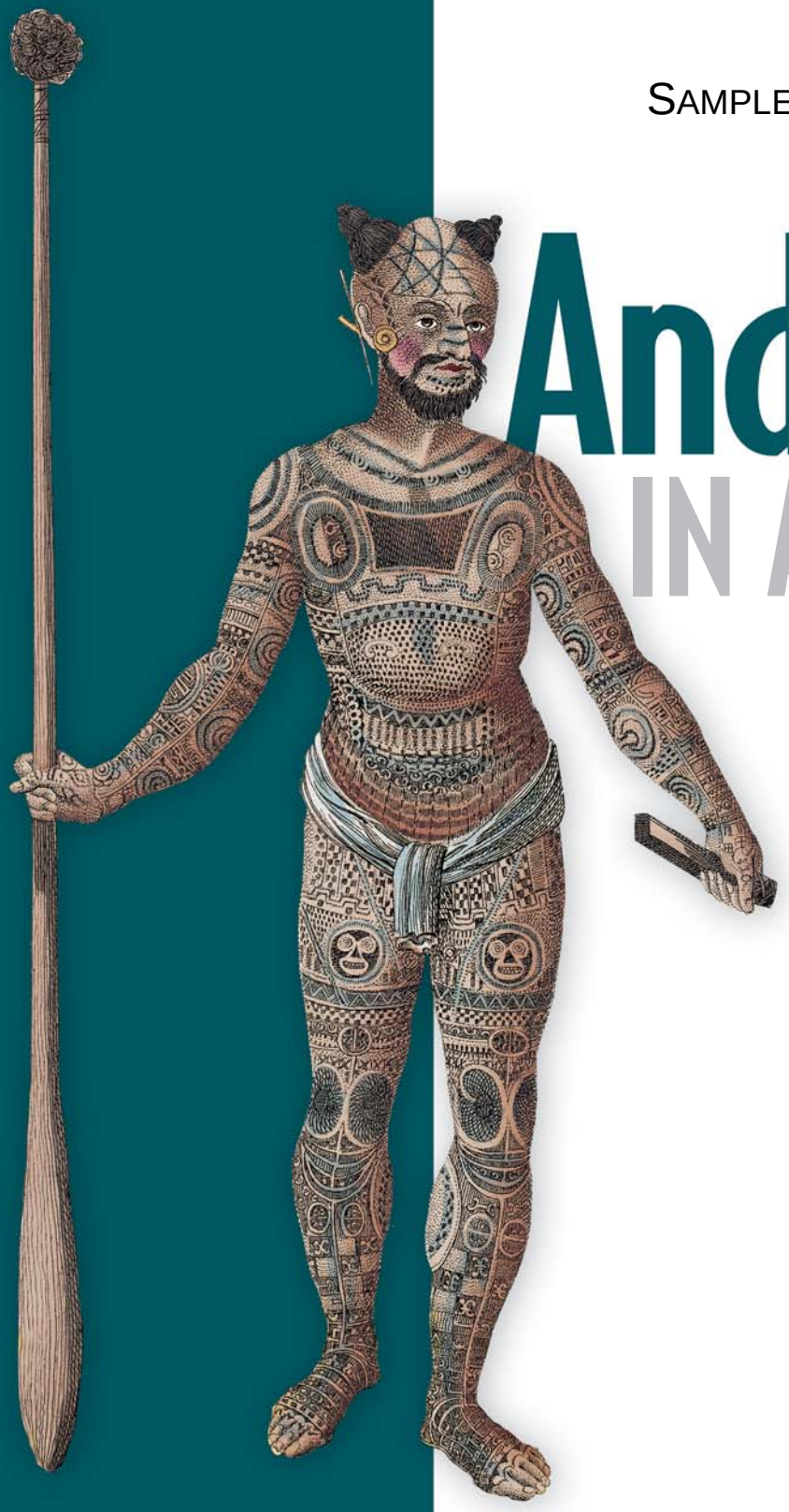

SAMPLE CHAPTER

Android IN ACTION

THIRD EDITION

W. Frank Ableson
Robi Sen
Chris King
C. Enrique Ortiz

MANNING

Android in Action, Third Edition

by W. Frank Abelson
Robi Sen
Chris King
C. Enrique Ortiz

Chapter 1

brief contents

1 Introducing Android

This chapter covers

- Exploring Android, the open source phone and tabtet platform
- Android `Intent`s, 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

