

SAMPLE CHAPTER

Unlocking

Android

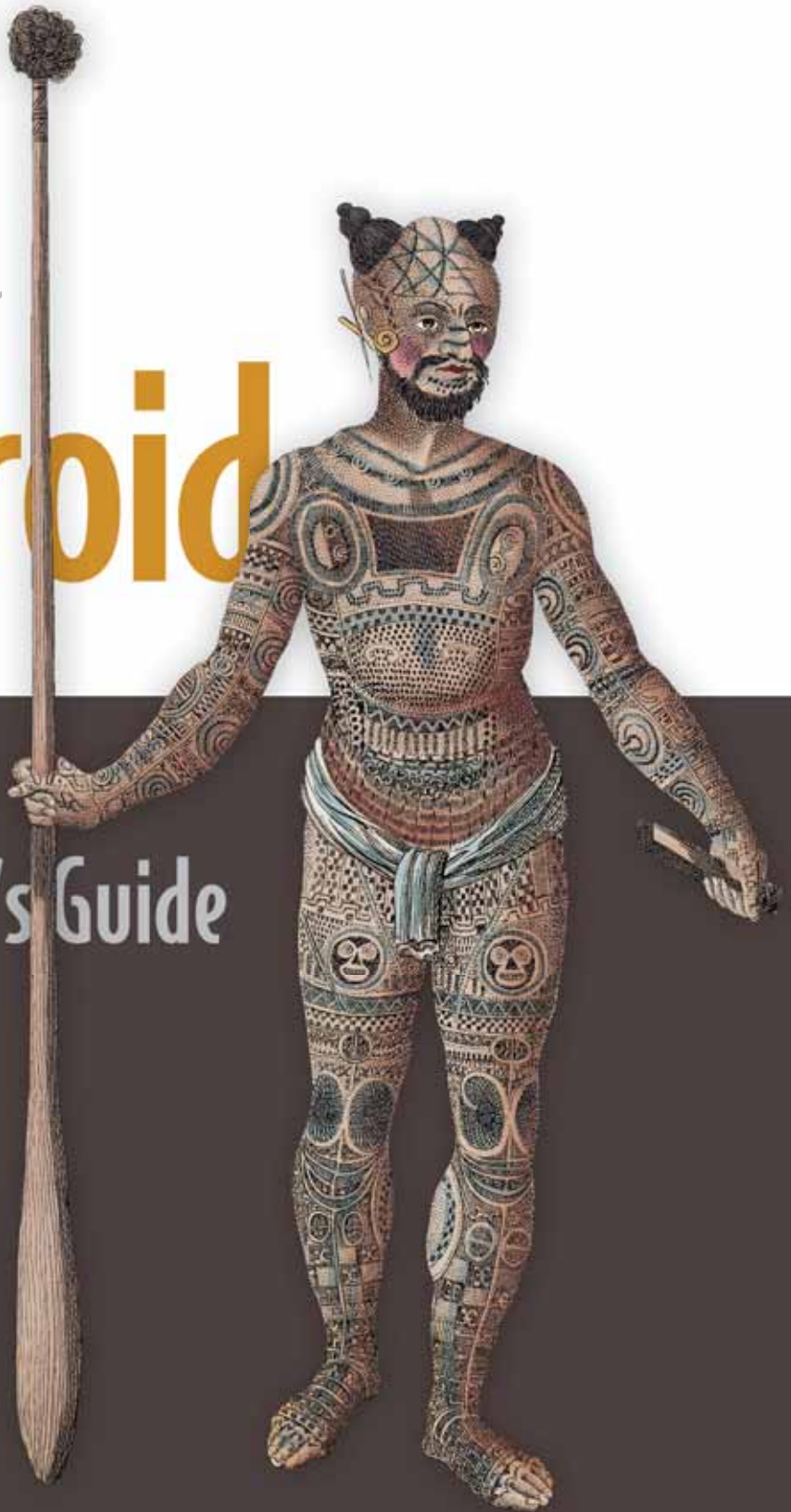
A Developer's Guide

Frank Ableson

Charlie Collins

Robi Sen

FOREWORD BY DICK WALL



 MANNING



Unlocking Android

by W. Frank Ableson
Charlie Collins
and Robi Sen

Chapter 1

Copyright 2009 Manning Publications

brief contents

PART 1	WHAT IS ANDROID?-THE BIG PICTURE	1
	1 ■ Targeting Android	3
	2 ■ Development environment	32
PART 2	EXERCISING THE ANDROID SDK.....	57
	3 ■ User interfaces	59
	4 ■ Intents and services	97
	5 ■ Storing and retrieving data	126
	6 ■ Networking and web services	167
	7 ■ Telephony	195
	8 ■ Notifications and alarms	211
	9 ■ Graphics and animation	226
	10 ■ Multimedia	251
	11 ■ Location, location, location	266
PART 3	ANDROID APPLICATIONS.....	293
	12 ■ Putting it all together—the Field Service Application	295
	13 ■ Hacking Android	341

Part 1

What is Android? —The Big Picture

Android promises to be a market-moving technology platform—not just because of the functionality available in the platform but because of how the platform has come to market. Part 1 of this book brings you into the picture as a developer of the open source Android platform.

We begin with a look at the Android platform and the impact it has on each of the major “stakeholders” in the mobile marketplace (chapter 1). We then bring you on board to developing applications for Android with a hands-on tour of the Android development environment (chapter 2).

Targeting Android



This chapter covers:

- Examining Android, the open source mobile platform
- Activating Android
- Rapidly changing smartphones

You've heard about Android. You've read about Android. Now it is time to begin *Unlocking Android*.

Android is the software platform from Google and the Open Handset Alliance that has the potential to revolutionize the global cell phone market. This chapter introduces Android—what it is, and importantly, what it is not. After reading this chapter you will have an understanding of how Android is constructed, how it compares with other offerings in the market and its foundational technologies, plus you'll get a preview of Android application architecture. The chapter concludes with a simple Android application to get things started quickly.

