SUMMARY

“Location, location, location,” as they say in real estate, could also be the mantra for the future of mobile computing. Android supports readily available location information and includes smart-mapping APIs and other location-related utilities.

In this chapter, we explored the location and mapping capabilities of the Android platform. You built an application that acquired a `LocationManager` and `LocationProvider`, to which you attached several `LocationListener` instances. You did this so that you could keep your application informed about the current device location by using updates delivered to your listeners. Along with the `LocationListener`, we also briefly discussed several other ways to get location updates from the Android platform.

After we covered location-awareness basics, we showed you how to add information from a unique data source, the National Data Buoy Center, to provide a draggable, zoomable, interactive map. To build the map you used a `MapActivity`, along with `MapView` and `MapController`. These classes make it fairly easy to set up and display maps. Once you had your `MapView` in place, you created an `ItemizedOverlay` to include points of interest, using individual `OverlayItem` elements. From the individual points, in this case buoys, you linked into another `Activity` class to display more detailed information, thereby demonstrating how to go from the map to any other kind of `Activity` and back.

Our water-based sample application didn’t include the important mapping feature of converting from an address into a latitude and longitude and vice versa. To demonstrate this capability, we showed you how to build a separate small sample and discussed usage of the `Geocoder` class.

With our exploration of the mapping capabilities of Android complete, including a fully functional sample application that combines mapping with many other Android tenets we’ve previously explored, we’ll move into a new stage of the book. In the next few chapters, we’ll explore complete nontrivial applications that bring together intents, activities, data storage, networking, and more.

Chapter 12. Putting Android to work in a field service application

This chapter covers

- Designing a real-world Android application
- Mapping out the application flow
- Writing application source code
- Downloading, data parsing, and signature capture
- Uploading data to a server

Now that we've introduced and examined Android and some of its core technologies, it's time to put together a more comprehensive application. Exercising APIs can be informative, educational, and even fun for a while, but at some point a platform must demonstrate its worth via an application that can be used outside of the ivory tower—and that's what this chapter is all about. In this chapter, we systematically design, code, and test an Android application to aid a team of field service technicians in performing their job. The application syncs XML data with an internet-hosted server, presents data to the user via intuitive user interfaces, links to Google Maps, and concludes by collecting customer signatures via Android's touch screen. Many of the APIs introduced earlier are exercised here, demonstrating the power and versatility of the Android platform.

In addition to an in-depth Android application, this chapter's sample application works with a custom website application that manages data for use by a mobile worker. This server-side code is presented briefly toward the end of the chapter. All of the source code for the server-side application is available for download from the book's companion website.

If this example is going to represent a useful real-world application, we need to put some flesh on it. Beyond helping you understand the application, this definition process will get you thinking about the kinds of impact a mobile application can have on our economy. This chapter's sample application is called a *field service application*. A pretty generic name perhaps, but it'll prove to be an ample vehicle for demonstrating key elements required in mobile applications, as well as demonstrating the power of the Android platform for building useful applications quickly.

Our application's target user is a fleet technician who works for a national firm that makes its services available to a number of contracted customers. One day our

technician, who we'll call a *mobile worker*, is replacing a hard drive in the computer at the local fast-food restaurant, and the next day he may be installing a memory upgrade in a piece of pick-and-place machinery at a telephone system manufacturer. If you've ever had a piece of equipment serviced at your home or office and thought the technician's uniform didn't really match the job he was doing, you've experienced this kind of service arrangement. This kind of technician is often referred to as *hands and feet*. He has basic mechanical or computer skills and is able to follow directions reliably, often guided by the manufacturer of the equipment being serviced at the time. Thanks to workers like these, companies can extend their reach to a much broader geography than internal staffing levels would ever allow. For example, a small manufacturer of retail music-sampling equipment might contract with such a firm to provide tech support to retail locations across the country.

Because of our hypothetical technician's varied schedule and lack of experience on a particular piece of equipment, it's important to equip him with as much relevant and timely information as possible. But he can't be burdened with thick reference manuals or specialized tools. So, with a toolbox containing a few hand tools and of course an Android-equipped device, our fearless hero is counting on us to provide an application that enables him to do his job. And remember, this is the person who restores the ice cream machine to operation at the local Dairy Barn, or perhaps fixes the farm equipment's computer controller so the cows get milked on time. You never know where a computer will be found in today's world!

If built well, this application can enable the efficient delivery of service to customers in many industries, where we live, work, and play. Let's get started and see what this application must be able to accomplish and how Android steps up to the task.

12.1. DESIGNING A REAL-WORLD ANDROID APPLICATION

We've established that our mobile worker will be carrying two things: a set of hand tools and an Android device. Fortunately, in this book we're not concerned with the applicability of the hand tools in his toolbox, leaving us free to focus on the capabilities and features of a field service application running on the Android platform. In this section, we define the basic and high-level application requirements.

12.1.1. Core requirements of the application

Before diving into the bits and bytes of data requirements and application features, it's helpful to enumerate some basic requirements and assumptions about our field service application. Here are a few items that come to mind for such an application:

- The mobile worker is dispatched by a home office/dispatching authority, which takes care of prioritizing and distributing job orders to the appropriate technician.
- The mobile worker is carrying an Android device, which has full data service—a device capable of browsing rich web content. The application needs to access the internet for data transfer as well.
- The home office dispatch system and the mobile worker share data via a wireless internet connection on an Android device; a laptop computer isn't necessary or even desired.
- A business requirement is the proof of completion of work, most readily accomplished with the capture of a customer's signature. Of course, an electronic signature is preferred.
- The home office wants to receive job completion information as soon as possible, as this accelerates the invoicing process, which improves cash flow.
- The mobile worker is also eager to perform as many jobs as possible, because he's paid by the job, not by the hour, so getting access to new job information as quickly as possible is a benefit to him.
- The mobile worker needs information resources in the field and can use as much information as possible about the problem he's being asked to resolve. The mobile worker may have to place orders for replacement parts while in the field.
- The mobile worker will require navigation assistance, as he's likely covering a rather large geographic area.
- The mobile worker needs an intuitive application—one that's simple to use with a minimum number of requirements.

There are likely additional requirements for such an application, but this list is adequate for our purposes. One of the most glaring omissions from our list is security.

Security in this kind of an application comes down to two fundamental aspects. The first is physical security of the Android device. Our assumption is that the device itself is locked and only the authorized worker is using it. A bit naïve perhaps, but there are more important topics we need to cover in this chapter. If this bothers you, just assume there's a sign-in screen with a password field that pops up at the most inconvenient times, forcing you to tap in your password on a small keypad. Feel better now? The second security topic is the secure transmission of data between the Android device and the dispatcher. This is most readily accomplished through the use of a *Secure Sockets Layer (SSL)* connection whenever required.

The next step in defining this application is to examine the data flows and discuss the kind of information that must be captured to satisfy the functional requirements.

12.1.2. Managing the data

Throughout this chapter, the term *job* refers to a specific task or event that our mobile worker engages in. For example, a request to replace a hard drive in a computer at the

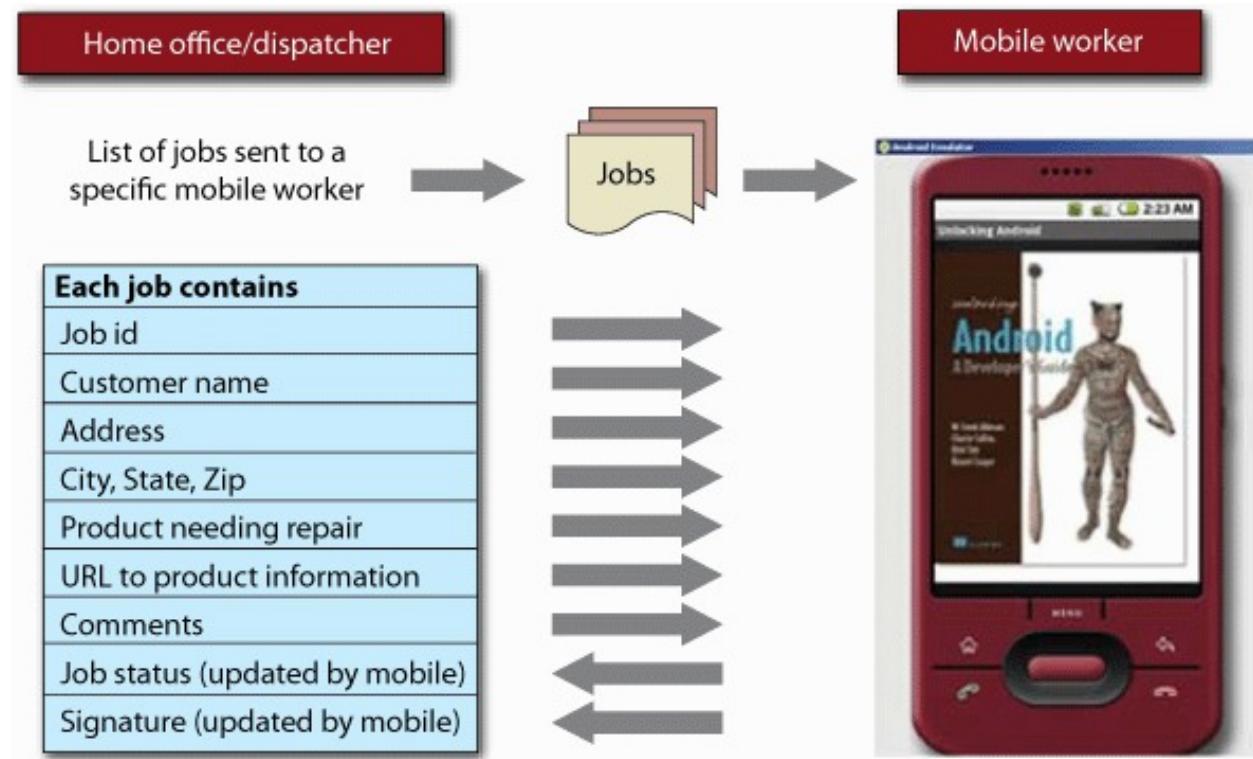
bookstore is a job. A request to upgrade the firmware in the fuel-injection system at the refinery is likewise a job. The home office dispatches one or more jobs to the mobile worker on a regular basis. Certain data elements in the job are helpful to the mobile worker to accomplish his goal of completing the job. This information comes from the home office. Where the home office gets this information isn't our concern for this application.

In this chapter's sample application, there are only two pieces of information the mobile worker is responsible for submitting to the dispatcher:

- The mobile worker communicates to the home office that a job has been *closed*, or completed.
- The mobile worker collects an electronic signature from the customer, acknowledging that the job has, in fact, been completed.

Figure 12.1 depicts these data flows.

Figure 12.1. Data flows between the home office and a mobile worker



Of course, additional pieces of information exist that may be helpful here, such as the customer's phone number, the anticipated duration of the job, replacement parts required in the repair (including tracking numbers), any observations about the condition of related equipment, and much more. Although important to a real-world

application, these pieces of information are extraneous to the goals of this chapter and are left as an exercise for you to extend the application for your own learning and use.

The next objective is to determine how data is stored and transported.

12.1.3. Application architecture and integration

Now that you know which entities are responsible for the relevant data elements, and in which direction they flow, let's look at how the data is stored and exchanged. You'll be deep into code before too long, but for now we'll focus on the available options and continue to examine things from a requirements perspective, building to a proposed architecture.

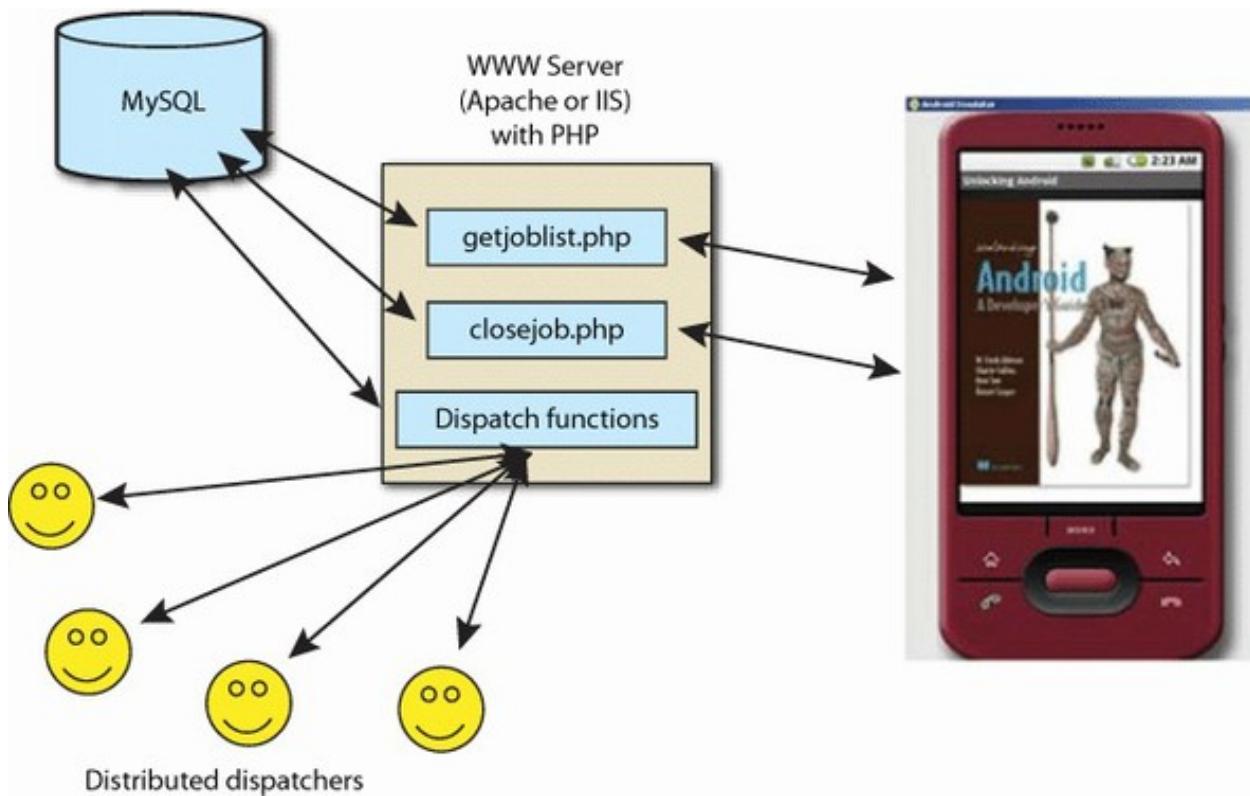
At the home office, the dispatcher must manage data for multiple mobile workers. The best tool for this purpose is a relational database. The options here are numerous, but we'll make the simple decision to use MySQL, a popular open source database. Not only are there multiple mobile workers, but the organization we're building this application for is quite spread out, with employees in multiple markets and time zones. Because of the nature of the dispatching team, it's been decided to host the MySQL database in a data center, where it's accessed via a browser-based application. For this sample application, the dispatcher system is supersimple and written in PHP.¹

¹ See Manning's *PHP in Action* for details on PHP development: www.manning.com/reiersol/.