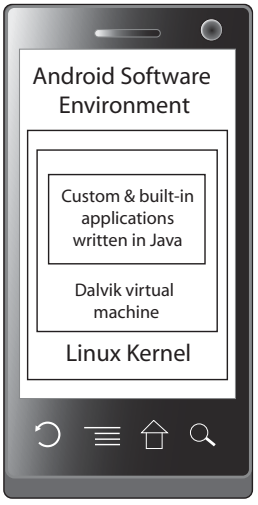

Figure 1.1 Android is software only. By leveraging its Linux kernel to interface with the hardware, Android runs on many different devices from multiple cell phone manufacturers. Developers write applications in Java.

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.

> **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 non-trivial 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)*.

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

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

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

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.

> **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.

(continued)
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.

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.

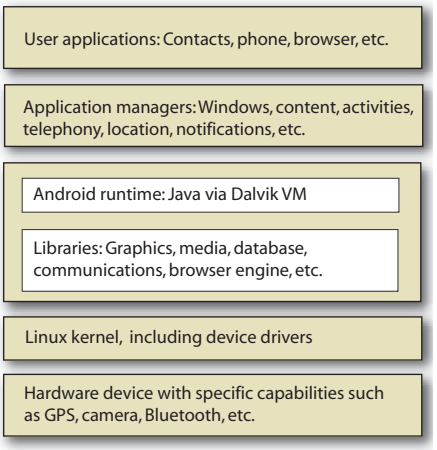

Figure 1.3 The Android stack offers an impressive array of technologies and capabilities.

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." `Intent`s 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 `Intent`s are important, and then we'll describe how `Intent`s 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 `Intent`s? 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.

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 `Activitys`, `Services`, and registered `BroadcastReceivers` (more on these in section 1.5) and

Table 1.1 Commonly employed URIs in Android

Type of information	URI data
Contact lookup	content://contacts/people
Map lookup/search	Geo:0,0?q=23+Route+206+Stanhope+NJ
Browser launch to a specific website	http://www.google.com/

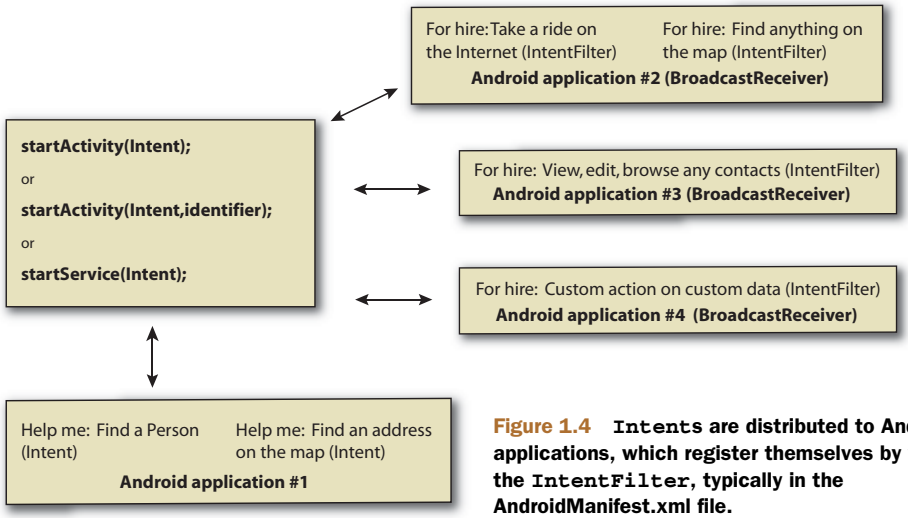

Figure 1.4 `Intents` are distributed to Android applications, which register themselves by way of the `IntentFilter`, typically in the AndroidManifest.xml file.

dispatches the `Intent` to the most appropriate recipient. Figure 1.4 depicts this relationship among `Intents`, `IntentFilters`, and `BroadcastReceivers`.