This introductory chapter answers basic questions about what Android is and where it fits. While there are code examples in this chapter, they are not very in-depth—just enough to get a taste for Android application development and to convey the key concepts introduced. Aside from some context-setting discussion in the introductory chapter, this book is about understanding Android's capabilities and

will hopefully inspire you to join the effort to unlock the latent potential in the cell phone of the future.

1.1 *Introducing Android*

Android is the first open source mobile application platform that has the potential to make significant inroads in many markets. When 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.

Android is the product of primarily Google, but more appropriately the Open Handset Alliance. Open Handset Alliance is an alliance of approximately 30 organizations committed to bringing a “better” and “open” mobile phone to market. A quote taken from its website says it best: “Android was built from the ground up with the explicit goal to be the first open, complete, and free platform created specifically for mobile devices.” As discussed in this section, open is good, complete is good; “free” may turn out to be an ambitious goal. There are many examples of “free” in the computing market that are free from licensing, but there is a cost of ownership when taking support and hardware costs into account. And of course, “free” cell phones come tethered to two-year contracts, plus tax. No matter the way some of the details play out, the introduction of Android is a market-moving event, and Android is likely to prove an important player in the mobile software landscape.

With this background of who is behind Android and the basic ambition of the Open Handset Alliance, it is time to understand the platform itself and how it fits in the mobile marketplace.

1.1.1 *The Android platform*

Android is a software environment built for mobile devices. It is *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! While 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, all of the code examples in this book are written in Java using the Android SDK.

One feature of the Android platform is that there is no difference between the built-in applications and applications created with the SDK. This means that powerful applications can be written to tap into the resources available on the device. Figure 1.1 demonstrates the relationship between Android and the hardware it runs on. The most notable feature of Android may be that it is an open source platform; missing elements can and will be provided by the global developer community. Android’s Linux kernel-based OS does not come with a sophisticated shell environment, but because the platform is open, shells can be written and installed on a device. Likewise,

multimedia codecs can be supplied by third-party developers and do not need to rely on Google or anyone else to provide new functionality. That is the power of an open source platform brought to the mobile market.

The mobile market is a rapidly changing landscape with many players with diverging goals. Consider the often-at-odds relationship among mobile operators, mobile device manufacturers, and software vendors. Mobile operators want to lock down their networks, controlling and metering traffic. Device manufacturers want to differentiate themselves with features, reliability, and price points. Software vendors want unfettered access to the metal to deliver cutting-edge applications. Layer onto that a demanding user base, both consumer and corporate, that has become addicted to the “free phone” and operators who reward churn but not customer loyalty. The mobile market becomes not only a confusing array of choices but also a dangerous fiscal exercise for the participants, such as the cell phone retailer who sees the underbelly of the industry and just wants to stay alive in an endless sea of change. What users come to expect on a mobile phone has evolved rapidly. Figure 1.2 provides a glimpse of the way we view mobile technology and how it has matured in a few short years.

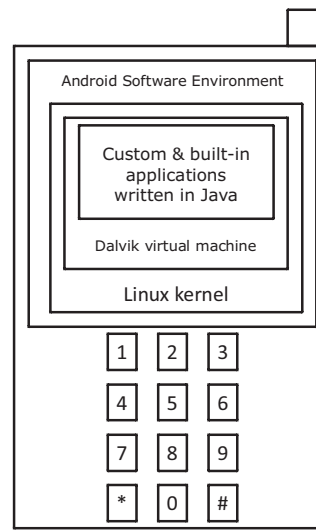


Figure 1.1 Android is software only. Leveraging its Linux kernel to interface with the hardware, you can expect Android to run on many different devices from multiple cell phone manufacturers. Applications are written in Java.

Platform vs. device

Throughout the book, wherever code must be tested or exercised on a device, a software-based emulator is employed. 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 of the binaries, code libraries, and tool chains. This book is focused on the Android platform. The Android emulators available in the SDK are simply one of many 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. If anyone has the clout to move the mobile market, it is Google and its entrant into the mobile marketplace, Android.

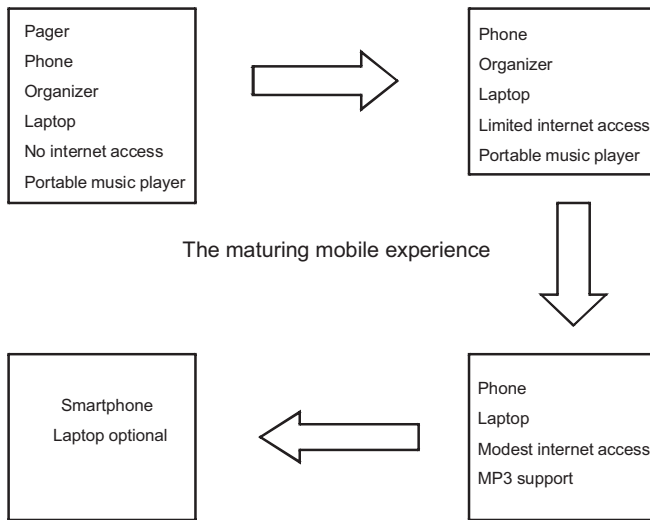


Figure 1.2 The mobile worker can be pleased with the reduction in the number of devices that need to be toted. Mobile device functionality has converged at a very rapid pace. The laptop computer is becoming an optional piece of travel equipment.

The next section begins and ends 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 and exploiting the platform itself!

1.1.2 In the market for an Android?

Android promises to have something for everyone. Android looks to support a variety of hardware devices, not just high-end ones typically associated with expensive “smartphones.” Of course, Android will run better on a more powerful device, particularly considering it is sporting a comprehensive set of computing features. The real question is how well Android can scale up and down to a variety of markets and gain market and mind share. This section provides conjecture on Android from the perspective of a few existing players in the marketplace. When talking about the cellular market, the place to start is at the top, with the carriers, or as they are sometimes referred to, mobile operators.