Data storage requirements on the mobile device are modest. At any point, a given mobile worker may have only a half-dozen or so assigned jobs. Jobs may be assigned at any time, so the mobile worker is encouraged to refresh the list of jobs periodically. Although you learned about how to use SQLite in [chapter 5](#), we have little need for sharing data between multiple applications and don't need to build a `ContentProvider`, so we've decided to use an XML file stored on the filesystem to serve as a persistent store of our assigned job list.

The field service application uses HTTP to exchange data with the home office. Again, we use PHP to build the transactions for exchanging data. Though more complex and sophisticated protocols can be employed, such as SOAP, this application simply requests an XML file of assigned jobs and submits an image file representing the captured signature. In fact, SOAP is simple in name only and should be avoided. A better solution that's coming on strong in the mobile and web space is the JSON format. This architecture is depicted in [figure 12.2](#).

Figure 12.2. The field service application and dispatchers both leverage server-side transactions.



The last item to discuss before diving into the code is configuration. Every mobile worker needs to be identified uniquely. This way, the field service application can retrieve the correct job list, and the dispatchers can assign jobs to workers in the field. Similarly, the mobile application may need to communicate with different servers, depending on locale. A mobile worker in the United States might use a server located in Chicago, but a worker in the United Kingdom may need to use a server in Cambridge. Because of these requirements, we've decided that both the mobile worker's identifier and the server address need to be readily accessed within the application. Remember, these fields would likely be secured in a deployed application, but for our purposes they're easy to access and not secured.

We've identified the functional requirements, defined the data elements necessary to satisfy those objectives, and selected the preferred deployment platform. The next section examines the high-level solution to the application's requirements.

12.2. MAPPING OUT THE APPLICATION FLOW

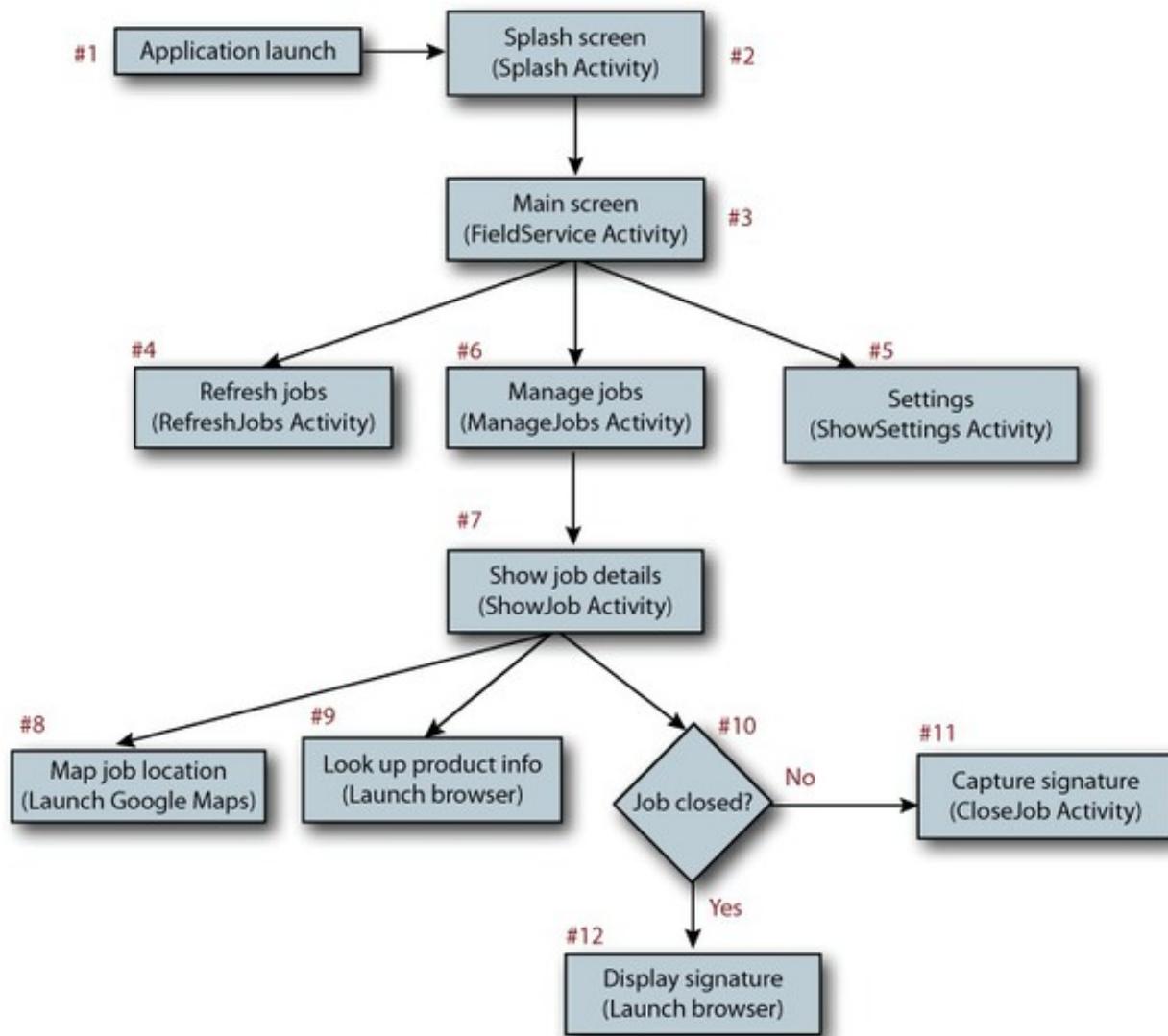
Have you ever downloaded an application's source code, excited to get access to all that code, but found it overwhelming to navigate? You want to make your own changes, to put your own spin on the code, but you unzip the file into all the various subdirectories, and you don't know where to start. Before we jump directly into examining the source

code, we need to pay attention to the architecture, in particular the flow from one screen to the next.

12.2.1. Mapping out the field service application

In this section, we'll examine the application flow to better understand the relation among the application's functionality, the UI, and the classes used to deliver this functionality. Doing this process up-front helps ensure that the application delivers the needed functionality and assists in defining which classes we require when it comes time to start coding (which is soon)! [Figure 12.3](#) shows the relation between the high-level classes in the application, which are implemented as an Android `Activity`, as well as interaction with other services available in Android.

[Figure 12.3](#). This figure depicts the basic flow of the field service application.



Here's the procession of steps in the application:

1. The application is selected from the application launch screen on the Android device.
2. The application splash screen displays. Why? Some applications require setup time to get data structures initialized. As a practical matter, such time-consuming behavior is discouraged on a mobile device, but it's an important aspect to application design, so we include it in this sample application.
3. The main screen displays the currently configured user and server settings, along with three easy-to-hit-with-your-finger buttons.
4. The Refresh Jobs button initiates a download procedure to fetch the currently available jobs for this mobile worker from the configured server. The download includes a `ProgressDialog`, which we discuss in [section 12.3.5](#).
5. The Settings button brings up a screen that allows you to configure the user and server settings.
6. Selecting Manage Jobs lets our mobile worker review the available jobs assigned to him and proceed with further steps specific to a chosen job.
7. Selecting a job from the list of jobs on the Manage Jobs screen brings up the Show Job Details screen with the specific job information listed. This screen lists the available information about the job and presents three additional buttons.
8. The Map Job Location button initiates a geo query on the device using an `Intent`. The default handler for this `Intent` is the Maps application.
9. Because our mobile worker may not know much about the product he's being asked to service, each job includes a product information URL. Clicking this button brings up the built-in browser and takes the mobile worker to a (hopefully) helpful internet resource. This resource may be an online manual or an instructional video.
10. The behavior of the third button depends on the current status of the job. If the job is still marked OPEN, this button is used to initiate the closeout or completion of this job.
11. When the close procedure is selected, the application presents an empty canvas upon which the customer can take the stylus (assuming a touch screen-capable Android device) and sign that the work is complete. A menu on that screen presents two options: Sign & Close and Cancel. If the user selects Sign & Close option, the application submits the signature as a JPEG image to the server, and the server marks the job as CLOSED. In addition, the local copy of the job is marked as CLOSED. The Cancel button causes the Show Job Details screen to be restored.
12. If the job being viewed has already been closed, the browser window is opened to a page displaying the previously captured signature.

Now that you have a feel for what the requirements are and how you're going to tackle the problem from a functionality and application-flow perspective, let's examine the code that delivers this functionality.

12.2.2. List of source files

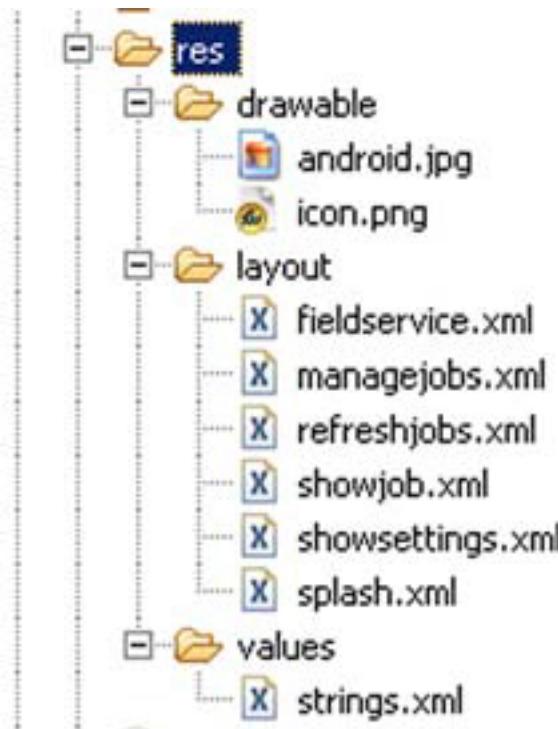
The source code for this application consists of 12 Java source files, one of which is the R.java file, which you'll recall is automatically generated based on the resources in the application. This section presents a quick introduction to each of these files. We won't explain any code yet; we want you to know a bit about each file, and then it'll be time to jump into the application, step by step. [Table 12.1](#) lists the source files in the Android field service application.

Table 12.1. Source files used to implement the field service application

Source filename	Description
Splash.java	Activity provides splash screen functionality.
ShowSettings.java	Activity provides management of the username and server URL address.
FieldService.java	Activity provides the main screen of the application.
RefreshJobs.java	Activity interacts with the server to obtain an updated list of jobs.
ManageJobs.java	Activity provides access to the list of jobs.
ShowJob.java	Activity provides detailed information on a specific job, such as an address lookup, or initiates the signature-capture process.
CloseJob.java	Activity collects an electronic signature and interacts with the server to upload images and mark jobs as CLOSED.
R.java	Automatically generated source file representing identifiers in the resources.
Prefs.java	Helper class encapsulating SharedPreferences.
JobEntry.java	Class that represents a job. Includes helpful methods used when passing JobEntry objects from one Activity to another.
JobList.java	Class representing the complete list of JobEntry objects. Includes methods for marshaling and unmarshaling to nonvolatile storage.
JobListHandler.java	Class used for parsing the XML document containing job data.

The application also relies on `layout` resources to define the visual aspect of the UI. In addition to the `layout` XML files, an image used by the `Splash Activity` is placed in the drawable subfolder of the `res` folder along with the stock Android icon image. This icon is used for the home application launch screen. [Figure 12.4](#) depicts the resources used in the application.

Figure 12.4. Resource files used in the sample application



In an effort to make navigating the code as easy as possible, table 2.2 shows the field service application resource files. Note that each is clearly seen in [figure 12.4](#), which is a screenshot from our project open in Eclipse.

Examining the source files in this application tells us that we have more than one `Activity` in use. To enable navigation between one `Activity` and the next, our application must inform Android of the existence of these `Activity` classes. Recall from [chapter 1](#) that this registration step is accomplished with the `AndroidManifest.xml` file.

Table 12.2. Resource files used in the sample application

Filename	Description
android.jpg	Image used in the Splash Activity.
icon.jpg	Image used in the application launcher.
fieldservice.xml	Layout for the main application screen, FieldService Activity.
managejobs.xml	Layout for the list of jobs, ManageJobs Activity.
refreshjobs.xml	Layout for the screen shown when refreshing the job list, RefreshJobs Activity.
showjob.xml	Layout for the job detail screen, ShowJob Activity.
showsettings.xml	Layout for the configuration/settings screen, ShowSettings Activity.
splash.xml	Layout for the splash screen, Splash Activity.

Filename	Description
strings.xml	Strings file containing extracted strings. Ideally, all text is contained in a strings file for ease of localization. This application's file contains only the application title.

12.2.3. Field service application's AndroidManifest.xml

Every Android application requires a manifest file to let Android properly “wire things up” when an `Intent` is handled and needs to be dispatched. Take a look at the `AndroidManifest.xml` file used by our application, shown in the following listing.

