

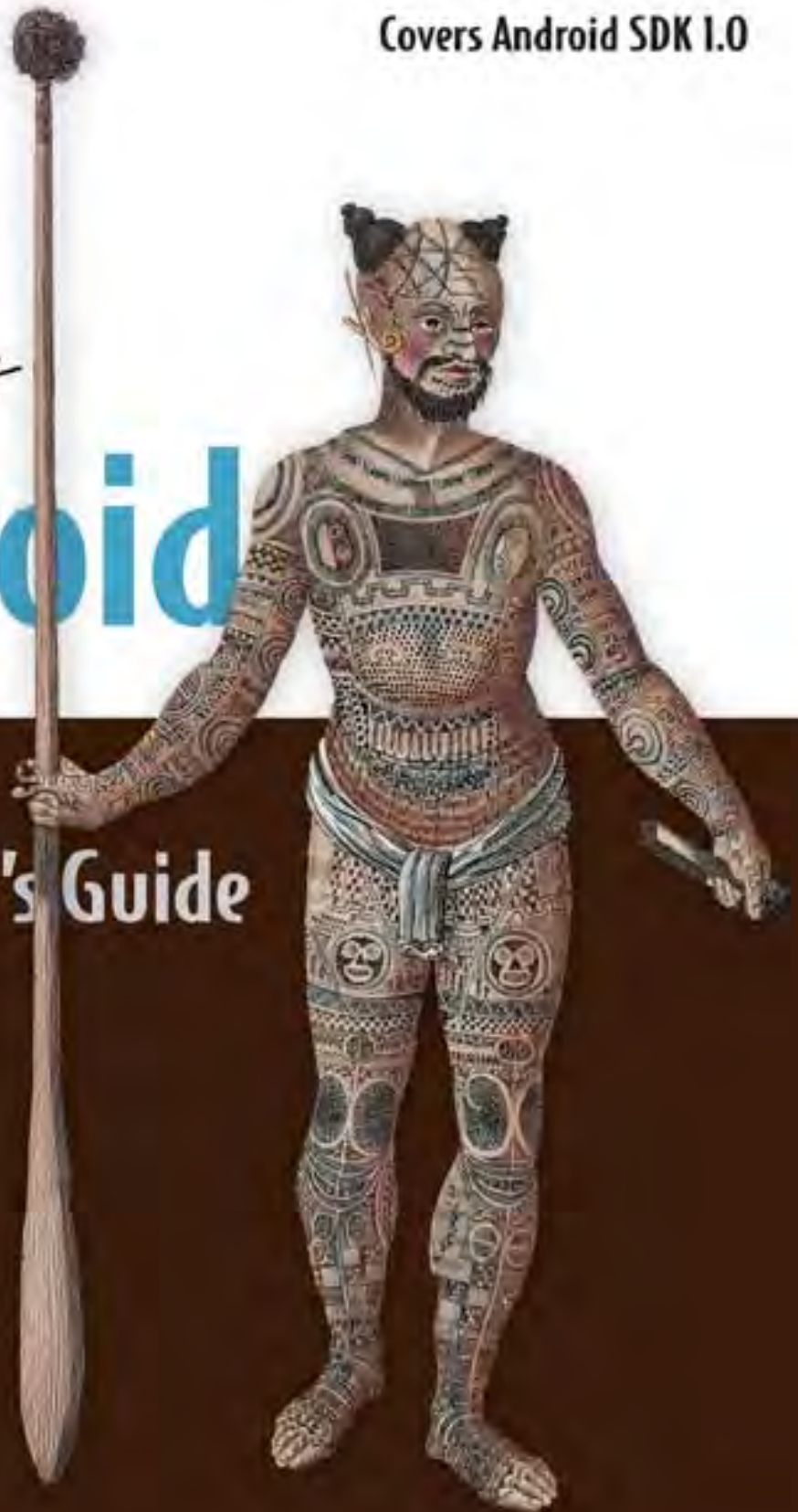
Covers Android SDK 1.0

unlocking **Android**

A Developer's Guide

W. Frank Ableson
Charlie Collins
Robi Sen

 MANNING





**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Contents

Part I: What is Android—The Big Picture

Chapter 1: Targeting Android

Chapter 2: Development environment

Part II: Learning Android's Key Technologies

Chapter 3: User Interfaces

Chapter 4: Intents and Services

Chapter 5: Storing and Retrieving Data

Chapter 6: Networking

Chapter 7: Telephony

Chapter 8: Notification and Alarms

Chapter 9: Graphics and Animation

Chapter 10: Multimedia

Chapter 11: Location Based Services

Part III: Android applications for the Real Device

Chapter 12: Putting it all together: A Field Service Application

Chapter 13: Hacking Android

Appendix: Installing the Android SDK

Targeting Android

In this chapter,

- Android the Open Source Mobile Platform
- Android Application Architecture
- A Sample Android Application

You've heard about Android. You've read about Android. Now it is time to Unlock 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, Android's foundational technologies, and a preview of Android application architecture. The chapter concludes with a simple Android application to get things started quickly.

This introductory chapter answers some of the 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 unlocking Android's capabilities and will hopefully inspire you to join the effort to unlock the latent potential in the cell phone of the very near 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 dimensions to consider, both technical and market related. This first section introduces the platform and provides some context to help 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. The Open Handset Alliance is an organization of approximately 30 organizations committed to bringing a “better” and “open” mobile phone to market. A quote taken from their 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, however “free” may be turn out to be an ambitious goal. There are many examples of “free” in the computing market that are free from licensing, but of course 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 a little more about 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 operating system, a rich User Interface, end-user applications, code libraries, application