MOBILE OPERATORS

Mobile operators are in the business, first and foremost, of selling subscriptions to their services. Shareholders want a return on their investment, and it is hard to imagine an industry where there is a larger investment than in a network that spans such broad geographic territory. To the mobile operator, cell phones are—at the same time—a conduit for services, a drug to entice subscribers, and an annoyance to support and lock down.

The optimistic view of the mobile operator’s response to Android is that it is embraced with open arms 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.

The pessimistic view of the mobile operator's response to Android is that the operator feels threatened by Google and the potential of "free wireless," driven by advertising revenues and an upheaval of the market. Another challenge with mobile operators is that they want the final say on what services are enabled across their network. Historically, one of the complaints of handset manufacturers is that their devices are handicapped and not exercising all of the features designed into them because of the mobile operator's lack of capability or lack of willingness to support those features. An encouraging sign is that there are mobile operators involved in the Open Handset Alliance.

Enough conjecture; let's move on to a comparison of Android and existing cell phones on the market today.

ANDROID VS. THE FEATURE PHONES

The overwhelming majority of cell phones on the market are the consumer flip phones and feature phones. These are the phones consumers get when they walk into the retailer and ask what can be had for "free"; these are the "I just want a phone" customers. Their primary interest is a phone for voice communications and perhaps an address book. They might even want a camera. Many of these phones have additional capabilities such as mobile web browsing, but because of a relatively poor user experience, these features are not 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, or as it is typically known, 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. Part of this is due to screen size, but equally challenging is the browser technology itself, which often struggles to match the rich web experience of the desktop computer. Android features the market-leading WebKit browser engine, which brings desktop compatible browsing to the mobile arena. Figure 1.3 demonstrates the WebKit in action on Android. If this can be effectively scaled down to the feature phones, it would go a long way toward penetrating this end of the market.



Figure 1.3 Android's built-in browser technology is based on Webkit's browser engine.

NOTE The WebKit (<http://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, the browser found on the iPhone. It is not a stretch to say that the browser experience is what makes the iPhone popular, so its inclusion in Android is a strong plus for Android's architecture.

Software at this 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 to the software developer desiring to gain access to this market is that the bar to get an application on this platform is very 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 are recurring monthly. Just about everything else is a challenge to the software developer, however. Android's open application environment is more accessible than BREW.
- *J2ME, or Java Micro Edition*, is a very popular platform for this class of device. The barrier to entry is much lower for software developers. J2ME developers will find a "same but different" environment in Android. Android is not strictly a J2ME-compatible platform; however, the Java programming environment found in Android is a plus for J2ME developers. Also, as Android matures, it is very likely that J2ME support will be added in some fashion.

Gaming, a better browser, and anything to do with texting or social applications present fertile territory for Android at this end of the market.

While the masses carry the feature phones described in this section, Android's capabilities will put Android-capable devices into the next market segment with the higher-end devices, as discussed next.

ANDROID VS. THE SMARTPHONES

The market leaders in the smartphone race are Windows Mobile/SmartPhone and BlackBerry, with Symbian (huge in non-U.S. markets), iPhone, and Palm rounding out the market. While we could focus on market share and pros versus cons of each of the smartphone platforms, one of the major concerns of this market is a platform's ability to 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 or a jog dial.

Android's opportunity in this market is that it promises to deliver more performance on the same hardware and at a lower software acquisition cost. The challenge Android faces is the same challenge faced by Palm—scaling the Enterprise walls. 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. Finally, the iPhone has enjoyed unprecedented

success as an intuitive yet capable consumer device with a tremendous wealth of available software applications.

The next section poses an interesting question: can Android, the open source mobile platform, succeed as an open source project?

ANDROID VS. ITSELF

Perhaps the biggest challenge of all is Android's commitment to open source. Coming from the lineage of Google, Android will likely always be an open source project, but in order to succeed in the mobile market, it must sell millions of units. Android is not the first open source phone, but it is the first from a player with the market-moving weight of Google leading the charge.

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 push the ball up the hill and deliver desirable features is a force to be reckoned with, particularly in comparison with a traditional, commercial approach to software development. This is a trite topic unto itself by now, because the benefits of open source development are well documented. The other side of the open source equation is that, without a centralized code base that has some stability, Android could splinter and not gain the critical mass it needs to penetrate the mobile market. Look at the Linux platform as an alternative to the "incumbent" Windows OS. As a kernel, Linux has enjoyed tremendous success: it is found in many operating systems, appliances such as routers and switches, and a host of embedded and mobile platforms such as Android. Numerous Linux distributions are available for the desktop, and ironically, the plethora of choices has held it back as a desktop alternative to Windows. Linux is arguably the most successful open source project; as a desktop alternative to Windows, it has become splintered and that has hampered its market penetration from a product perspective. As an example of the diluted Linux market, here is an abridged list of Linux distributions:

- Ubuntu
- openSUSE
- Fedora (Red Hat)
- Debian
- Mandriva (formerly Mandrake)
- PCLinuxOS
- MEPIS
- Slackware
- Gentoo
- Knoppix

The list contains a sampling of the most popular Linux desktop software distributions. How many people do you know who use Linux as their primary desktop OS, and if so, do they all use the same version? Open source alone is not enough; Android must stay focused as a product and not get diluted in order to penetrate the market in a meaningful way. This is the classic challenge of the intersection between commercialization

and open source. This is Android's challenge, among others, because Android needs to demonstrate staying power and the ability scale from the mobile operator to the software vendor, and even at the grass-roots level to the retailer. Becoming diluted into many distributions is not a recipe for success for such a consumer product as a cell phone.

The licensing model of open source projects can be sticky. Some software licenses are more restrictive than others. 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.