Listing 12.1. The field service application's `AndroidManifest.xml` file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.UnlockingAndroid">
    <application android:icon="@drawable/icon">
        <activity android:name=".Splash"
            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=".FieldService" >
        </activity>
        <activity android:name=".RefreshJobs" >
        </activity>
        <activity android:name=".ManageJobs" >
        </activity>
        <activity android:name=".ShowJob" >
        </activity>
        <activity android:name=".CloseJob" >
        </activity>
        <activity android:name=".ShowSettings" >
        </activity>
    </application>
    <uses-sdk android:minSdkVersion="6"/>
    <uses-permission android:name="android.permission.INTERNET">
        </uses-permission>
</manifest>
```

12.3. APPLICATION SOURCE CODE

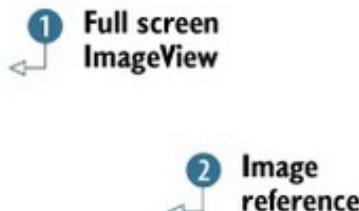
After a rather long introduction, it's time to look at the source code for the field service application. The approach is to follow the application flow, step by step. Let's start with the splash screen. In this portion of the chapter, we work through each of the application's source files, starting with the splash screen and moving on to each subsequent `Activity` of the application.

12.3.1. Splash Activity

We're all familiar with a splash screen for a software application. It acts like a curtain while important things are taking place behind the scenes. Ordinarily, splash screens are visible until the application is ready—this could be a brief amount of time or much longer when a bit of housekeeping is necessary. As a rule, a mobile application should focus on economy and strive to consume as few resources as possible. The splash screen in this sample application is meant to demonstrate how such a feature may be constructed—we don't need one for housekeeping purposes. But that's okay; you can learn in the process. Two code snippets are of interest to us: the implementation of the `Activity` as well as the layout file that defines what the UI looks like. First, examine the layout file in the following listing.

Listing 12.2. `splash.xml`, defining the layout of the application's splash screen

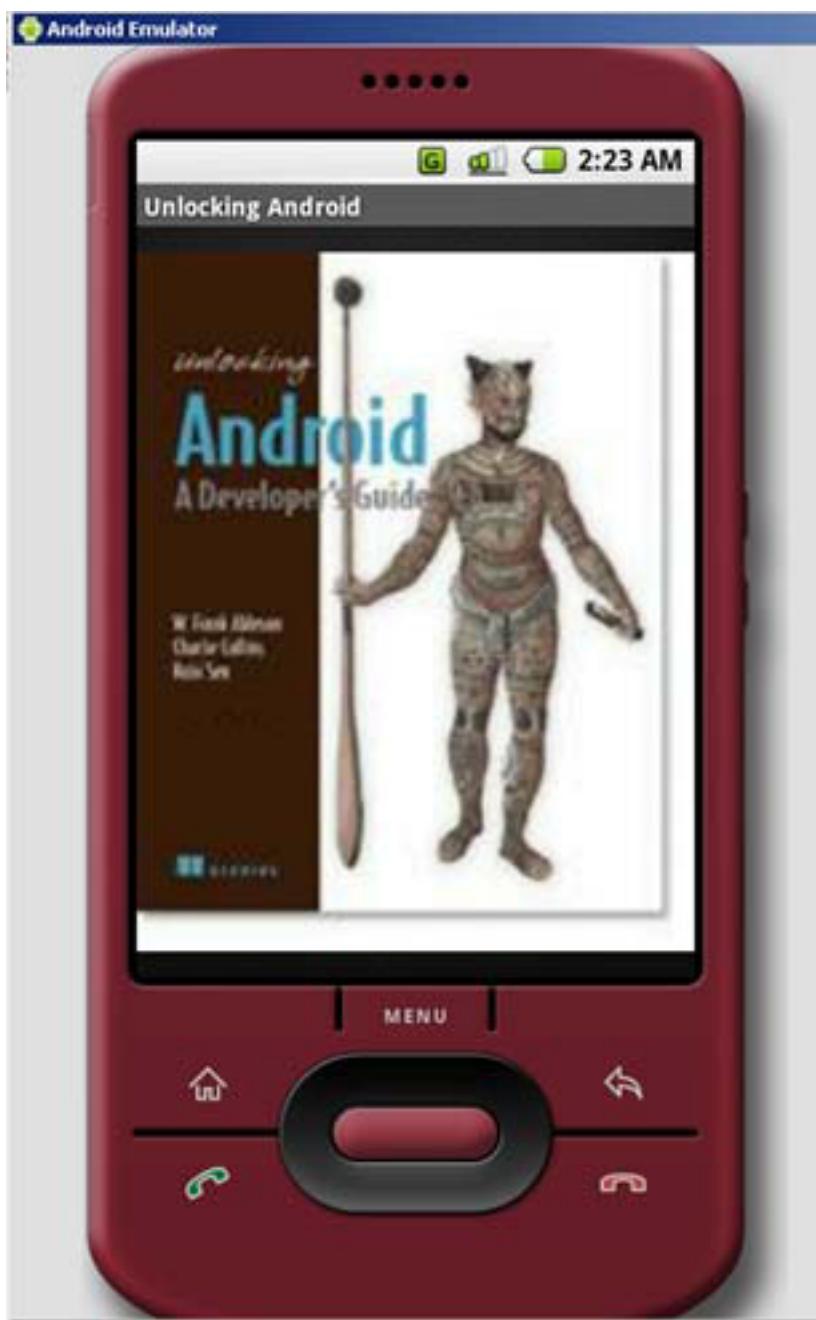
```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <ImageView
        android:layout_width="fill_parent"
      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



The `splash.xml` layout contains a single `ImageView` ①, set to fill the entire screen. The source image for this view is defined as the `drawable` resource ②, named `android`. Note that this is simply the name of the file (minus the file extension) in the `drawable` folder, as shown earlier.

Now you must use this layout in an `Activity`. Aside from the referencing of an image resource from the layout, this part is not that interesting. [Figure 12.5](#) shows the splash screen running on the Android Emulator.

Figure 12.5. The splash screen



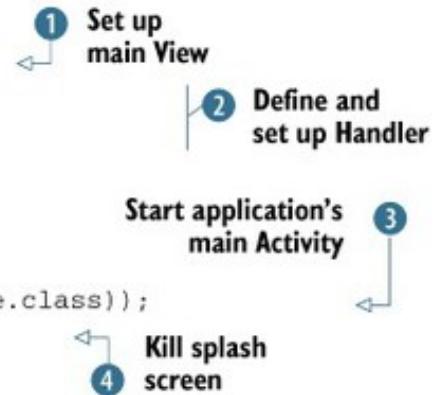
Of interest to us is the code that creates the splash page functionality, shown in the following listing.

Listing 12.3. `Splash.java`, which implements the splash screen functionality

```

package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source code
public class Splash extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.splash);
        Handler x = new Handler();
        x.postDelayed(new SplashHandler(), 2000);
    }
    class SplashHandler implements Runnable {
        public void run() {
            startActivity(
                new Intent(getApplicationContext(), FieldService.class));
            Splash.this.finish();
        }
    }
}

```



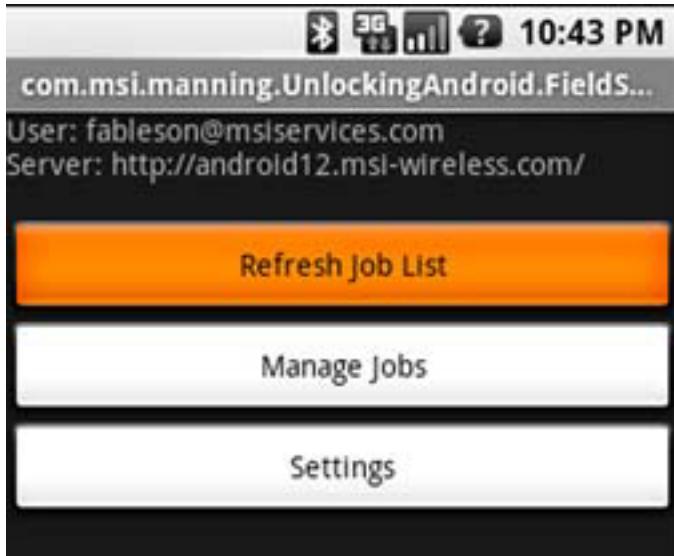
As with most `Activity` classes in Android, we want to associate the splash layout with this `Activity`'s View ①. A `Handler` is set up ②, which is used to close down the splash screen after an elapsed period of time. Note that the arguments to the `postDelayed()` method are an instance of a class that implements the `Runnable` interface and the desired elapsed time in milliseconds. In this snippet of code, the screen will be shown for 2,000 milliseconds, or 2 seconds. After the indicated amount of time has elapsed, the class `splashhandler` is invoked. The `FieldService` `Activity` is instantiated with a call to `startActivity()` ③. Note that an `Intent` isn't used here—we explicitly specify which class is going to satisfy our request. Once we've started the next `Activity`, it's time to get rid of the splash screen `Activity` ④.

The splash screen is happily entertaining our mobile worker each time he starts the application. Let's move on to the main screen of the application.

12.3.2. Preferences used by the `FieldService` Activity

The goal of the `FieldService` `Activity` is to put the functions the mobile worker requires directly in front of him and make sure they're easy to access. A good mobile application is often one that can be used with one hand, such as using the five-way navigation buttons, or in some cases a thumb tapping on a button. In addition, if there's helpful information to display, you shouldn't hide it. It's helpful for our mobile worker to know that he's configured to obtain jobs from a particular server. [Figure 12.6](#) demonstrates the field service application conveying an easy-to-use home screen.

Figure 12.6. The home screen. Less is more.



Before reviewing the code in `FieldService.java`, let's take a break to discuss how the user and server settings are managed. This is important because these settings are used throughout the application, and as shown in the `fieldservice.xml` layout file, we need to access those values to display to our mobile worker on the home screen.

As you learned in [chapter 5](#), there are a number of means for managing data. Because we need to persist this data across multiple invocations of our application, the data must be stored in a nonvolatile fashion. This application employs private `SharedPreferences` to accomplish this. Why? Despite the fact that we're largely ignoring security for this sample application, using private `SharedPreferences` means that other applications can't casually access this potentially important data. For example, we presently use only an identifier (let's call it an email address for simplicity) and a server URL in this application. But we might also include a password or a PIN in a production-ready application, so keeping this data private is a good practice.

The `Prefs` class can be described as a helper or wrapper class. This class wraps the `SharedPreferences` code and exposes simple getter and setter methods, specific to this application. This implementation knows something about what we're trying to accomplish, so it adds value with some default values as well. Let's look at the following listing to see how our `Prefs` class is implemented.

Listing 12.4. Prefs class

```

package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source code
public class Prefs {
    private SharedPreferences _prefs = null;      ← ① SharedPreferences object
    private Editor _editor = null;
    private String _useremailaddress = "Unknown";
    private String _serverurl =
        "http://android12.msi-wireless.com/getjoblist.php";
    public Prefs(Context context) {
        _prefs = context.getSharedPreferences(
    "PREFS_PRIVATE",
    Context.MODE_PRIVATE);
        _editor = _prefs.edit();
    }

    public String getValue(String key, String defaultvalue) {
        if (_prefs == null) return "Unknown";
        return _prefs.getString(key, defaultvalue);
    }
    public void setValue(String key, String value) {
        if (_editor == null) return;
        _editor.putString(key, value);
    }
    public String getEmail() {
        if (_prefs == null) return "Unknown";
        _useremailaddress = _prefs.getString("emailaddress", "Unknown");
        return _useremailaddress;
    }
    public void setEmail(String newemail) {
        if (_editor == null) return;
        _editor.putString("emailaddress", newemail);
    }
    ... (abbreviated for brevity)
    public void save() {
        if (_editor == null) return;
        _editor.commit();
    }
}

```

The diagram illustrates the flow of logic in the Prefs class. It starts with the declaration of the SharedPreferences object at step 1. Step 2 shows the implementation of the Editor object. Step 3 indicates the use of default values for the shared preferences. Step 4 involves initializing the SharedPreferences object. Step 5 provides generic set and get methods. Step 6 extracts the email value. Step 7 sets the email value. Finally, step 8 saves the preferences.

To persist the application's settings data, we employ a `SharedPreferences` object 1. To manipulate data within the `SharedPreferences` object, here named `_prefs`, you use an instance of the `Editor` class 2. This snippet employs some default settings 3, which are appropriate for this application. The `Prefs()` constructor 4 does the necessary housekeeping so we can establish our private `SharedPreferences` object, including using a passed-in `Context` instance. The `Context` class is necessary because the `SharedPreferences` mechanism relies on a `Context` for segregating data. This snippet

shows a pair of `set` and `get` methods that are generic in nature **5**.

The `getEmail()` **6** and `setEmail()` methods **7** are responsible for manipulating the email setting value. The `save()` method **8** invokes a `commit()` on the `Editor`, which persists the data to the `SharedPreferences` store.

Now that you have a feel for how this important preference data is stored, let's return to examine the code of `FieldService.java`.

12.3.3. Implementing the FieldService Activity

Recall that the `FieldService.java` file implements the `FieldService` class, which is essentially the home screen of our application. This code does the primary dispatching for the application. Many of the programming techniques in this file have been shown earlier in the book, but please note the use of the `startActivityForResult()` and `onActivityResult()` methods as you read through the code shown in the following listing.

Listing 12.5. FieldService.java, which implements the FieldService Activity

```
package com.msi.manning.UnlockingAndroid;
// multiple imports trimmed for brevity, see full source code
public class FieldService extends Activity {
    final int ACTIVITY_REFRESHJOBS = 1;
    final int ACTIVITY_LISTJOBS = 2;
    final int ACTIVITY_SETTINGS = 3;
    Prefs myprefs = null;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.fieldservice);
        myprefs = new Prefs(this.getApplicationContext());
        RefreshUserInfo();
        final Button refreshjobsbutton =
            (Button) findViewById(R.id.getjobs);
        refreshjobsbutton.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                try {
                    startActivityForResult(new
Intent(v.getContext(), RefreshJobs.class), ACTIVITY_REFRESHJOBS);
                } catch (Exception e) {
                }
            }
        });
        // see full source comments
    }
```

The diagram illustrates the flow of code execution. It starts with the `onCreate` method. Inside, it initializes a `Prefs` instance (**1 Prefs instance**). Then, it sets up the UI by calling `setContentView` and `RefreshUserInfo` (**Set up UI**). Next, it creates a `refreshjobsbutton` and sets its `OnClickListener` (**Instantiate Prefs instance**). Finally, when the button is clicked, it calls `startActivityForResult` with the specified intent and activity code (**Connect button to UI**).