`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 `Intent`s 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 `Intent`s we've discussed thus far are known as *implicit* `Intent`s, which rely on the `IntentFilter` and the Android environment to dispatch the `Intent` to the appropriate recipient. Another kind of `Intent` is the e*xplicit* `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 `Intent`s 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.

`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*.

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 `Activity`s with specific `IntentFilter`s 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.

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

You say Intent; I say Intent

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

Some `Intent`s 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 `Intent`s, which are used with the `startActivity` and `startActivityForResult` methods.

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

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

`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 `View`s and are typically defined in XML layout files. Chapter 3 goes into more detail on `View`s and `Resources`.

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`. `Fragment`s, which are related to `Activity`s and have their own life cycle, provide more granular application control than `Activity`s. `Fragment`s 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 `Service`s 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;
```

1 Extend Service class

```
    @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 binding request
return null;
}

}
```

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 AndroidManfest.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 `intent-filter` 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" />
  <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" >
      <intent-filter>
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
      </intent-filter>
    </receiver>
  </application>
</manifest>
```

Required permission (1)

(2) Receiver tag; note dot prefix

Certain tasks within the Android platform require the application to have a designated privilege. To give an application the required permissions, use the `<uses-permission>` 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

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.

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.

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

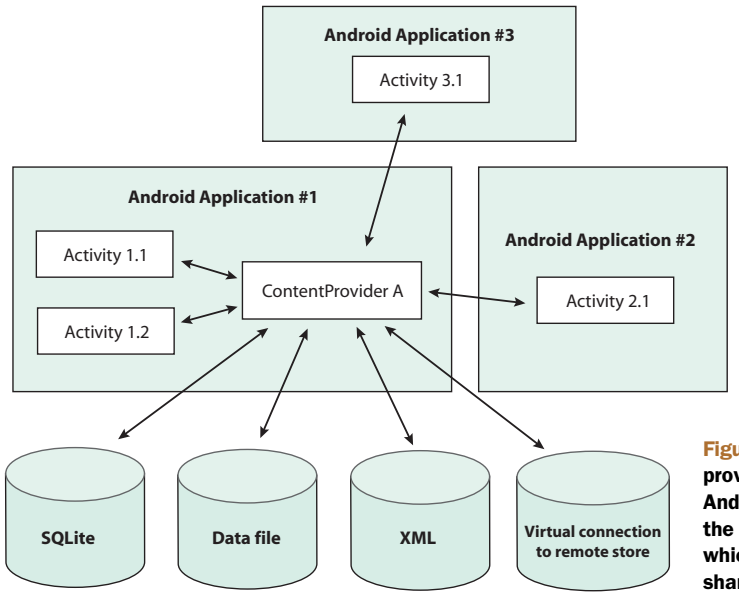

Figure 1.5 The content provider is the data tier for Android applications and is the prescribed manner in which data is accessed and shared on the device.

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.

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 `Intent`s 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 `Intent`s. AndroidManfest.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.