1.1.3 Licensing Android

Android is released under two different open source licenses. The Linux kernel is released under the GPL (GNU General Public License), 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). While 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 may find fault with anything but complete openness, source code sharing, and noncommercialization; the ASL attempts to balance the open source goals with commercial market forces. If there is not a financial incentive to deliver Android-capable devices to the market, devices will never appear in the meaningful volumes required to adequately launch Android.

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 AppStore contains software titles for the iPhone. However, Apple's somewhat draconian grip on the iPhone software market requires that all applications be sold through its venue. This results in a challenging environment for software developers who might prefer to make their application available through multiple channels.

Contrast Apple's approach to application distribution with the freedom an Android developer enjoys to ship applications via traditional venues such as freeware and shareware and commercially through various marketplaces, including a developer's very own website! For software publishers desiring the focus of an on-device shopping experience, Google has launched the Android Market. For software developers who already have titles for other platforms such as Windows Mobile, Palm, or BlackBerry, traditional software markets such as Handango (<http://www.Handango.com>) also support selling Android applications. This is important because consumers new to Android will likely visit sites like Handango because that may be where they first purchased one of their favorite applications for their prior device.

The high-level, touchy-feely 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. The next section begins a high-level breakdown of the components of the Android stack.

1.2 Stacking up 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 of the major components of a computing platform are here and read like a Who's Who of the open source community. Here is a quick run-down of some of the prominent components of the Android stack:

- A Linux kernel provides a foundational hardware abstraction layer as well as core services such as process, memory, and file-system management. The kernel is where hardware-specific drivers are implemented—capabilities such as Wi-Fi and Bluetooth are found here. The Android stack is designed to be flexible, with many optional components which largely rely on the availability of specific hardware on a given device. These include features like touch screens, cameras, GPS receivers, and accelerometers.
- Prominent code libraries include:
 - Browser technology from WebKit—the same open source engine powering Mac's Safari and the iPhone's Mobile Safari browser
 - Database support via SQLite an easy-to-use SQL database
 - Advanced graphics support, including 2D, 3D, animation from SGL, and OpenGL ES
 - Audio and video media support from Packet Video's OpenCore
 - SSL capabilities from the Apache project
- An array of managers providing services for:
 - Activities and views
 - Telephony
 - Windows
 - Resources
 - Location-based services
- The Android runtime provides:
 - Core Java packages for a nearly full-featured Java programming environment. Note that this is *not* a J2ME environment.
 - The Dalvik virtual machine 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 built in this book) run in the Dalvik virtual machine, atop the components just introduced. The relationship among these layers can be seen in figure 1.4.

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

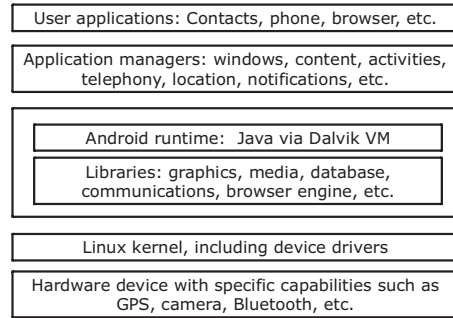


Figure 1.4 The Android stack offers an impressive array of technologies and capabilities.

Now that the obligatory stack diagram is shown and the layers introduced, let's look further at the runtime technology that underpins Android.

1.2.1 *Probing Android's foundation*

Android is built on a Linux kernel and an advanced, optimized virtual machine 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 very accessible to programmers because of both the number of Java software developers and the rich environment that Java programming has to offer. Mobile platforms that have relied on less-accessible programming environments have seen stunted adoption because of a lack of applications as developers have shied away from the platform.

BUILDING ON THE LINUX KERNEL

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. This 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, letting 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 is not 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.

User applications, as well as core Android applications, are written in the Java programming language and are compiled into *byte codes*. Byte codes are interpreted at runtime by an interpreter known as a *virtual machine*.

RUNNING IN THE DALVIK VIRTUAL MACHINE

The Dalvik virtual machine is an example of the needs of efficiency, the desire for a rich programming environment, and even some intellectual property constraints colliding, with innovation as a result. Android's Java environment provides a rich application platform and is very accessible because of the popularity of the Java language itself. Also, application performance, particularly in a low-memory setting such as is found in a mobile phone, is paramount for the mobile market. However this is not the only issue at hand.

Android is not a J2ME platform. Without commenting on whether this is ultimately good or bad for Android, there are other forces at play here. There is a matter of Java virtual machine licensing from Sun Microsystems. From a very high level, Android's code environment is Java. Applications are written in Java, which is compiled to Java bytecodes and subsequently translated to a similar but different representation called *dex files*. These files are logically equivalent to Java bytecodes, but they permit Android to run its applications in its own virtual machine that is both (arguably) free from Sun's licensing clutches and an open platform upon which Google, and potentially the open source community, can improve as necessary.

NOTE It is too early to tell whether there will be a big battle between the Open Handset Alliance and Sun over the use of Java in Android. From the mobile application developer's perspective, Android is a Java environment; however, the runtime is not strictly a Java virtual machine. This accounts for the incompatibilities between Android and "proper" Java environments and libraries.

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

After this discussion of the foundational technologies in Android, it is 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 are not comfortable or ready to begin coding, you might want to jump to chapter 2, where we introduce the development environment step by step.

1.3 **Booting Android development**

This section jumps right into the fray of Android development to focus on an important component of the Android platform, then expands 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. This may 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 discussed next, but they also represent the most important aspect of Android coding. *Understand the Intent, understand Android.*

NOTE Instructions for setting up the Eclipse development environment are found in appendix A. This environment is used for all 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. Classes are referenced and introduced without necessarily naming specific Java packages. Subsequent chapters take a more rigorous approach to introducing Android-specific packages and classes.

The next section provides foundational information about why Intents are important, then describes 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 application.