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) {
    switch (requestCode) {
        case ACTIVITY_REFRESHJOBS:
            break;
        case ACTIVITY_LISTJOBS:
            break;
        case ACTIVITY_SETTINGS:
            RefreshUserInfo();
            break;
    }
}
private void RefreshUserInfo() {
    try {
        final TextView emaillabel = (TextView)
findViewById(R.id.emailaddresslabel);
        emaillabel.setText("User: " + myprefs.getEmail() + "\nServer: " +
myprefs.getServer() + "\n");
    } catch (Exception e) {
    }
}
}

```

The diagram illustrates the flow of control in the code. A call to `startActivityForResult()` is labeled **3**. This leads to the `onActivityResult()` method, which is labeled **4** and enclosed in a brace. Inside this method, a call to `RefreshUserInfo()` is labeled **5**.

This code implements a simple UI that displays three distinct buttons. As each is selected, a particular `Activity` is started in a synchronous, call/return fashion.

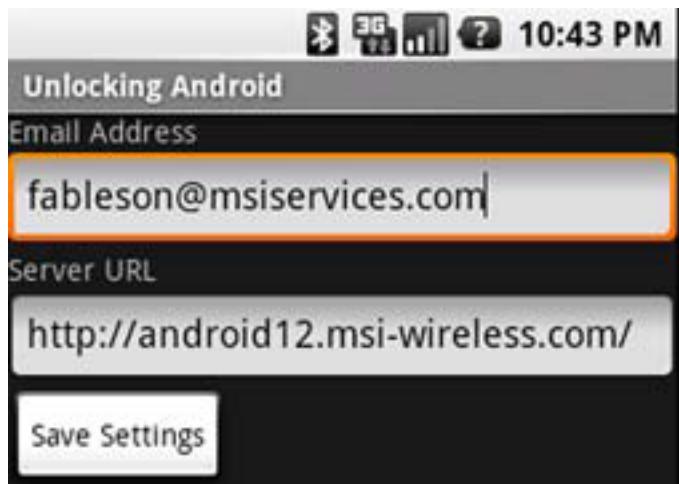
The `Activity` is started with a call to `startActivityForResult()` **3**. When the called `Activity` is complete, the results are returned to the `FieldService Activity` via the `onActivityResult()` method **4**. An instance of the `Prefs` class **1**, **2** is used to obtain values for displaying in the UI. Updating the UI is accomplished in the `RefreshUserInfo()` method **5**.

Because the settings are so important to this application, the next section covers the management of the user and server values.

12.3.4. Settings

When the user clicks the Settings button on the main application screen, an `Activity` is started that allows the user to configure her user ID (email address) and the server URL. The screen layout is basic (see [listing 12.6](#)). It's shown graphically in [figure 12.7](#).

Figure 12.7. Settings screen in use



Listing 12.6. showsettings.xml, which contains UI elements for the settings screen

```
<?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:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Email Address"
        />
    <EditText
        android:id="@+id/emailaddress"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        >
```

Annotations:

- A callout arrow points from the text "Email Address" to the corresponding `<TextView>` element in the XML, with the label "TextView for display of labels".
- A callout arrow points from the text "fableson@msiservices.com" to the corresponding `<EditText>` element in the XML, with the label "EditText for entry of data".

```
        android:autoText="true"
    />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Server URL"
/>
<EditText
    android:id="@+id/serverurl"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoText="true"
/>
<Button android:id="@+id/settingssave"
    android:text="Save Settings"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:enabled="true"
/>
</LinearLayout>
```

The code is annotated with three callout boxes pointing to specific UI elements:

- A callout box points to the first `<TextView>` element with the text "TextView for display of labels".
- A callout box points to the `<EditText>` element with the text "EditText for entry of data".
- A callout box points to the `<Button>` element with the text "Button to initiate saving data".

The source code behind the settings screen is also basic. Note the use of the `PopulateScreen()` method, which makes sure the `EditView` controls are populated with the current values stored in the `SharedPreferences`. Note also the use of the `Prefs` helper class to retrieve and save the values, as shown in the following listing.

Listing 12.7. ShowSettings.java, which implements code behind the settings screen

```
package com.msi.manning.UnlockingAndroid;
// multiple imports trimmed for brevity, see full source code
public class ShowSettings extends Activity {
    Prefs myprefs = null;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.showsettings);

        myprefs = new Prefs(this.getApplicationContext());
        PopulateScreen();
        final Button savebutton = (Button) findViewById(R.id.settingssave);
        savebutton.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                try {
                    final EditText email=
                        (EditText)findViewById(R.id.emailaddress);
                    if (email.getText().length() == 0) {
                        // display dialog, see full source code
                        return;
                    }
                    final EditText serverurl =
                        (EditText)findViewById(R.id.serverurl);
                    if (serverurl.getText().length() == 0) {
                        // display dialog, see full source code
                        return;
                    }

                    myprefs.setEmail(email.getText().toString());
                    myprefs.setServer(serverurl.getText().toString());
                    myprefs.save();
                    finish();
                } catch (Exception e) {
                }
            }
        });
    }
    private void PopulateScreen() {
        try {
            final EditText emailfield = (EditText) findViewById(R.id.emailaddress);
            final EditText serverurlfield = (EditText) findViewById(R.id.serverurl);
            emailfield.setText(myprefs.getEmail());
            serverurlfield.setText(myprefs.getServer());
        } catch Exception e) {
        }
    }
}
```

The diagram illustrates the sequence of steps in the Java code:

- Initialize Prefs instance**: Step 1, indicated by a callout pointing to the line where `myprefs = new Prefs(this.getApplicationContext());` is called.
- Populate UI elements**: Step 2, indicated by a callout pointing to the line where `PopulateScreen();` is called.
- Connect EditText to UI**: Step 3, indicated by a callout pointing to the two `final EditText` declarations.
- Store and save settings**: Step 4, indicated by a callout pointing to the line where `myprefs.save();` is called.
- Finish this Activity**: Step 5, indicated by a callout pointing to the line where `finish();` is called.
- PopulateScreen method sets up UI**: Step 6, indicated by a callout pointing to the line where the `PopulateScreen()` method is defined.

This Activity commences by initializing the `SharedPreferences` instance ①, which retrieves the setting's values and subsequently populates the UI elements ② by calling the application-defined `PopulateScreen()` method ③. When the user clicks the Save Settings button, the `onClick()` method is invoked, where the data is extracted from the UI elements ④ and put back into the `Prefs` instance ⑤. A call to the `finish()` method ⑥ ends this Activity.

Once the settings are in order, it's time to focus on the core of the application: managing jobs for our mobile worker. To get the most out the higher-level functionality of downloading (refreshing) and managing jobs, let's examine the core data structures in use in this application.

12.3.5. Managing job data

Data structures represent a key element of any software project and, in particular, projects consisting of multiple tiers, such as our field service application. Job data is exchanged between an Android application and the server, so the elements of the job are central to our application. In Java, you implement these data structures as classes, which include helpful methods in addition to the data elements. XML data shows up in many locations in this application, so let's start there.

The following listing shows a sample XML document containing a `joblist` with a single job entry.

Listing 12.8. XML document containing data for the field service application

```
<?xml version="1.0" encoding="UTF-8" ?>

<joblist>
  <job>
    <id>22</id>
    <status>OPEN</status>
    <customer>Big Tristan's Imports</customer>
    <address>2200 East Cedar Ave</address>
    <city>Flagstaff</city>
    <state>AZ</state>
    <zip>86004</zip>
```

```

<product>UnwiredTools UTCIS-PT</product>
<producturl>http://unwiredtools.com</producturl>
<comments>Requires tuning - too rich in the mid range RPM.
Download software from website before visiting.</comments>
</job>
</joblist>

```

Now that you have a feel for what the job data looks like, we'll show you how the data is handled in our Java classes.

JobEntry

The individual job is used throughout the application, and therefore it's essential that you understand it. In our application, we define the `JobEntry` class to manage the individual job, as shown in [listing 12.9](#). Note that many of the lines are omitted from this listing for brevity; please see the available source code for the complete code listing.

Listing 12.9. JobEntry.java

```

package com.msi.manning.UnlockingAndroid;
import android.os.Bundle;
public class JobEntry {
    private String _jobid="";
    private String _status = "";
    // members omitted for brevity
    private String _producturl = "";
    private String _comments = "";
    JobEntry() {
    }
    // get/set methods omitted for brevity
    public String toString() {
        return this._jobid + ":" + this._customer + ":" + this._product;
    }
}

```

Annotations for Listing 12.9:

- 1 Bundle class import**: An annotation pointing to the `import android.os.Bundle;` statement. It contains a blue circle with the number 1 and the text "Bundle class import".
- 2 Each member is a String**: An annotation pointing to the declarations of `_jobid`, `_status`, `_producturl`, and `_comments`. It contains a blue circle with the number 2 and the text "Each member is a String".
- 3 toString method**: An annotation pointing to the `toString()` method definition. It contains a blue circle with the number 3 and the text "toString method".

```

public String toXMLString() {
    StringBuilder sb = new StringBuilder("");
    sb.append("<job>");
    sb.append("<id>" + this._jobid + "</id>");
    sb.append("<status>" + this._status + "</status>");
    sb.append("<customer>" + this._customer + "</customer>");
    sb.append("<address>" + this._address + "</address>");
    sb.append("<city>" + this._city + "</city>");
    sb.append("<state>" + this._state + "</state>");
    sb.append("<zip>" + this._zip + "</zip>");
    sb.append("<product>" + this._product + "</product>");
    sb.append("<producturl>" + this._producturl + "</producturl>");
    sb.append("<comments>" + this._comments + "</comments>");
    sb.append("</job>");
    return sb.toString() + "\n";
}

public Bundle toBundle() {
    Bundle b = new Bundle();
    b.putString("jobid", this._jobid);
    b.putString("status", this._status);
    // assignments omitted for brevity
    b.putString("producturl", this._producturl);
    b.putString("comments", this._comments);
    return b;
}

public static JobEntry fromBundle(Bundle b) {
    JobEntry je = new JobEntry();
    je.set_jobid(b.getString("jobid"));
    je.set_status(b.getString("status"));
    // assignments omitted for brevity
    je.set_producturl(b.getString("producturl"));
    je.set_comments(b.getString("comments"));
    return je;
}
}

```

4 toXMLString method

5 toBundle method

6 fromBundle method

This application relies heavily on the `Bundle` class ① for moving data from one `Activity` to another. (We'll explain this in more detail later in this chapter.)

A `String` member ② exists for each element in the job, such as `jobid` or `customer`.

The `toString()` method ③ is rather important, as it's used when displaying jobs in the `ManageJobs` `Activity` (also discussed later in the chapter).

The `toXMLString()` method ④ generates an XML representation of this `JobEntry`, complying with the `job` element defined in the previously presented DTD.

The `toBundle()` method ⑤ takes the data members of the `JobEntry` class and packages

them into a `Bundle`. This `Bundle` is then able to be passed between activities, carrying with it the required data elements. The `fromBundle()` static method  returns a `JobEntry` when provided with a `Bundle`. `toBundle()` and `fromBundle()` work together to assist in the passing of `JobEntry` objects (at least the data portion thereof) between activities. Note that this is one of many ways in which to move data throughout an application. Another method, as an example, is to have a globally accessible class instance to store data.

Now that you understand the `JobEntry` class, we'll move on to the `JobList` class, which is a class used to manage a collection of `JobEntry` objects.

JobList

When interacting with the server or presenting the available jobs to manage on the Android device, the field service application works with an instance of the `JobList` class. This class, like the `JobEntry` class, has both data members and helpful methods. The `JobList` class contains a typed `List` data member, which is implemented using a `Vector`. This is the only data member of this class, as shown in the following listing.

Listing 12.10. JobList.java

```
package com.msi.manning.UnlockingAndroid;
import java.util.List;
import org.xml.sax.InputSource;
import android.util.Log;
// additional imports omitted for brevity, see source code
public class JobList {
    private Context _context = null;
    private List<JobEntry> _joblist;
    JobList(Context context) {
        _context = context;
        _joblist = new Vector<JobEntry>(0);
    }
    int addJob(JobEntry job) {
        _joblist.add(job);
        return _joblist.size();
    }
    JobEntry getJob(int location) {
        return _joblist.get(location);
    }
    List<JobEntry> getAllJobs() {
        return _joblist;
    }
    int getJobCount() {
        return _joblist.size();
    }
    void replace(JobEntry newjob) {
        try {
            JobList newlist = new JobList();
            for (int i=0;i<getJobCount();i++) {
                JobEntry je = getJob(i);
                if (je.get_jobid().equals(newjob.get_jobid())) {
                    newlist.addJob(newjob);
                } else {
                    newlist.addJob(je);
                }
            }
        }
    }
}
```

1 List class imported for Vector

2 InputSource imported, used by XML parser

3 Familiar logging mechanism

4 Constructor

5 addJob/getJob methods

6 getAllJobs method

7 replace method

```

        this._joblist = newlist._joblist;
        persist();
    } catch (Exception e) {
    }
}

void persist() {                                ← 7 persist method
    try {
        FileOutputStream fos = _context.openFileOutput("chapter12.xml",
        Context.MODE_PRIVATE);
        fos.write("<?xml version=\"1.0\" encoding=\"UTF-8\" ?>\n".getBytes());
        fos.write("<joblist>\n".getBytes());
        for (int i=0;i<getJobCount();i++) {
            JobEntry je = getJob(i);
            fos.write(je.toXMLString().getBytes());
        }
        fos.write("</joblist>\n".getBytes());
        fos.flush();
        fos.close();
    } catch (Exception e) {
        Log.d("CH12",e.getMessage());
    }
}