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 `AWhereDoYouLive`. 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>
```

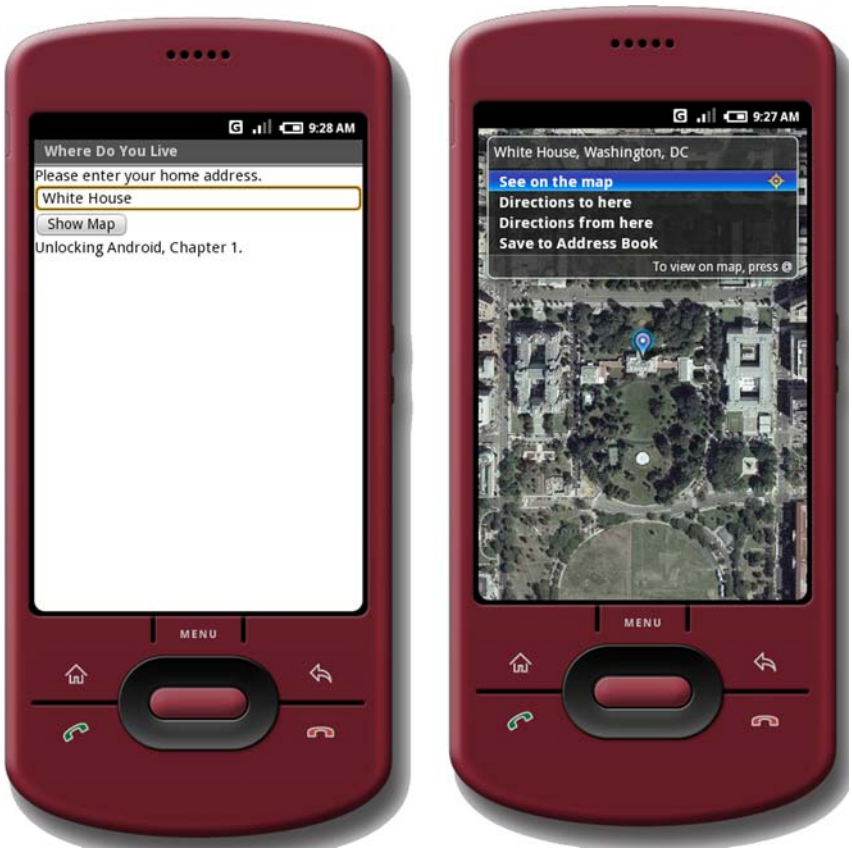

Figure 1.7 This Android application demonstrates a simple `Activity` and an `Intent`.

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.[2] Tablets will play a major role in both the consumer and enterprise spaces. The opportunities for tablet application development seem endless.

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.[3]

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:

- `Activity`s and `Fragment`s
- The Action bar
- Drag and drop
- Custom notifications
- Loaders
- Bluetooth

[2] www.yankeegroup.com/ResearchDocument.do?id=55390
[3] http://mng.bz/680r

This book will cover the major aspects of tablet development using Android 3.0, starting with `Activity`s and `Fragment`s. 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 `Intent`s in a slightly different manner, but the core facility of using `Intent`s to control application behavior enables the innovative and flexible Android environment. `Intent`s 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 `Intent`s 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.

MOBILE TECHNOLOGY

Android IN ACTION THIRD EDITION

Ableson • Sen • King • Ortiz

When it comes to mobile apps, Android can do almost anything—and with this book, so can you! Android, Google's popular mobile operating system and SDK for tablets and smart phones, is the broadest mobile platform available. It is Java-based, HTML5-aware, and loaded with the features today's mobile users demand.

Android in Action, Third Edition takes you far beyond "Hello Android." You'll master the SDK, build WebKit apps using HTML 5, and even learn to extend or replace Android's built-in features. You'll find interesting examples on every page as you explore cross-platform graphics with RenderScript, the updated notification system, and the Native Development Kit. This book also introduces important tablet concepts like drag and drop, fragments, and the Action Bar, all new in Android 3.

What's Inside

- Covers Android 3.x
- SDK and WebKit development from the ground up
- Driving a robot with Bluetooth and sensors
- Image processing with Native C code

This book is written for hobbyists and developers. A background in Java is helpful—no prior experience with Android is assumed.

Frank Ableson and **Robi Sen** are entrepreneurs focused on mobile and web products, and on novel wireless technologies, respectively. **Chris King** is a senior mobile engineer and **C. Enrique Ortiz** a mobile technologist, developer, and author.

For access to the book's forum and a free ebook for owners of this book, go to manning.com/AndroidinActionThirdEdition

"Gold standard of Android training books."
—Gabor Paller, Ericsson

"Still the best single book for both beginners and experts."
—Matthew Johnson
Sabaki Engineering

"Fully covers most Android tablet functionalities."
—Loïc Simon, SII

ISBN 13: 978-1-61729-050-3
ISBN 10: 1-61729-050-5
54999
9 781617 290503

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]