1.3.1 **Android’s good Intentions**

The power of Android’s application framework lies in the way in which it brings a web mindset to mobile applications. This doesn’t mean the platform has 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 itself works and how the user of the platform interacts with the mobile device. The power of the internet, should one be so bold to reduce it to a single statement, is that everything is just a click away. Those clicks are known to the user as Uniform Resource Locators (URLs), or alternatively, Uniform Resource Identifiers (URIs). The use of 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 a nontechnical but crucial response: *the way in which 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 hard-core power users. Deep menus, 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. While a consumer may purchase a device based on cool features enumerated in the marketing materials, instruction manuals are almost never touched. The ease of use of the UI of a computing

environment 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. This section introduces the concept of Intents and IntentFilters, Android's innovative navigation and triggering mechanism. Intents and IntentFilters bring the "click on it" paradigm to the core of mobile application use (and development!) for the Android platform.

- An Intent is a declaration of need.
- An IntentFilter is a declaration of capability and interest in offering assistance to those in need.
- An Intent is made up of a number of pieces of information describing the desired action or service. This section examines the requested action and, generically, the data that accompanies the requested action.
- An IntentFilter may 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. Application developers can create new actions as well. To view a piece of information, an application would employ 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 URI examples.

Table 1.1 Intents employ URIs, and some of the commonly employed URIs in Android are listed here.

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/

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*. A 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 the next section) and dispatches the Intent to the most appropriate recipient. Figure 1.5 depicts this relationship among Intents, IntentFilters, and BroadcastReceivers.

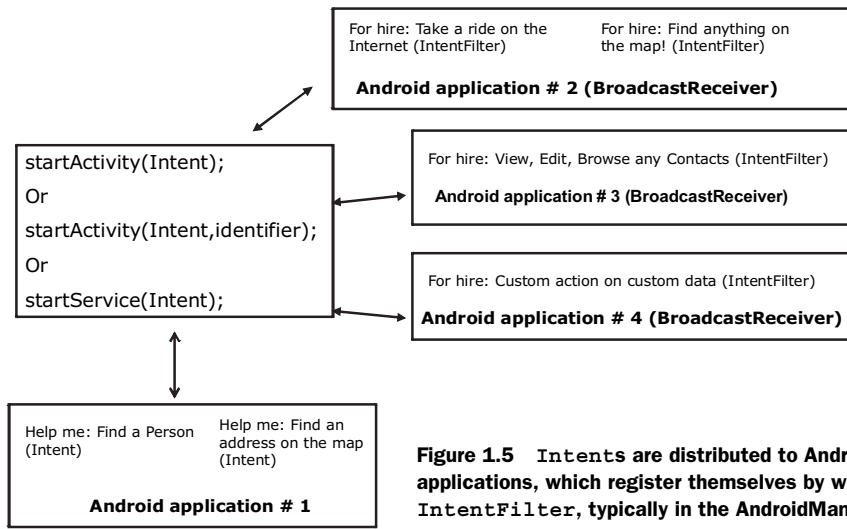


Figure 1.5 Intents are distributed to Android applications, which register themselves by way of the `IntentFilter`, typically in the `AndroidManifest.xml` file.

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

A common task on a mobile device is the lookup of a specific contact record for the purpose of initiating a call, sending an SMS (Short Message Service), or looking up a snail-mail address when you are standing in line at the neighborhood pack-and-ship store. A user may desire to view a specific piece of information, say a contact record for user 1234. In this case, the action is `ACTION_VIEW` and the data is a specific contact record identifier. This is accomplished by creating an `Intent` with the action set to `ACTION_VIEW` and a URI that represents the specific person of interest.

Here is an example of the URI for use with the `android.content.Intent.ACTION_VIEW` action:

```
content://contacts/people/1234
```

Here is an example of the URI for obtaining a list of all contacts, the more generalized URI of

```
content://contacts/people
```

Here is a snippet of code demonstrating the PICKing of a contact record:

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

This `Intent` is evaluated and passed to the most appropriate handler. In this case, the recipient would likely be a built-in `Activity` named `com.google.android.phone.Dialer`. However, the best recipient of this `Intent` may 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 requesting 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 it leads to the same UIs being 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 have 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. There are also *explicit* Intents, where we can specify the exact class we desire to handle the Intent. This is helpful when we know exactly which Activity we want to handle the Intent and do not want to leave anything up 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, as shown here:

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

These examples show how an Android application 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 already created on the system, thereby publishing new functionality for the platform. This facet alone should bring joy to independent software vendors (ISVs) who have made a living by offering better contact manager 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, so specific Intent-handling features can be added to a device, which may 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 is not hard to imagine that an absolutely unique user experience is possible with Android because of the variety of Activities with specific IntentFilters installed on any given device. It is architecturally feasible to upgrade various aspects of an Android installation to provide sophisticated functionality and customization. While this may be a desirable characteristic for the user, it can be a bit troublesome for someone providing tech support and having to navigate a number of components and applications to troubleshoot a problem.

Because of this potential for added complexity, this approach of ad hoc system patching to upgrade specific functionality should be entertained cautiously and with one's 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. There are also Intents that are more event driven than task-oriented, as the earlier contact record example described. For example, the Intent class is also used to notify applications that a text message has arrived. Intents are a very central element to Android and will be revisited on more than one occasion.

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

1.3.2 **Activating Android**

This section builds on the knowledge of the Intent and IntentFilter classes introduced in the previous section and explores the four primary components of Android applications as well as their relation to the Android process model. Code snippets are included to provide a taste of Android application development. More in-depth examples and discussion are left for later chapters.

NOTE A particular Android application may not contain all of these elements, but it will have at least one of these elements and could in fact have all of them.

ACTIVITY

An application may or may not have a UI. If it has a UI, it will have at least one Activity.

The easiest way to think of an Android Activity is to relate a visible screen to an Activity, as more often than not there is a one-to-one relationship between an Activity and a UI screen. An Android application will 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 listing 1.1.