static JobList parse(Context context) {          ← 8 parse method
    try {
        FileInputStream fis = context.openFileInput("chapter12.xml");
        if (fis == null) {
            return null;
        }
        InputSource is = new InputSource(fis);
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        XMLReader xmlreader = parser.getXMLReader();
        JobListHandler jlHandler =
new JobListHandler(null /* no progress updates when reading file */);
        xmlreader.setContentHandler(jlHandler);
        xmlreader.parse(is);
        fis.close();
        return jlHandler.getList();
    } catch (Exception e) {
        return null;
    }
}
}

```

The list of jobs is implemented as a `Vector`, which is a type of `List` 1. The XML structure containing job information is parsed with the SAX parser, so we need to be sure to import those required packages 2. `JobEntry` objects are stored in the typed `List` object named `_joblist` 3. Helper methods for managing the list are

included as `addJob()` and `getJob()` ④. The `getAllJobs()` method ⑤ returns the list of `JobEntry` items. Note that generally speaking, the application uses the `getJob()` method for individual `JobEntry` management, but the `getAllJobs()` method is particularly useful when we display the full list of jobs in the `ManageJobs` Activity, discussed later in this chapter.

The `replace()` method ⑥ is used when we've closed a job and need to update our local store of jobs. Note that after it has updated the local list

of `JobEntry` items, `replace()` calls the `persist()` ⑦ method, which is responsible for writing an XML representation of the entire list of `JobEntry` items to storage. This method invokes the `toXMLString()` method on each `JobEntry` in the list.

The `openFileOutput()` method creates a file within the application's private file area. This is essentially a helper method to ensure we get a file path to which we have full read/write privileges.

The `parse()` method ⑧ obtains an instance of a `FileInputStream` to gain access to the file and creates an instance of an `InputStream`, which is required by the SAX XML parser. In particular, take note of the `JobListHandler`. SAX is a callback parser, meaning that it invokes a user-supplied method to process events in the parsing process. It's up to the `JobListHandler` (in our example) to process the data as appropriate.

We have one more class to go before we can jump back to the higher-level functionality of our application. The next section takes a quick tour of the `JobListHandler`, which is responsible for putting together a `JobList` from an XML data source.

JobListHandler

As presented earlier, our application uses an XML data storage structure. This XML data can come from either the server or a local file on the filesystem. In either case, the application must parse this data and transform it into a useful form. This is accomplished through the use of the SAX XML parsing engine and the `JobListHandler`, which is shown in [listing 12.11](#). The `JobListHandler` is used by the SAX parser for our XML data, regardless of the data's source. Where the data comes from dictates how the SAX parser is set up and invoked in this application. The `JobListHandler` behaves slightly differently depending on whether the class's constructor includes a `Handler` argument. If the `Handler` is provided, the `JobListHandler` will pass messages back for use in a `ProgressDialog`. If the `Handler` argument is null, this status message passing is bypassed. When parsing data from the server, the `ProgressDialog` is employed; the parsing of a local file is done quickly and without user feedback. The rationale for this is simple—the network connection may be slow, and we need to show progress information to the user. An argument could be made for always showing the

progress of the parse operation, but this approach gives us an opportunity to demonstrate more conditionally operating code.

Listing 12.11. JobListHandler.java

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source code
public class JobListHandler extends DefaultHandler {
    Handler phandler = null;
    JobList _list;
    JobEntry _job;
    String _lastElementName = "";
    StringBuilder sb = null;
    Context _context;
    JobListHandler(Context c, Handler progressHandler) {           ← 1 JobListHandler
        _context = c;                                              constructor
        if (progressHandler != null) {                                ← 2 Check for
            phandler = progressHandler;                            progress handler
            Message msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Processing List");
            phandler.sendMessage(msg);
        }
    }
    public JobList getList() {                                     ← 3 getList method
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object) ("Fetching List");
        if (phandler != null) phandler.sendMessage(msg);
        return _list;
    }
    public void startDocument() throws SAXException {           ← 4 startDocument
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object) ("Starting Document");
    }
}
```

```

        if (phandler != null) phandler.sendMessage(msg);
        _list = new JobList(_context);
        _job = new JobEntry();
    }
    public void endDocument() throws SAXException {
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object) ("End of Document");
        if (phandler != null) phandler.sendMessage(msg);
    }
    public void startElement
        (String namespaceURI, String localName, String qName,
        Attributes atts) throws SAXException {
        try {
            sb = new StringBuilder("");
            if (localName.equals("job")) {
                Message msg = new Message();
                msg.what = 0;
                msg.obj = (Object) (localName);
                if (phandler != null) phandler.sendMessage(msg);
                _job = new JobEntry();
            }
        } catch (Exception ee) {
        }
    }
    public void endElement
        (String namespaceURI, String localName, String qName)
        throws SAXException {
        if (localName.equals("job")) {
            // add our job to the list!
            _list.addJob(_job);
            Message msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Storing Job # " + _job.get_jobid());
            if (phandler != null) phandler.sendMessage(msg);
            return;
        }
        // portions of the code omitted for brevity
    }
    public void characters(char ch[], int start, int length) {
        String theString = new String(ch,start,length);
        Log.d("CH12","characters[" + theString + "]");
        sb.append(theString);
    }
}

```

5 endDocument method

6 Check for end of job element

7 Build up String incrementally

The `JobListHandler` constructor ① takes a single argument of `Handler`. This value may be null. If null, Message passing is omitted from the operation. When reading from a local storage file, this `Handler` argument is null. When reading data from the server over

the internet, with a potentially slow connection, the `Message`-passing code is utilized to provide feedback for the user in the form of a `ProgressDialog`. The `ProgressDialog` code is shown later in this chapter in the discussion of the `RefreshJobs` Activity. A local copy of the `Handler` ② is set up when using the `ProgressDialog`, as described in ①.

The `getList()` ③ method is invoked when parsing is complete. The role of `getList()` is to return a copy of the `JobList` that was constructed during the parse process. When the `startDocument()` callback method ④ is invoked by the SAX parser, the initial class instances are established. The `endDocument()` method ⑤ is invoked by the SAX parser when all of the document has been consumed. This is an opportunity for the `Handler` to perform additional cleanup as necessary. In our example, a message is posted to the user by sending a `Message`.

For each element in the XML file, the SAX parser follows the same pattern: `startElement()` is invoked, `characters()` is invoked (one or more times), and `endElement()` is invoked. In the `startElement()` method, we initialize `StringBuilder` and evaluate the element name. If the name is “job,” we initialize the class-level `JobEntry` instance.

In the `endElement()` method, the element name is evaluated. If the element name is “job” ⑥, the `JobListHandler` adds this `JobEntry` to the `JobList` data member, `_joblist`, with a call to `addJob()`. Also in the `endElement()` method, the data members of the `JobEntry` instance (`_job`) are updated. Please see the full source code for more details.

The `characters()` method is invoked by the SAX parser whenever data is available for storage. The `JobListHandler` simply appends this string data to a `StringBuilder` instance ⑦ each time it's invoked. It's possible that the `characters()` method may be invoked more than once for a particular element's data. That's the rationale behind using a `StringBuilder` instead of a single `String` variable; `StringBuilder` is a more efficient class for constructing strings from multiple substrings.

After this lengthy but important look into the data structures and the accompanying explanations, it's time to return to the higher-level functionality of the application.

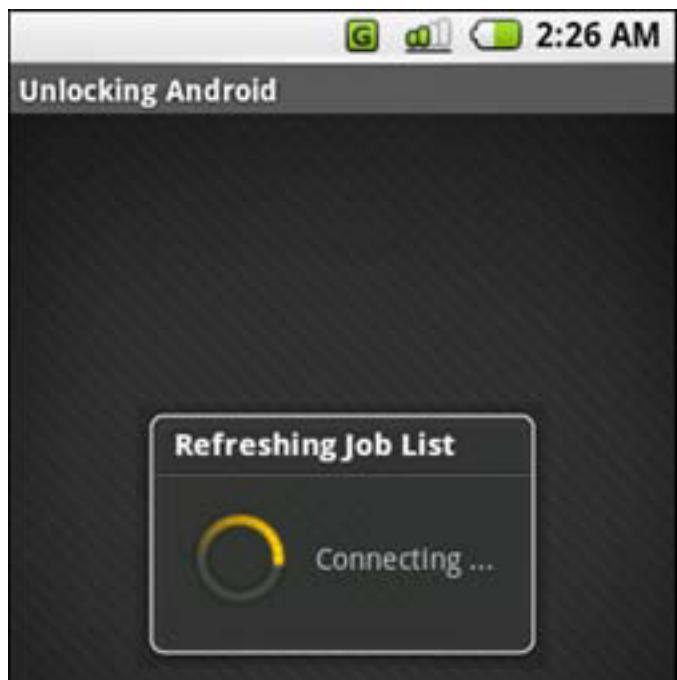
12.4. SOURCE CODE FOR MANAGING JOBS

Most of the time our mobile worker is using this application, he'll be reading through comments, looking up a job address, getting product information, and performing other aspects of working on a specific job. Our application must supply the functionality for the worker to accomplish each of these job-management tasks. We examine each of these `Activity`s in detail in this section. The first thing we review is fetching new jobs from the server, which gives us the opportunity to discuss the `JobListHandler` and the management of the jobs list used throughout the application.

12.4.1. RefreshJobs

The `RefreshJobs` `Activity` performs a simple yet vital role in the field service application. Whenever requested, the `RefreshJobs` `Activity` attempts to download a list of new jobs from the server. The UI is super simple—just a blank screen with a `ProgressDialog` informing the user of the application's progress, as shown in [figure 12.8](#).

Figure 12.8. The `ProgressDialog` in use during `RefreshJobs`



The code for `RefreshJobs` is shown in [listing 12.12](#). The code is straightforward, as most of the heavy lifting is done in the `JobListHandler`. This code's responsibility is to fetch configuration settings, initiate a request to the server, and put a mechanism in place for showing progress to the user.

[Listing 12.12. RefreshJobs.java](#)

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class RefreshJobs extends Activity {
    Prefs myprefs = null;
    Boolean bCancel = false;
    JobList mList = null;
    ProgressDialog progress;
    Handler progresshandler;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.refreshjobs);
        myprefs = new Prefs(this.getApplicationContext());
        myprogress = ProgressDialog.show(this, "Refreshing Job List",
            "Please Wait",true,false);
        progresshandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                switch (msg.what) {
                    case 0:
                        myprogress.setMessage("" + (String) msg.obj);
                        break;
                    case 1:
                        myprogress.cancel();
                        finish();
                        break;
                    case 2:    // error occurred
                        myprogress.cancel();
                        finish();
                        break;
                }
                super.handleMessage(msg);
            }
        };
    }
}
```

The diagram illustrates the flow of steps 1 through 6 from the code annotations:

- Step 1: Progress indicator (Annotation 1)
- Step 2: Set up ProgressDialog (Annotation 2)
- Step 3: Define Handler (Annotation 3)
- Step 4: Update UI with textual message (Annotation 4)
- Step 5: Handle cancel and cancel with error (Annotation 5)
- Step 6: Use openFileInput for stream (Annotation 6)

Annotations are placed near specific code snippets:

- Annotation 1: Next to the line `myprogress = ProgressDialog.show(this, "Refreshing Job List", "Please Wait",true,false);`
- Annotation 2: Next to the line `progresshandler = new Handler()`
- Annotation 3: Next to the line `super.handleMessage(msg);`
- Annotation 4: Next to the line `myprogress.setMessage("" + (String) msg.obj);`
- Annotation 5: Next to the line `myprogress.cancel();`
- Annotation 6: Next to the line `super.handleMessage(msg);`

```

};

Thread workthread = new Thread(new DoReadJobs());
workthread.start();
}

class DoReadJobs implements Runnable {
    public void run() {
        InputSource is = null;
        Message msg = new Message();
        msg.what = 0;
        try {
            //Looper.prepare();
            msg.obj = (Object) ("Connecting ...");
            progresshandler.sendMessage(msg);
            URL url = new URL(myprefs.getServer() +
                "getjoblist.php?identifier=" + myprefs.getEmail());
            is = new InputSource(url.openStream());
            SAXParserFactory factory = SAXParserFactory.newInstance();
            SAXParser parser = factory.newSAXParser();
            XMLReader xmlreader = parser.getXMLReader();
            JobListHandler jlHandler =
new JobListHandler(progresshandler);
            xmlreader.setContentHandler(jlHandler);
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Parsing ...");
            progresshandler.sendMessage(msg);
            xmlreader.parse(is);
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Parsing Complete");
            progresshandler.sendMessage(msg);
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Saving Job List");
            progresshandler.sendMessage(msg);
            jlHandler.getList().persist();           ← 13 Persist data
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Job List Saved.");
            progresshandler.sendMessage(msg);
            msg = new Message();
            msg.what = 1;
            progresshandler.sendMessage(msg);
        } catch (Exception e) {
            Log.d("CH12","Exception: " + e.getMessage());
            msg = new Message();
            msg.what = 2;      // error occurred
            msg.obj = (Object) ("Caught an error retrieving
                Job data: " + e.getMessage());
            progresshandler.sendMessage(msg);
        }
    }
}

```

7 Initiate DoReadJobs class instance

8 Create Message object

9 Define looping construct

10 Prepare status message

11 Prepare to parse data

12 Instantiate JobListHandler

13 Persist data

14 Set status flag for completion

15 Set status flag for error

A `ProgressDialog` 1 is used to display progress information to the user. There are a number of ways to display progress in Android. This is perhaps the most straightforward approach. A `Handler` is employed to process `Message` instances. Though the `Handler` itself is defined as an anonymous class, the code requires a reference to it 12 for passing to the `JobListHandler` when parsing, which is shown in 2. When instantiating the `ProgressDialog` , the arguments include

- Context
- Title of Dialog
- Initial Textual Message
- Indeterminate
- Cancelable

Using `true` for the `Indeterminate` parameter means that you're not providing any clue as to when the operation will complete (such as percentage remaining), just an indicator that something is still happening, which can be a best practice when you don't have a good handle on how long an operation may take. A new `Handler` 3 is created to process messages sent from the parsing routine, which will be introduced momentarily. An important class that has been mentioned but thus far not described is `Message`. This class is used to convey information between different threads of execution. The `Message` class has some generic data members that may be used in a flexible manner. The first of interest is the `what` member, which acts as a simple identifier, allowing recipients to easily jump to desired code based on the value of the `what` member. The most typical (and used here) approach is to evaluate the `what` data member via a `switch` statement.

In this application, a `Message` received with its `what` member equal to 0 represents a textual update message 4 to be displayed in the `ProgressDialog`. The textual data itself is passed as a `String` cast to an `Object` and stored in the `obj` data member of the `Message`. This interpretation of the `what` member is purely arbitrary. We could've used 999 as the value meaning textual update, for example. A `what` value of 1 or 2 indicates that the operation is complete 5 , and this `Handler` can take steps to initiate another thread of execution. For example, a value of 1 indicates successful completion, so the `ProgressDialog` is canceled, and the `RefreshJobs` Activity is completed with a call to `finish()`. The value of 2 for the `what` member has the same effect as a value of 1, but it's provided here as an example of handling different result conditions: for example, a failure response due to an encountered error. In a production-ready application, this step should be fleshed out to perform an additional step of instruction to the user and/or a retry step. Any `Message` not explicitly handled by the `Handler` instance should be passed to the super class 6 . In this way, system messages may be processed.

When communicating with a remote resource, such as a remote web server in our case, it's a good idea to perform **the** communications steps in a thread other than the primary

GUI thread. A new Thread **7** is created based on the `DoReadJobs` class

which implements the `Runnable` Java interface. A new `Message` object **8** is instantiated and initialized. This step takes place over and over throughout the `run()` method of the `DoReadJobs` class. It's important to not reuse a `Message` object, as they're literally passed and enqueued. It's possible for them to stack up in the receiver's queue, so reusing a `Message` object will lead to losing data or corrupting data at best and Thread synchronization issues or beyond at worst.

Why are we talking about a commented-out line of code **9**? Great question—because it caused so much pain in the writing of this application! A somewhat odd and confusing element of Android programming is the `Looper` class. This class provides static methods to help Java Threads to interact with Android. Threads by default don't have a message loop, so presumably `Messages` don't go anywhere when sent. The first call to make is `Looper.prepare()`, which creates a `Looper` for a Thread that doesn't already have one established. Then by placing a call to the `loop()` method, the flow of `Messages` takes place. Prior to implementing this class as a `Runnable` interface, we experimented with performing this step in the same thread and attempted to get the `ProgressDialog` to work properly. That said, if you run into funny Thread/Looper messages on the Android Emulator, consider adding a call to `Looper.prepare()` at the beginning of your Thread and then `Looper.loop()` to help `Messages` flow.

When we want to send data to the user to inform him of our progress, we update an instance of the `Message` class **10** and send it to the assigned `Handler`.

To parse an incoming XML data stream, we create a new `InputSource` from the URL stream **11**. This step is required for the SAX parser. This method reads data from the network directly into the parser without a temporary storage file.

Note that the instantiation of the `JobListHandler` **12** takes a reference to the `progresshandler`. This way the `JobListHandler` can (optionally) propagate messages back to the user during the parse process. Once the parse is complete,

the `JobListHandler` returns a `JobList` object, which is then persisted **13** to store the data to the local storage. Because this parsing step is complete, we let the `Handler` know by passing a `Message` **14** with the `what` field set to 1. If an exception occurs, we pass a message with `what` set to 2, indicating an error **15**.

Congratulations, your Android application has now constructed a `URL` object with persistently stored configuration information (user and server) and successfully connected over the internet to fetch XML data. That data has been parsed into a `JobList` containing `JobEntry` objects, while providing our patient mobile worker with feedback, and subsequently storing the `JobList` to the filesystem for later use. Now we want to work with those jobs, because after all, those jobs have to be completed for our mobile worker friend to make a living!

12.4.2. Managing jobs: the `ManageJobs` Activity

The `ManageJobs` Activity presents a scrollable list of jobs for review and action. At the top of the screen is a simple summary indicating the number of jobs in the list, and each individual job is enumerated in a `ListView`.

Earlier we mentioned the importance of the `JobEntry`'s `toString()` method:

```
public String toString() {  
    return this._jobid + ": " + this._customer + ": " + this._product;  
}
```

This method generates the string that's used to represent the `JobEntry` in the `ListView`, as shown in [figure 12.9](#).

Figure 12.9. The `ManageJobs` Activity lists downloaded jobs.



The layout for this Activity's View is simple: just a `TextView` and a `ListView`, as shown in the following listing.

Listing 12.13. `managejobs.xml`

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

```
<TextView      android:id="@+id/statuslabel"  
    android:text="list jobs here "  
    android:layout_height="wrap_content"  
    android:layout_width="fill_parent"  
/>  
  
<ListView    android:id="@+id/joblist"  
    android:layout_height="fill_parent"  
    android:layout_width="fill_parent"  
/>  
  
</LinearLayout>
```

The code in [listing 12.14](#) for the `ManageJobs` Activity connects a `JobList` to the GUI and reacts to the selection of a particular job from the `ListView`. In addition, this class demonstrates taking the result from another, synchronously invoked `Activity` and processing it according to its specific requirement. For example, when a job is completed and closed, that `JobEntry` is updated to reflect its new status.

Listing 12.14. `ManageJobs.java`, which implements the `ManageJobs` Activity

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class ManageJobs extends Activity implements OnItemClickListener {
    final int SHOWJOB = 1;
    Prefs myprefs = null;
    JobList _joblist = null;
    ListView jobListView;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.managejobs);
        myprefs = new Prefs(this.getApplicationContext());
        TextView tv =
            (TextView) findViewById(R.id.statuslabel);
        _joblist = JobList.parse(this.getApplicationContext());
        if (_joblist == null) {
            _joblist = new JobList(this.getApplicationContext());
        }
        if (_joblist.getJobCount() == 0){
            tv.setText("There are No Jobs Available");
        } else {
            tv.setText("There are " + _joblist.getJobCount() + " jobs.");
        }
        jobListView = (ListView) findViewById(R.id.joblist);
        ArrayAdapter<JobEntry> adapter = new ArrayAdapter<JobEntry>(this,
            android.R.layout.simple_list_item_1, _joblist.getAllJobs());
        jobListView.setAdapter(adapter);
        jobListView.setOnItemClickListener(this);
        jobListView.setSelection(0);
    }
}
```

Connect
TextView to UI

1 Parse
data in
storage

2 Handle
bad parse

Check for
empty JobList

Process click
events on List

Connect ListView to UI 3

Use a
built-in
list layout

Connect list with
dataevents on List

```

public void onItemClick(AdapterView parent,
    View v, int position, long id) {
    JobEntry je = _joblist.getJob(position);   ↪ 4 Fetch job from
    Log.i("CH12", "job clicked! [" + je.get_jobid() + "]");
    Intent jobintent = new Intent(this, ShowJob.class);
    Bundle b = je.toBundle();
    jobintent.putExtras(b);
    startActivityForResult(jobintent, SHOWJOB);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
    data) {
    switch (requestCode) {
        case SHOWJOB:
            if (resultCode == 1){           ↪ 7 Check
                Log.d("CH12", "Good Close, let's update our list");
                JobEntry je = JobEntry.fromBundle(data.getExtras());
                _joblist.replace(je);      ↪ 8 Update the list with
            }                           ↪ via replace method
            break;
    }
}

```

The diagram illustrates the flow of the onItemClick method. It starts with step 4: 'Fetch job from list by ordinal'. This leads to step 5: 'Use Bundle to store Job data'. Then, it moves to step 6: 'Start ShowJob Activity'. After the startActivityForResult call, the flow goes to step 7: 'Check return code'. Finally, step 8: 'Update the list with via replace method' is performed. A vertical bar on the right is labeled 'Prepare Intent for showing Job details'.

The objective of this code is to display a list of available jobs to the user in a `ListView` ③. To display the list of jobs, we must first parse the list stored on the device ①. Note that the `Context` argument is required to allow the `JobList` class access to the private file area for this application. If the parse fails, we initialize the `JobList` instance to a new, empty list. This is a somewhat simplistic way to handle the error without the GUI falling apart ②.

When a specific job is selected, its details are extracted via a call to the `getJob()` method ④. The job is stored in a `Bundle`, put into an `Intent` ⑤, and subsequently sent to the `ShowJob Activity` for display and/or editing ⑥. Note the use of the constant `SHOWJOB` as the last parameter of the `startActivityForResult()` method. When the called `Activity` returns, the second parameter to `startActivityForResult()` is “passed back” when the `onActivityResult()` method is invoked ⑦ and the return code checked. To obtain the changed `JobEntry`, we need to extract it from the `Intent` with a call to `getExtras()`, which returns a `Bundle`. This `Bundle` is turned into a `JobEntry` instance via the static `fromBundle()` method of

the `JobEntry` class. To update the list of jobs to reflect this changed `JobEntry`, call the `replace()` method 8.

More on bundles

You need to pass the selected job to the `ShowJob Activity`, but you can't casually pass an object from one `Activity` to another. You don't want the `ShowJob Activity` to have to parse the list of jobs again; otherwise you could simply pass back an index to the selected job by using the integer storage methods of a `Bundle`. Perhaps you could store the currently selected `JobEntry` (and `JobList` for that matter) in a global data member of the `Application` object, had you chosen to implement one. If you recall in [chapter 1](#) when we discussed the ability of Android to dispatch `Intents` to any `Activity` registered on the device, you want to keep the ability open to an application other than your own to perhaps pass a job to you. If that were the case, using a global data member of an `Application` object would never work! The likelihood of such a step is low, particularly considering how the data is stored in this application. This chapter's sample application is an exercise of evaluating some mechanisms you might employ to solve data movement when programming for Android. The chosen solution is to package the

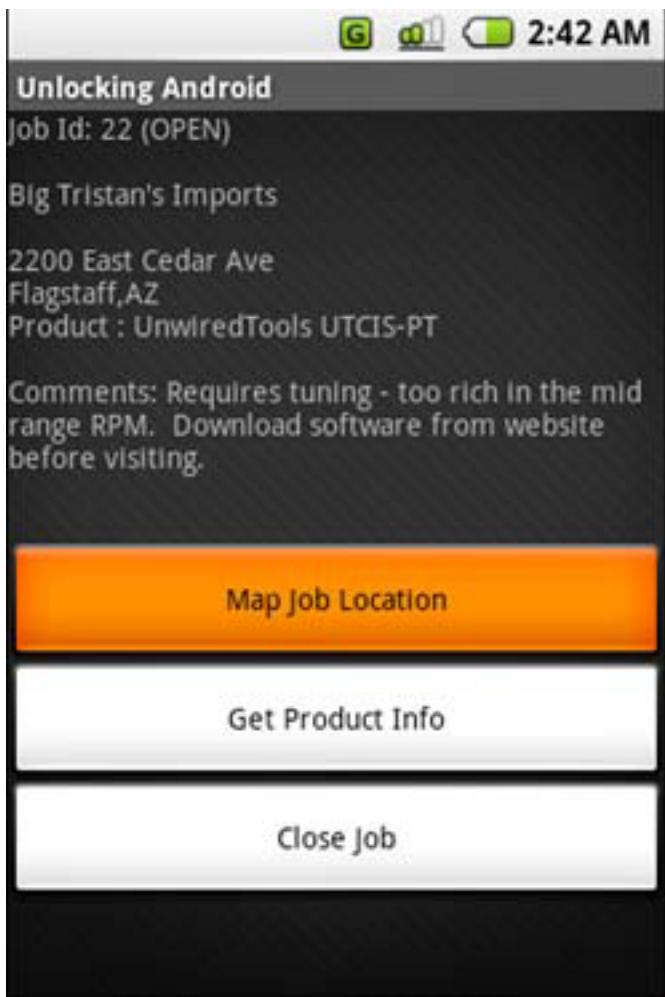
data fields of the `JobEntry` in a `Bundle` (5 in [listing 12.14](#)) to move a `JobEntry` from one `Activity` to another. In the strictest sense, you're not moving a real `JobEntry` object but a representation of a `JobEntry`'s data members. The net of this discussion is that this method creates a new `Bundle` by using the `toBundle()` method of the `JobEntry`.

Now that you can view and select the job of interest, it's time to look at just what you can do with that job. Before diving into the next section, be sure to review the `ManageJobs` code carefully to understand how the `JobEntry` information is passed between the two activities.

12.4.3. Working with a job with the `ShowJob Activity`

The `ShowJob Activity` is the most interesting element of the entire application, and it's certainly the screen most useful to the mobile worker carrying around his Android-capable device and toolbox. To help in the discussion of the various features available to the user on this screen, take a look at [figure 12.10](#).

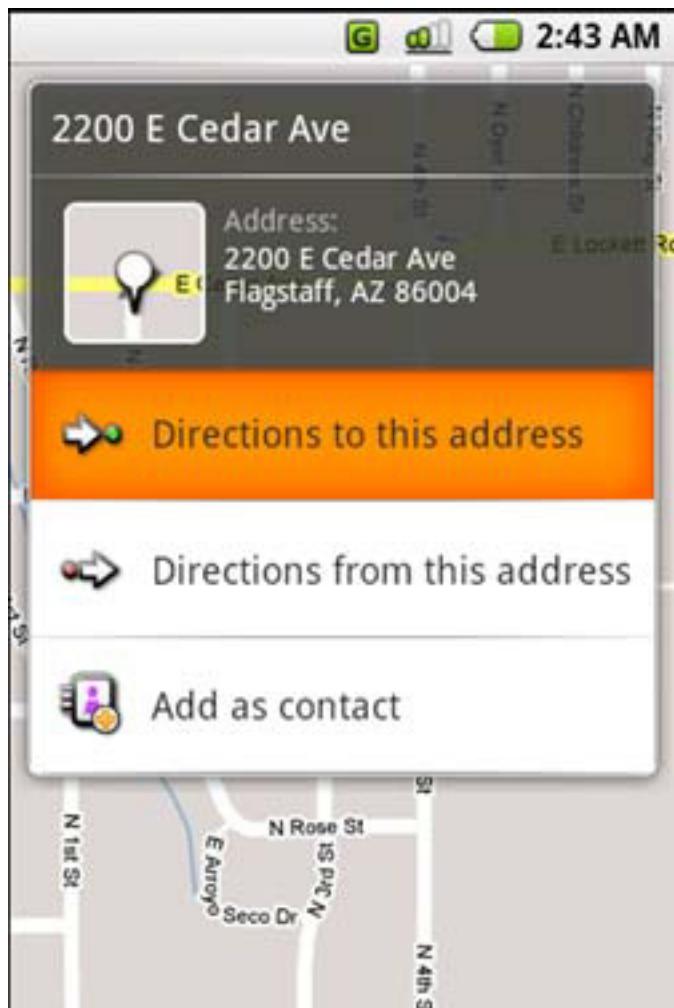
Figure 12.10. An example of a job shown in the `ShowJob Activity`



The layout is straightforward, but this time you have some `Buttons` and you'll be changing the textual description depending on the condition of a particular job's status. A `TextView` is used to present job details such as address, product requiring service, and comments. The third `Button` will have the `text` property changed, depending on the status of the job. If the job's status is marked as `CLOSED`, the functionality of the third button will change.