Listing 1.1 A very basic Activity in an Android application

```
package com.msi.manning.chapter1;

import android.app.Activity;  ← ❶ Activity class import
import android.os.Bundle;

public class activity1 extends Activity {  ← ❷ Activity class extension
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.main);  ← ❸ Set up the UI
    }
}
```

The Activity class ❶ 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 ❷ extends the class Activity. For more examples of using an Activity, please see chapter 3. One of the primary tasks an Activity performs is the display of

UI elements, which are implemented as Views and described in XML layout files **3**. Chapter 3 goes into more detail on Views and Resources.

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

You say Intent; I say Intent

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

There are Intents used to assist in navigation 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 used with the `startActivity` or `startActivityForResult` methods.

Services can be started by passing an Intent to the `startService` method.

BroadcastReceivers receive Intents when responding to systemwide events such as the phone ringing or an incoming text message.

The Activity represents a very visible application component within Android. With assistance from the View class covered in chapter 3, the Activity is the most common type of Android application. The next topic of interest is the Service, which runs in the background and does not generally present a direct UI.

SERVICE

If an application is to have a long lifecycle, it should be put into a Service. For example, a background data synchronization utility running continuously should be implemented as a Service.

Like the Activity, a Service is a class provided in the Android runtime that should be extended, as seen in listing 1.2, which sends a message to the Android log periodically.

Listing 1.2 A simple example of an Android Service

```
package com.msi.manning.chapter1;

import android.app.Service;    ← 1 Service import
import android.os.IBinder;
import android.util.Log;      ← 2 Log import

public class service1 extends Service implements Runnable { ← 3 Extending the
    public static final String tag = "service1";
    private int counter = 0;
    @Override
    protected void onCreate() { ← 4 Initialization in the
        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) { ← ❸ Binding to the Service
    return null;
}
}

```

This example requires that the package `android.app.Service` ❶ be imported. This package contains the `Service` class. This example also demonstrates Android's logging mechanism ❷, which is useful for debugging purposes. Many of the examples in the book include using the logging facility. Logging is discussed in chapter 2. The `service1` class ❸ extends the `Service` class. This class also implements the `Runnable` interface to perform its main task on a separate thread. The `onCreate` ❹ method of the `Service` class permits the application to perform initialization-type tasks. The `onBind()` method ❺ is discussed in further detail in chapter 4 when the topic of inter-process communication in general is explored.

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 long-running task in a `Service`, it is time to explore the `BroadcastReceiver`, another form of Android application that is dedicated to processing `Intents`.

BROADCASTRECEIVER

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

- The application may 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, it does not have to be running in order to be triggered. When the event occurs, the application is started automatically upon notification of the triggering event. All of this housekeeping is managed by the Android OS itself.
- An application may register at runtime via the `Context` class's `registerReceiver` method.

Like Services, BroadcastReceivers do not have a UI. Of even more importance, 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 is recommended that the code initiate a request to a `Service` to complete the requested functionality.

NOTE The familiar `Intent` class is used in the triggering of `BroadcastReceivers`; the use of these `Intents` is mutually exclusive from the `Intents` used to start an `Activity` or a `Service`, as previously discussed.

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.

Listing 1.3 is an example of a `BroadcastReceiver` triggering upon an incoming text message.

Listing 1.3 A sample `IntentReceiver`

```
package com.msi.manning.unlockingandroid;

import android.content.Context;
import android.content.Intent;
import android.content.IntentReceiver;
import android.util.Log;

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!");
        }
    }
}
```

Looking at listing 1.3 we find a few items to discuss. The class `MySMSMailBox` extends the `BroadcastReceiver` class ❶. This subclass approach is the most straightforward way to employ a `BroadcastReceiver`. Note the class name `MySMSMailBox`, as it will be used in the `AndroidManifest.xml` file, shown in listing 1.4. The tag variable ❷ 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 filtering and organizing log messages in the console. Chapter 2 discusses the log mechanism in further detail. The `onReceive` method ❸ is where all of the work takes place in a `BroadcastReceiver`—this method must be implemented. Note that a given `BroadcastReceiver` can register multiple `IntentFilters` and can therefore be instantiated for an arbitrary number of `Intents`.

It is important to make sure to handle the appropriate `Intent` by checking the action of the incoming `Intent`, as shown in ❹. Once the desired `Intent` is received,

carry out the specific functionality required. A common task in an SMS-receiving application would be to parse the message and display it to the user via a Notification Manager display. In this snippet, we simply record the action to the log ⑤.

In order for this `BroadcastReceiver` to fire and receive this `Intent`, it must be listed in the `AndroidManifest.xml` file, as shown in listing 1.4. This listing contains the elements required 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=".chapter1" 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 the "."**

③ **IntentFilter definition**

Certain tasks within the Android platform require the application to have a designated privilege. To give an application the required permissions, the `<uses-permission>` tag is used ①. This is discussed in detail later in this chapter in the `AndroidManifest.xml` section. 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. The dot is required. If your application is not behaving as expected, one of the first places to check is your `Android.xml` file, and look for the dot! The `IntentFilter` is defined in the `<intent-filter>` tag. The desired action in this example is `android.provider.Telephony.SMS_RECEIVED` ③. The Android SDK enumerates the available actions for the standard `Intents`. In addition, remember that user applications can define their own `Intents` as well as listen for them.

Now that we have 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.

CONTENT PROVIDER

If an application manages data and needs to expose that data to other applications running in the Android environment, a `ContentProvider` should be implemented. Alternatively, if an application component (`Activity`, `Service`, or `BroadcastReceiver`) needs to access data from another application, the other application's

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 and out of network coverage and placing phone calls. This section's example demonstrated another feature of the emulator, the receipt of an SMS message.

To send an SMS message to the emulator, telnet to port 5554 (the port # may 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` from the prompt.

These tools are discussed in more detail in chapter 2.

`ContentProvider` is used. The `ContentProvider` implements a standard set of methods to permit an application to access a data store. The access may be for read and/or write operations. A `ContentProvider` may provide data to an `Activity` or `Service` in the same containing application as well as an `Activity` or `Service` contained in other applications.

A `ContentProvider` may 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 is not required. In essence, the `ContentProvider` is a data layer providing data abstraction for its clients and centralizing storage and retrieval routines in a single place.

Directly sharing files or databases is discouraged on the Android platform and is further enforced by the Linux security system, which prevents ad hoc file access from one application space to another without explicitly granted permissions.

Data stored in a `ContentProvider` may be of traditional data types such as integers and strings. Content providers can also manage binary data such as image data. When binary data is retrieved, suggested practice is to return a string representing the filename containing the binary data. In the event a filename is returned as part of a `ContentProvider` query, the file should not be accessed directly, but rather you should use the helper class, `ContentResolver`'s `openInputStream` method, to access the binary data. This approach negates Linux process/security hurdles as well as keeps all data access normalized through the `ContentProvider`. Figure 1.6 outlines the relationship among `ContentProviders`, data stores, and their clients.

A `ContentProvider`'s data is accessed through the familiar Content URI. A `ContentProvider` defines this 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");
```

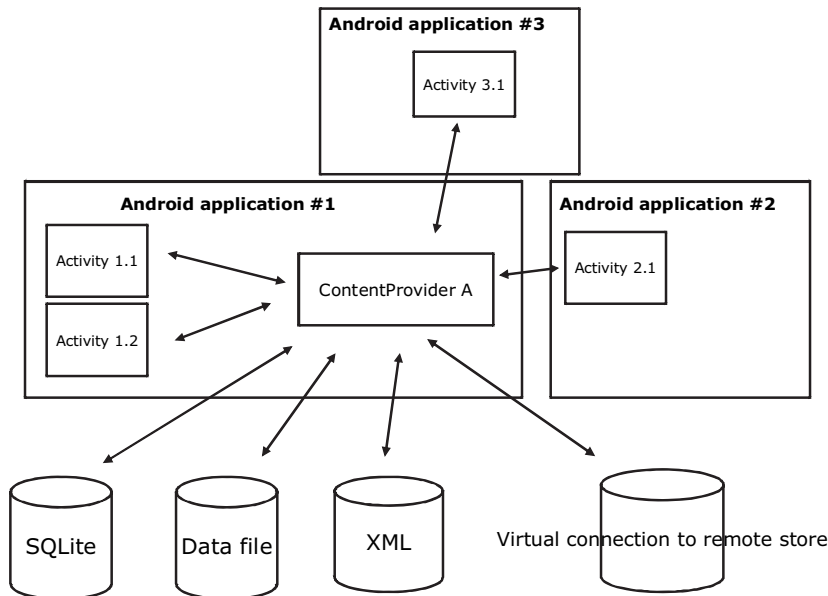


Figure 1.6 The content provider is the data tier for Android applications and is the prescribed manner in which data is accessed and shared on the device.

From this point, accessing a `ContentProvider` is similar to using Structured Query Language (SQL) in other platforms, though a complete SQL statement is not employed. A query is submitted to the `ContentProvider`, including the columns desired and optional `Where` and `Order By` clauses. For those familiar with parameterized queries in SQL, parameter substitution is even supported. Results are returned in the `Cursor` class, of course. A detailed `ContentProvider` example is provided in chapter 5.

NOTE In many ways, a `ContentProvider` acts like a database server. While 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`, and/or `BroadcastReceiver`.

This concludes the 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 it was copied and not installed properly. Every software platform has environmental concerns, though they vary by platform. For example, when connecting to a remote resource such as a database server or FTP server, which username and password should you use? What about the necessary libraries to run your application? These are all topics related to software deployment. Each Android application requires a file named

AndroidManifest.xml, which ties together the necessary pieces to run an Android application on a device.

1.3.3 AndroidManifest.xml

The previous sections introduced the common elements of an Android application. To restate: an Android application will contain at least one Activity, Service, BroadcastReceiver, or ContentProvider. Some of these elements will advertise the Intents they are interested in processing via the IntentFilter mechanism. All of these pieces of information need to be tied together in order 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 of the design-time relationships of a specific application and Intents. AndroidManifest.xml files act as deployment descriptors for Android applications. Listing 1.5 is an example of a very simple AndroidManifest.xml file.

Listing 1.5 AndroidManifest.xml file for a very 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" <— ① Package name
    <application android:icon="@drawable/icon">
        <activity android:name=".chapter1" 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>
```

IntentFilter definition ③

Application name ②

Looking at this simple AndroidManifest.xml, we see that the manifest element contains the obligatory namespace as well as the Java package name ① containing this application. This application contains a single Activity, with a class name of chapter1 ②. Note also the @string syntax. Anytime an @ symbol is used in an AndroidManifest.xml file, it is referencing information stored in one of the resource files. In this case, the label attribute is obtained from the app_name string resource defined elsewhere in the application. Resources are discussed in further detail later in chapter 3. This application’s lone Activity contains a single IntentFilter definition ③. 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 1.7. It is possible to have multiple Activity elements in a manifest file (and thereby an application), with more than one of them visible in the launcher window.

In addition to the elements used in this sample manifest file, other common tags include:

- The `<service>` tag represents a Service. The attributes of the service tag include its class and label. A Service may also include the `<intent-filter>` tag.
- The `<receiver>` tag represents a BroadcastReceiver, which may or may not 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 revisit the AndroidManifest.xml file a number of times throughout the book because we need to add more detail for certain elements.

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 it actually executes. The next section discusses the relationship between an Android application and its Linux and Dalvik virtual machine runtime.

1.3.4 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 virtual machine. The OS may need to unload, or even kill, an application from time to time to accommodate resource allocation demands. There is a hierarchy or sequence the system uses to select the victim of a resource shortage. In general, the rules are as follows:

- Visible, running activities have top priority.
- Visible, nonrunning activities are important, because they are recently paused and are likely to be resumed shortly.
- A running service is 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 Activities.

It's time to wrap up this chapter with a simple Android application.