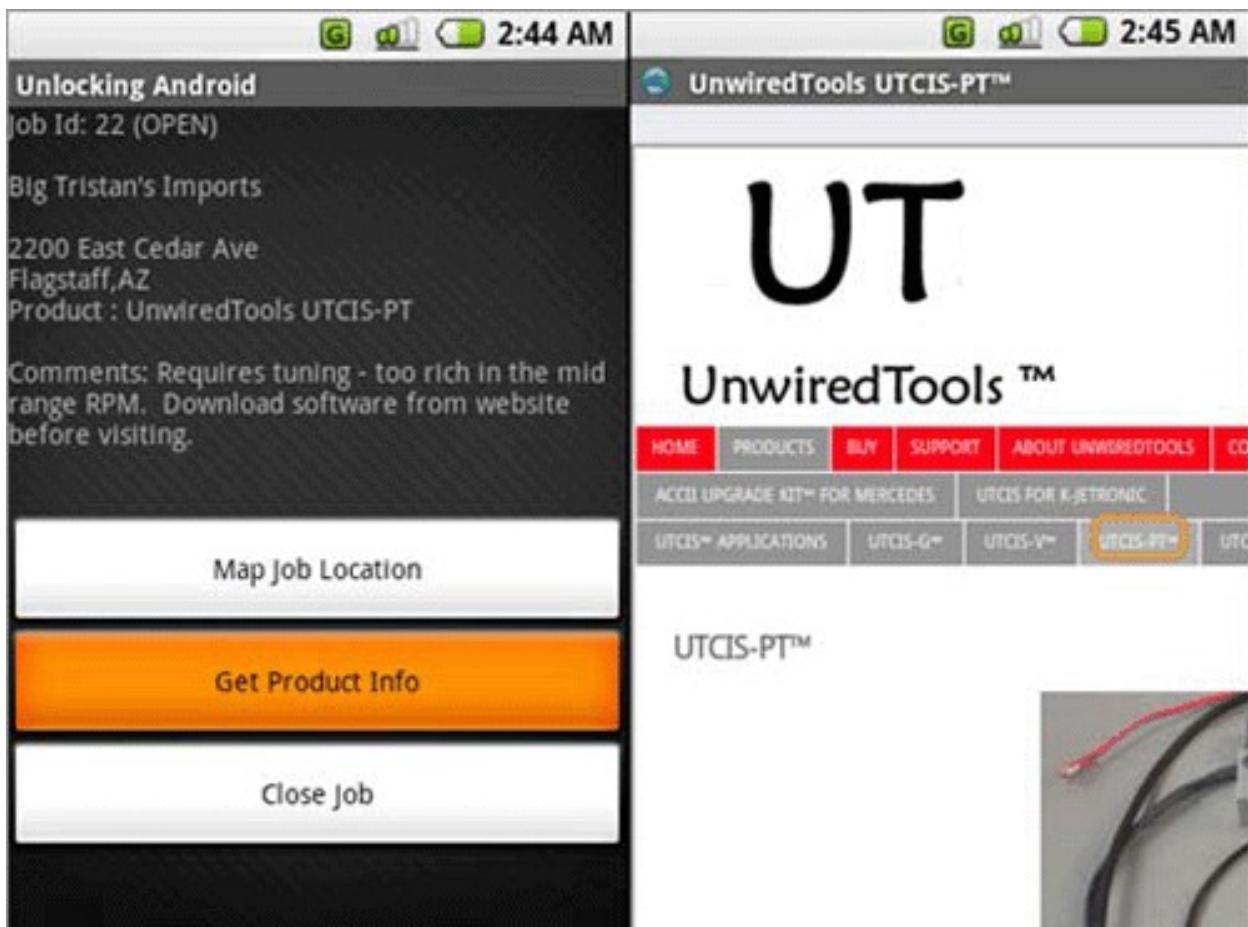
To support the functionality of this `Activity`, first the code needs to launch a new `Activity` to show a map of the job's address, as shown in [figure 12.11](#).

Figure 12.11. Viewing a job address in the Maps application



The second button, Get Product Info, launches a browser window to assist users in learning more about the product they're being called on to work with. [Figure 12.12](#) shows this in action.

Figure 12.12. Get Product Info takes the user to a web page specific to this job.



The third requirement is to allow the user to close the job or to view the signature if it's already closed; we'll cover the details in the next section on the `CloseJob` Activity.

Fortunately, the steps required for the first two operations are quite simple with Android—thanks to the `Intent`. The following listing and the accompanying annotations show you how.

Listing 12.15. ShowJob.java

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class ShowJob extends Activity {
    Prefs myprefs = null;
    JobEntry je = null;
    final int CLOSEJOBTASK = 1;
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.showjob);
        myprefs = new Prefs(this.getApplicationContext());
        StringBuilder sb = new StringBuilder();
        String details = null;
        Intent startingIntent = getIntent();           ← Get Intent
        if (startingIntent != null) {
            Bundle b = startingIntent.getExtras();
            if (b == null) {
                details = "bad bundle?";
            } else {
                je = JobEntry.fromBundle(b);
                sb.append("Job Id: " + je.get_jobid() + " (" + je.get_status()+
                    ")\n\n");
            }
        }
    }
}
```

← Extract
Bundle
from Intent

```

        sb.append(je.get_customer() + "\n\n");
        sb.append(je.get_address() + "\n" + je.get_city() + "," +
            je.get_state() + "\n");
        sb.append("Product : " + je.get_product() + "\n\n");
        sb.append("Comments: " + je.get_comments() + "\n\n");
        details = sb.toString();
    }
} else {
    details = "Job Information Not Found.";
    TextView tv = (TextView) findViewById(R.id.details);
    tv.setText(details);
    return;
}
TextView tv = (TextView) findViewById(R.id.details);
tv.setText(details);
Button bmap = (Button) findViewById(R.id.mapjob);
bmap.setOnClickListener(new Button.OnClickListener() {
    public void onClick(View v) {
        // clean up data for use in GEO query
        String address = je.get_address() + " " +
je.get_city() + " " +
            je.get_zip();
        String cleanAddress = address.replace(",", " ");
        cleanAddress = cleanAddress.replace(' ', '+');
        try {
            Intent geoIntent = new Intent("android.intent.action.VIEW",
                android.net.Uri.parse("geo:0,0?q=" +
                    cleanAddress));
            startActivityForResult(geoIntent);
        } catch (Exception ee) {
        }
    }
});
Button bproductinfo = (Button) findViewById(R.id.productinfo);
bproductinfo.setOnClickListener(new Button.OnClickListener() {

```

← **Update UI upon error and return**

← **Build and launch geo query**

Upon completion of the `CloseJob` Activity, the `onActivityResult()` callback is

invoked. When this situation occurs, this method receives a `Bundle` containing the data

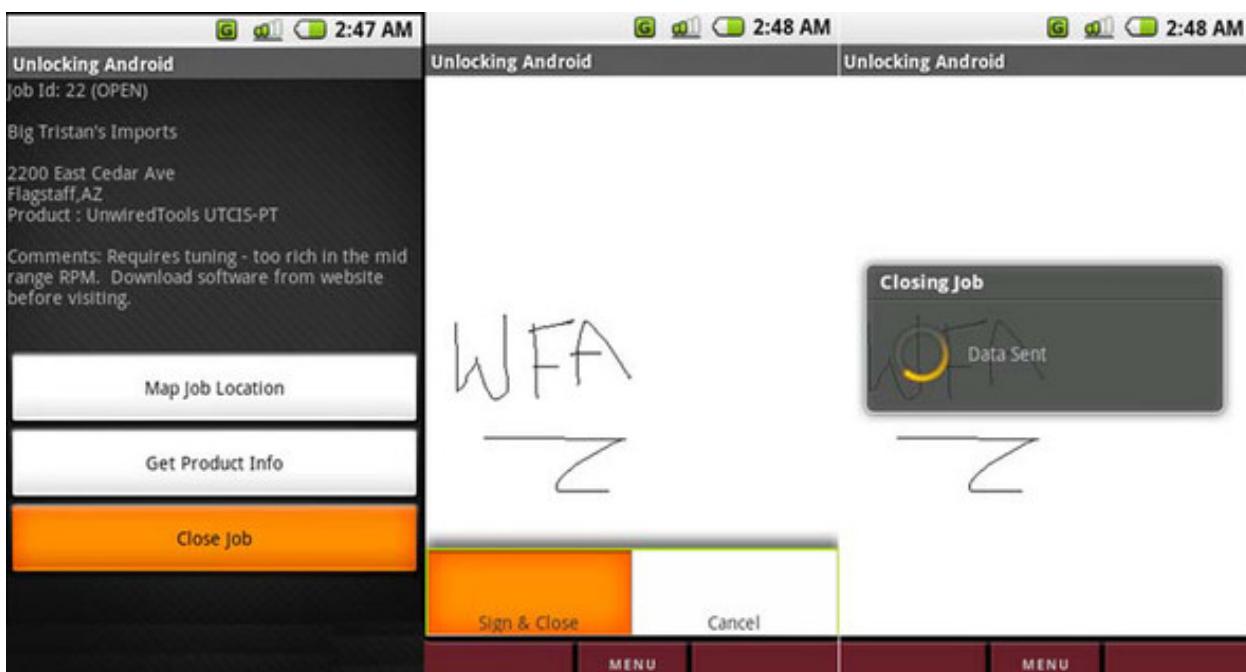
elements for the recently closed `JobEntry` ①. If you recall, the `ShowJob` Activity was launched “for result,” which permits a synchronous pattern, passing the result back to the caller. The requirement is to propagate this `JobEntry` data back up to the calling `Activity`, `ManageJobs`. Calling `setResult()` and passing the `Bundle` (obtained with `getExtras()`) fulfills this requirement.

Despite the simple appearance of some text and a few easy-to-hit buttons, the `ShowJob` Activity provides a significant amount of functionality to the user. All that remains is to capture the signature to close out the job. Doing so requires an examination of the `CloseJob` Activity.

12.4.4. Capturing a signature with the CloseJob Activity

Our faithful mobile technician has just completed the maintenance operation on the part and is ready to head off to lunch before stopping for another job on the way home, but first he must close out this job with a signature from the customer. To accomplish this, the field service application presents a blank screen, and the customer uses a stylus (or a mouse in the case of the Android Emulator) to sign the device, acknowledging that the work has been completed. Once the signature has been captured, the data is submitted to the server. The proof of job completion has been captured, and the job can now be billed. [Figure 12.13](#) demonstrates this sequence of events.

Figure 12.13. The `CloseJob Activity` capturing a signature and sending data to the server



This `Activity` can be broken down into two basic functions: the capture of a signature and the transmittal of job data to the server. Notice that this `Activity`'s UI has no layout resource. All of the UI elements in this `Activity` are generated dynamically, as shown in [listing 12.16](#). In addition, the `ProgressDialog` introduced in the `RefreshJobs Activity` is brought back for an encore, to let our mobile technician know that the captured signature is being sent when the Sign & Close menu option is selected. If the user selects Cancel, the `ShowJob Activity` resumes control. Note that the signature should be made prior to selecting the menu option.

Local queuing

One element not found in this sample application is the local queuing of the signature. Ideally this would be done in the event that data coverage isn't available. The storage of the image is quite simple; the perhaps more challenging piece is the logic on when to attempt to send the data again. Considering all the development of this sample application is done on the Android Emulator with near-perfect connectivity, it's of little concern here. But in the interest of best preparing you to write real-world applications, it's worth reminding you of local queuing in the event of communications trouble in the field.

Listing 12.16. CloseJob.java—GUI setup

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class CloseJob extends Activity {
    ProgressDialog myprogress;
    Handler progresshandler;
    Message msg;
    JobEntry je = null;
    private closejobView sc = null;
@Override
public void onCreate(Bundle icicle)  {
    super.onCreate(icicle);
    Intent startingIntent = getIntent();
    if (startingIntent != null) {
        Bundle b = startingIntent.getExtras()
        if (b != null) {
            je = JobEntry.fromBundle(b);
        }
    }
    sc = new closejobView(this);
    setContentView(sc);
    if (je == null) {

        finish();
    }
}
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);
    menu.add(0,0,"Sign & Close");
    menu.add(0,1,"Cancel");
    return true;
}
```

1 Instantiate instance of closejobView

2 Define available menus

```

public boolean onOptionsItemSelected(Menu.Item item) {
    Prefs myprefs = new Prefs(CloseJob.this.getApplicationContext());
    switch (item.getItemId()) {
        case 0:
            try {
                myprogress = ProgressDialog.show(this, "Closing Job",
                    "Saving Signature to Network",true,false);
                progresshandler = new Handler() {
                    @Override
                    public void handleMessage(Message msg) {
                        switch (msg.what) {
                            case 0:
                                myprogress.setMessage(" " + (String) msg.obj);
                                break;
                            case 1:
                                myprogress.cancel();
                                finish();
                                break;
                        }
                    }
                };
                super.handleMessage(msg);
            }
        );
        Thread workthread = new
        Thread(new DoCloseJob(myprefs));
        workthread.start();
    } catch (Exception e) {
        Log.d("closejob",e.getMessage());
        msg = new Message();
        msg.what = 1;
        progresshandler.sendMessage(msg);
    }
    return true;
    case 1:
        finish();
        return true;
    }
    return false;
}

```

Handle selected menu
3

4 Start Thread to CloseJob

Unlike previous activities in this chapter, the UI doesn't come from a design time-defined layout, but rather an instance of a `closejobView` 1 is the primary UI. The `closejobView` is defined in [listing 12.17](#).

The `onCreateOptionsMenu()` method 2 is an override of the base `View`'s method, allowing a convenient way to add menus to this screen. Note that two menus are added, one for Sign & Close and one for Cancel. The `onOptionsItemSelected()` method 3 is invoked when the user selects a menu item. A `ProgressDialog` and

accompanying `Handler` are instantiated when the user chooses the menu to close a job. Once the progress-reporting mechanism is in place, a new `Thread` is created and started

in order to process the steps required to close the job ④. Note that an instance of `Prefs` is passed in as an argument to the constructor, as that will be needed to store a signature, as we'll show in [listing 12.18](#).

The UI at this point is only partially set up; we need a means to capture a signature on the screen of our Android device. The next listing implements the class `closejobView`, which is an extension of the `View` class.

Listing 12.17. CloseJob.java—`closejobView` class

```
public class closejobView extends View {  
    Bitmap _bitmap;  
    Canvas _canvas;  
    final Paint _paint;  
    int lastX;  
    int lastY;  
    public closejobView(Context c) {  
        super(c);  
        _paint = new Paint();  
        _paint.setColor(Color.BLACK);  
        lastX = -1;  
    }  
    public boolean Save(OutputStream os){  
        try {  
            _canvas.drawText("Unlocking Android", 10, 10, _paint);  
            _canvas.drawText("http://manning.com/ableson", 10, 25, _paint);  
            _canvas.drawText("http://android12.msi-wireless.com",  
                10, 40, _paint);  
            _bitmap.compress(Bitmap.CompressFormat.JPEG, 100, os);  
            invalidate();  
            return true;  
        } catch (Exception e) {  
            return false;  
        }  
    }  
    @Override  
    protected void onSizeChanged(int w, int h, int oldw, int oldh) {  
        Bitmap img =  
            Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888);  
        Canvas canvas = new Canvas();  
        canvas.setBitmap(img);  
        if (_bitmap != null) {  
            canvas.drawBitmap(_bitmap, 0, 0, null);  
        }  
    }  
}
```

- ① `closejobView` extends base class `View`
- ② Required classes for drawing
- ③ Initialize drawing classes
- ④ Save method persists signature
- ⑤ Add contextual data to image

```
_bitmap = img;
_canvas = canvas;
_canvas.drawColor(Color.WHITE);
}
@Override
protected void onDraw(Canvas canvas) {
    if (_bitmap != null) {
        canvas.drawBitmap(_bitmap, 0, 0, null);
    }
}
@Override
public boolean onTouchEvent(MotionEvent event) {
    int action = event.getAction();
    int X = (int)event.getX();
    int Y = (int)event.getY();
    switch (action) {
        case MotionEvent.ACTION_UP:
            // reset location
            lastX = -1;
            break;
        case MotionEvent.ACTION_DOWN:
            if (lastX != -1){
                if ((int) event.getX() != lastX) {
                    _canvas.drawLine(lastX, lastY, X, Y, _paint);
                }
            }
            lastX = (int)event.getX();
            lastY = (int)event.getY();
            break;
        case MotionEvent.ACTION_MOVE:
            if (lastX != -1){
                _canvas.drawLine(lastX, lastY, X, Y, _paint);
            }
            lastX = (int)event.getX();
            lastY = (int)event.getY();
            break;
    }
    invalidate();
    return true;
}
}
```

5 Draw image on screen

6 Handle touch events

The `closeJobView` extends the base `View` class 1. The `Bitmap` and `Canvas` classes work together to form the drawing surface for this `Activity`. Note the call to the `Canvas.drawColor()` method, which sets the background color to white. When the `onDraw()` method is invoked, the canvas draws its associated bitmap with a call to `drawBitmap()` 5.

The logic for where to draw relies on the `onTouchEvent()` method **6**, which receives an instance of the `MotionEvent` class. The `MotionEvent` class tells what happened and where. `ACTION_UP`, `ACTION_DOWN`, and `ACTION_MOVE` are the events captured, with some logic to guide when and where to draw. Once the signature is complete,

the `Save()` method **3** is responsible for converting the contents of the image to a form usable for submission to the server. Note that additional text is drawn on the

signature **4**. In this case, it's little more than a shameless plug for this book's web page, but this could also be location-based data. Why is this important? Imagine someone forging a signature. It could happen, but it would be more challenging and of less value to a rogue mobile technician if the GPS/location data were stamped on the job, along with the date and time. When converting the image to our desired JPEG format, there's an additional input argument to this method—an `OutputStream`, used to store the image data. This `OutputStream` reference was an input argument to the `Save()` method.

Now that the UI has been created and a signature drawn on the screen, let's look at the code used to close the job. Closing the job involves capturing the signature and sending it to the server via an HTTP POST. The class `DoCloseJob` is shown in the following listing.

Listing 12.18. CloseJob.java—`DoCloseJob` class

```
class DoCloseJob implements Runnable {  
    Prefs _myprefs;  
    DoCloseJob(Prefs p) {  
        _myprefs = p;  
    }  
    public void run() {  
        try {  
            FileOutputStream os =  
                getApplication().openFileOutput("sig.jpg", 0);  
            sc.Save(os);  
            os.flush();  
            os.close();  
            // reopen to so we can send this data to server  
            File f = new  
                File(getApplicationContext().getFileStreamPath("sig.jpg").toString());  
            long flength = f.length();  
            FileInputStream is =  
                getApplication().openFileInput("sig.jpg");  
            byte data[] = new byte[(int) flength];  
            int count = is.read(data);  
            if (count != (int) flength) {  
                // bad read?  
            }  
            msg = new Message();  
            msg.what = 0;  
            msg.obj = (Object) ("Connecting to Server");  
            progresshandler.sendMessage(msg);  
            URL url = new URL(_myprefs.getServer() +  
                "/closejob.php?jobid=" + je.get_jobid());  
            URLConnection conn = url.openConnection();  
            conn.setDoOutput(true);  
            BufferedOutputStream wr = new  
                BufferedOutputStream(conn.getOutputStream());  
            wr.write(data);  
            wr.flush();  
            wr.close();  
            msg = new Message();  
        }  
    }  
}
```

Constructor uses
Prefs instance

1 Open file for
storing
signature

2 Construct
storage
URL

3 Write data
to server

```

        msg.what = 0;
        msg.obj = (Object) ("Data Sent");
        progresshandler.sendMessage(msg);
        BufferedReader rd = new BufferedReader(new
            InputStreamReader(conn.getInputStream()));
        String line = "";
        Boolean bSuccess = false;
        while ((line = rd.readLine()) != null) {
            if (line.indexOf("SUCCESS") != -1) {
                bSuccess = true;
            }
        }
        wr.close();
        rd.close();
        if (bSuccess) {
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Job Closed Successfully");
            progresshandler.sendMessage(msg);
            je.set_status("CLOSED");
            CloseJob.this.setResult(1, "", je.toBundle());
        } else {
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object) ("Failed to Close Job");
            progresshandler.sendMessage(msg);
            CloseJob.this.setResult(0);
        }
    } catch (Exception e) {
        Log.d("CH12", "Failed to submit job close signature: " +
        e.getMessage());
    }
    msg = new Message();
    msg.what = 1;
    progresshandler.sendMessage(msg);
}
}

```



The diagram illustrates the sequence of steps 4 through 7. Step 4, 'Read server response', is indicated by an arrow pointing to the code where the server's response is read via a BufferedReader. Step 5, 'Check for successful processing', is indicated by an arrow pointing to the code where the 'bSuccess' variable is checked. Step 6, 'Update local JobEntry status', is indicated by an arrow pointing to the code where the JobEntry status is updated to 'CLOSED'. Step 7, 'Set result and store updated JobEntry', is indicated by an arrow pointing to the code where the job result is set and the updated JobEntry is stored.

At this point, we have a signature on the screen and need to capture it. A

`new FileOutputStream` 1 is obtained for a file on the local filesystem, and the signature is written to this file. We're now ready to transmit this file to the server—remember, we want to bill the client as soon as possible for work completed!

In preparation for sending the signature to the server, the signature file contents are read into a byte array via an instance of a `FileInputStream`. Using the `Prefs` instance to

get specific configuration information, a URL 2 is constructed in order to `POST` data to the server. The query `String` of the URL contains the `jobid`, and the `POST` data contains

the image itself. A `BufferedOutputStream` **3** is employed to `POST` data, which consists of the captured signature in JPEG format.

Once the job data and signature have been sent to the server, the response data is read back from the server **4**. A specific string indicates a successful transmission **5**.

Upon successful closing, the `JobEntry` status member is marked as CLOSED **6**, and this `JobEntry` is converted to a `Bundle` so that it may be communicated to the caller by invoking the `setResult()` method **7**. Once the `Handler` receives the “I’m done” message and the `Activity` finishes, this data is propagated back to the `ShowJob` and all the way back to the `ManageJob Activity`.

And that thankfully wraps up the source code review for the Android side of things! There were some methods omitted from this text to limit this already very long chapter, so please be sure to examine the full source code. Now it’s time to look at the server application.

12.5. SERVER CODE

A mobile application often relies on server-side resources, and our field service application is no exception. This isn’t a book on server-side development techniques, server-related code, and discussion, so we’ll present these things briefly. We’ll introduce the UI and the accompanying database structure that makes up our list of job entries, and then we’ll review the two server-side transactions that concern the Android application. The server code relies on open source staples: MySQL and PHP. Let’s get started with the interface used to enter new jobs, used by the dispatcher.

12.5.1. Dispatcher user interface

Before jumping into any server code–specific items, it’s important to understand how the application is organized. All jobs entered by a dispatcher are assigned to a particular mobile technician. That identifier is interpreted as an email address, as seen in the Android example where the user ID was used throughout the application. Once the user ID is specified, all of the records revolve around that data element. For example, [figure 12.14](#) demonstrates this by showing the jobs assigned to the author, fableson@msiservices.com.

Figure 12.14. The server-side dispatcher screen

Unlocking Android, Chapter 12 Sample Application

For assistance with this application, please contact [Frank Ableson](#) of MSI Services, Inc.

Job List for [fableson@msiservices.com].

Job Id#	Customer	Address	City	State Zip	Product	Product URL	Comments	Status
10	Path of Growth, LLC	123 Main Street	Chester	NU 07930	Wireless Router	http://turco.com	SID broadcast not working	CLOSED
12	Indy Products	49 Route 206	Stanhope	NU 07874	Water Cooler	http://whirlpool.com	Water is not cold enough	CLOSED
21	Shin's Boats, Inc	1 Orchard Lane	Chester	NU 07930	Cigarette Boat	http://shincraft.com/	needs a light	CLOSED
22	Big Tritan	2200 East Cedar Ave	Flagstaff	AZ 86004	UniviewTools UTCIS-PT	http://uniview-tools.com	Requires tuning - too rich in the mid range RPM. Download software from website before visiting.	CLOSED
23	DJ's Ices	17 Route 206	Stanhope	NU 07874	Gelato Machine	http://ice.com	Ice pops	CLOSED
24	Meyer Grocer	144 Whitehall Road	Andover	NU 07821	Roboteller	http://roboteller.com	Required firmware upgrade.	CLOSED
25	Google	123 Main Street	Somewhere	CA 12345	Android	http://google.com	test	CLOSED

[Export Your Job List](#)

Add a [Job](#)

[Home](#)

MSI Wireless is a division of [MSI Services](#).
Check out [Unlocking Android](#)

Note

This application is available for you to test. It's located at <http://android12.msi-wireless.com>. Sign on and add jobs for your email address.

Let's now turn our attention to the underlying data structure, which contains the list of jobs.

12.5.2. Database

As mentioned earlier in [section 12.1.3](#), the database in use in this application is MySQL,² with a single database table called `tbl_jobs`. The SQL to create this table is provided in the next listing.

² For more on development using MySQL, try the developer zone: <http://dev.mysql.com/>.

Listing 12.19. Data definition for `tbl_jobs`

```

CREATE TABLE IF NOT EXISTS 'tbl_jobs' (
  'jobid' int(11) NOT NULL auto_increment,          ← 1 Unique record ID
  'status' varchar(10) NOT NULL default 'OPEN',
  'identifier' varchar(50) NOT NULL,                ← 2 User identification
  'address' varchar(50) NOT NULL,
  'city' varchar(30) NOT NULL,
  'state' varchar(2) NOT NULL,
  'zip' varchar(10) NOT NULL,
  'customer' varchar(50) NOT NULL,
  'product' varchar(50) NOT NULL,
  'producturl' varchar(100) NOT NULL,              ← 3 Product URL
  'comments' varchar(100) NOT NULL,
  UNIQUE KEY 'jobid' ('jobid')
) ENGINE=MyISAM DEFAULT CHARSET=ascii AUTO_INCREMENT=25 ;

```

Each row in this table is uniquely identified by the `jobid` 1, which is an auto-incrementing integer field. The `identifier` field 2 corresponds to the user ID/email of the assigned mobile technician. The `producturl` field 3 is designed to be a specific URL to assist the mobile technician in the field in quickly gaining access to helpful information for completing the assigned job.

The next section provides a road map to the server code.

12.5.3. PHP dispatcher code

The server-side dispatcher system is written in PHP and contains a number of files working together to create the application. [Table 12.3](#) presents a brief synopsis of each source file to help you navigate the application if you choose to host a version of it yourself.

Table 12.3. Server-side source code

Source file	Description
<code>addjob.php</code>	Form for entering new job information
<code>closejob.php</code>	Used by the Android application to submit a signature
<code>db.php</code>	Database connection information
<code>export.php</code>	Used to export list of jobs to a CSV file
<code>footer.php</code>	Used to create a consistent look and feel for the footer of each page
<code>getjoblist.php</code>	Used by the Android application to request a job XML stream
<code>header.php</code>	Used to create a consistent look and feel for the header of each page

Source file	Description
index.php	Home page, including the search form
manage.php	Used to delete jobs on the web application
savejob.php	Used to save a new job (called from addjob.php)
showjob.php	Used to display job details and load into a form for updating
showjobs.php	Displays all jobs for a particular user
updatejob.php	Used to save updates to a job
utils.php	Contains various routines for interacting with the database

Of all these files, only two concern the Android application. We'll discuss them in the next section.

12.5.4. PHP mobile integration code

When the Android application runs the `RefreshJobs` Activity, the server side generates an XML stream. Without going into excessive detail on the server-side code, we explain the `getjoblist.php` file in the following listing.

Listing 12.20. getjoblist.php

```
<?
require('db.php');
require('utils.php');
$theuser = $_GET['identifier'];
print (getJobsXML($theuser));
?>
```

The `getJobsXML()` function retrieves data from the database and formats each row into an XML representation. It wraps the list of XML-wrapped job records in the `<joblist>` tags along with the `<?xml ...>` header declaration to generate the expected XML structure used by the Android application. Remember, this is the data ultimately parsed by the SAX-based `JobListHandler` class, as shown in [listing 12.11](#).

The other transaction that's important to our Android field service application is the `closejob.php` file, examined in the next listing.

Listing 12.21. closejob.php

```
<?
```

```

require('db.php');

require('utils.php');

$data = file_get_contents('php://input');

$jobid = $_GET['jobid'];

$f = fopen("~/pathofiles/sigs/".$jobid.".jpg", "w");

fwrite($f, $data);

fclose($f);

print(closeJob($_GET['jobid']));

?>

```

The POSTED image data is read via the `file_get_contents()` function. The secret is the special identifier of `php://input`. This is the equivalent of a binary read. This data is read into a variable named `$data`. The `jobid` is extracted from the query string. The image file is written out to a directory that contains signatures as JPEG files, keyed by the `jobid` as part of the filename. When a job has been closed and the signature is requested by the Android application, this file is requested in the Android browser. The `closeJob()` function (implemented in `utils.php`) updates the database to mark the selected job as CLOSED.

That wraps up the review of the source code for this chapter's sample application.

12.6. SUMMARY

The intent of the sample application was to tie together many things learned in previous chapters into a composite application. Our field service application has real-world applicability to the kind of uses an Android device is capable of bringing to fruition. Is this sample application production ready? Of course not, but almost! That, as they say, is an exercise for the reader.

Starting with a simple splash screen, this application demonstrates the use of `Handlers` and displaying images stored in the resources section of an Android project. Moving along to the main screen, a simple UI leads to different activities useful for launching various aspects of the realistic application.

Communications with the server involve downloading XML data, while showing the user a `ProgressDialog` along the way. Once the data stream commences, the data is parsed by the SAX XML parser, using a custom `Handler` to navigate the XML document.

We demonstrated that managing jobs in a `ListView` is as easy as tapping on the desired job in the list. The next screen, the `ShowJobs Activity`, allows even more functionality, with the ability to jump to a `Map` showing the location of the job and even a specific product information page customized to this job. Both of those functions are as simple as preparing an `Intent` and a call to `startActivity()`.

Once the mobile technician completes the job in the field, the `CloseJob Activity` brings the touch-screen elements into play by allowing the user to capture a signature from his customer. That digital signature is then stamped with additional, contextual information and transmitted over the internet to prove the job was done. Jumping back to what you learned earlier, it would be straightforward to add location-based data to further authenticate the captured signature.

The chapter wrapped up with a quick survey of the server-side components to demonstrate some of the steps necessary to tie the mobile and the server sides together.

The sample application is hosted on the internet and is free for you to test out with your own Android application, and the full source code is provided for the Android and server applications discussed in this chapter.

Now that we've shown what can be accomplished when exercising a broad range of the Android SDK, the next chapter takes a decidedly different turn, as we explore the underpinnings of Android a little deeper and look at building native C applications for the Android platform.