Figure 1.7 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.

ps -a

The Linux environment is complete, including process management. It is possible to launch and kill applications directly from the shell on the Android platform. However, this is largely a developer's debugging task, not something the average Android handset user is likely to be carrying out. It is nice to have for troubleshooting application issues. It is unheard of on commercially available mobile phones to "touch the metal" in this fashion. For more in-depth exploration of the Linux foundations of Android, see chapter 13.

1.4 An Android application

This section presents a simple Android application demonstrating a single Activity, with one View. The Activity collects data, a street address to be specific, and creates an Intent to find this address. The Intent is ultimately dispatched to Google Maps. Figure 1.8 is a screen shot of the application running on the emulator. The name of the application is Where Do You Live.



Figure 1.8 This Android application demonstrates a simple Activity and Intent.

As previously introduced, the `AndroidManifest.xml` file contains the descriptors for the high-level classes of the application. This application contains a single Activity named `AWhereDoYouLive`. The application's `AndroidManifest.xml` file is shown in listing 1.6.

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>
</manifest>
```

The sole Activity is implemented in the file `AWhereDoYouLive.java`, presented in listing 1.7.

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 icle) {
        super.onCreate(icle);
        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) {
                    ...
                }
            }
        });
    }
}
```

Annotations in the diagram:

- 1 Set up GUI**: Points to `setContentView(R.layout.main);`
- 2 Reference Edit field**: Points to `findViewById(R.id.address);`
- 3 Reference button**: Points to `findViewById(R.id.launchmap);`
- 4 Get address**: Points to `addressfield.getText().toString();`
- 5 Prepare Intent**: Points to `Intent.ACTION_VIEW,`
- 6 Initiate lookup**: Points to `startActivity(geoIntent);`

In this example application, the `setContentView` method **1** creates the primary UI, which is a layout defined in `main.xml` in the `/res/layout` directory. The `EditText` view

collects information, which is in this case an address. The `EditText` view is a text box or edit box in generic programming parlance. The `findViewById` method ❷ 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 ❸. When this button is clicked, the application obtains the entered address by invoking the `getText` method of the associated `EditText` ❹.

Once the address has been retrieved from the UI, we 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, as seen in ❺.

Finally, the application asks Android to perform the `Intent`, which ultimately results in the mapping application displaying the chosen address. This is accomplished with a call to the `startActivity` method ❻.

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. Note that you should never modify the `R.java` file manually, as it is automatically built every time the underlying resources change.

Listing 1.8 `R.java` contains 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;
    }
}

```

Android resources are covered in greater depth in chapter 3.

The primary screen of this application is defined as a `LinearLayout` view, as shown in listing 1.9. It is a single layout containing one label, one text entry element, and one button control.

Listing 1.9 Main.xml defines the UI elements for our 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 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 previously, 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 for this application is shown in listing 1.10. The strings.xml 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>

```

This concludes our first Android application.

1.5 Summary

This chapter has introduced the Android platform and briefly touched on market positioning, including what Android is up against as a newcomer to the mobile marketplace. Android is such a new platform that there are sure to be changes and

announcements as it matures and more and varied hardware hits the market. New platforms need to be adopted and flexed to identify the strengths and expose the weaknesses so they can be improved. Perhaps the biggest challenge for Android is to navigate the world of the mobile operators and convince them that Android is good for business. Fortunately with Google behind it, Android should have some ability to flex its muscles, and we'll see significant inroads with device manufacturers and carriers alike.

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. While Android development is done in the Java programming language, the runtime is executed in the Dalvik virtual machine, as an alternative to the Java virtual machine from Sun. Regardless of the VM, Java coding skills are an important aspect of Android development. The bigger issue is the degree to which existing Java libraries can be leveraged.

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

The `AndroidManifest.xml` descriptor file ties all of the details together for an Android application. It includes all of 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 new elements are added and explained.

Finally, this chapter provided a taste of Android application development with a very simple example tying a simple UI, an Intent, and Google Maps into one seamless user experience. This is just scratching the surface of what Android can do. The next chapter takes a deeper look into the Android SDK to learn more about what is in the toolbox to assist in *Unlocking Android*.

Unlocking Android A Developer's Guide

Frank Ableson • Charlie Collins • Robi Sen • Foreword by Dick Wall



Mobile app developers don't have to accept vendor lock-in any more. Android is an open (and free) Java-based platform that provides the device OS, SDK, server components, and numerous helper applications you need to build efficient—and extremely cool—mobile apps.

Unlocking Android is a concise, hands-on developer's guide for the Android operating system and development tools. It starts by introducing Android basics as well as the architectural concepts you need. It then presents practical examples showing you how to build apps that use, extend, or replace Android's features, large and small. It's ideal for corporate developers and hobbyists alike who have an interest, or a mandate, to deliver high quality and cost-effective mobile phone software. This book requires previous experience with Java but no prior knowledge of Android.

What's Inside

- Thorough coverage of Android 1.x
- Uses Eclipse for Android development
- UI design, networking, notification, and more
- Many code examples
- Bonus chapter on hacking Android

About the Authors

Frank Ableson is an entrepreneur who helps leading mobile software companies bring their products to market. **Charlie Collins** is a Java developer with experience in mobile, embedded, and alternative languages on the JVM. **Robi Sen** focuses on developing novel wireless solutions.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/UnlockingAndroid

“Valuable, useful”

—From the Foreword by Dick Wall
Senior Engineer, Former Android
Advocate for Google, and
Java Posse Co-host

“For newbies and experts alike,
this book is like a lighthouse.”

—Kevin Galligan
CTO, Medical Research Forum

“Chock-full of valuable code
and tips.”

—Scott Webster
AndroidGuys Editor

“Take your app from zero
to running in no time flat.”

—Charles Hudson
President and Founder, Aduci

“Highly recommended!”

—Horaci Macias
Software Architect, Avaya



MANNING

\$39.99 / Can \$49.99 [INCLUDING eBook]