frameworks, multimedia support and much more. And, yes, even telephone functionality is included! While components of the underlying operating system are written in C or C++, user applications are built for Android using the Java programming language. Even the built-in applications are written in Java. With the exception of some Linux exploratory exercises in Chapter 14, all of the code examples in this book are written in Java using the Android SDK.

One of the most powerful features of the Android platform is that there is “no difference” between the built-in applications shipped on the device and user 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. Perhaps the most notable feature of Android is that it is an open source platform, therefore missing elements can and will be provided by the global developer community. For example, Android’s Linux kernel based operating system 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.

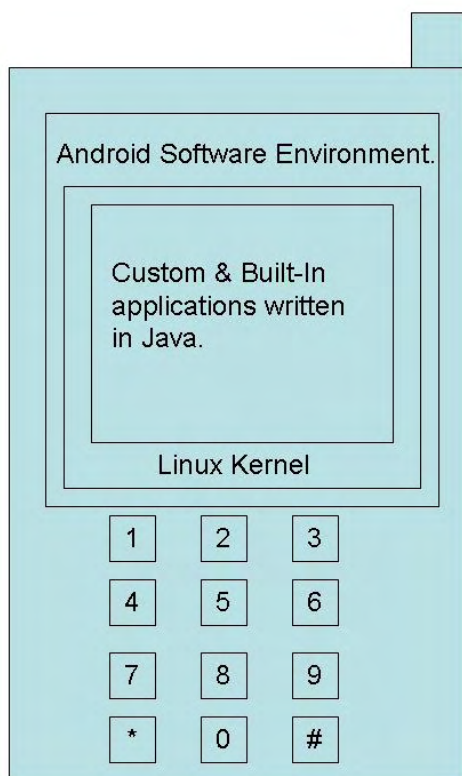


Figure 1.1. Android is software only. Leveraging its Linux kernel to interface with the hardware, Android can be expected to run on many different devices from multiple cell phone manufacturers. Applications are written in the Java programming language.

Platform vs. Device

Android capable devices are just hitting the market while this book was written though there are often references to the “device” in this book. The term *device* here represents the future family of mobile phones capable of running Android. Throughout the book, wherever code must be tested or exercised on a “device”, a software based emulator is employed.

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.

The mobile market is a rapidly changing landscape with many players who often have diverging goals. For example, consider the often at-odds relationship between mobile operators, mobile device manufacturers and software vendors. Mobile operators want to lock down their networks, controlling and of course 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. Then layer on to that a demanding user base, both consumer and corporate, that has become addicted to the “free phone”, and operators who reward churn and 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. For example, examine Figure 1.2 which provides a glimpse of the way we view “mobile technology” and how it has matured in a few short years.

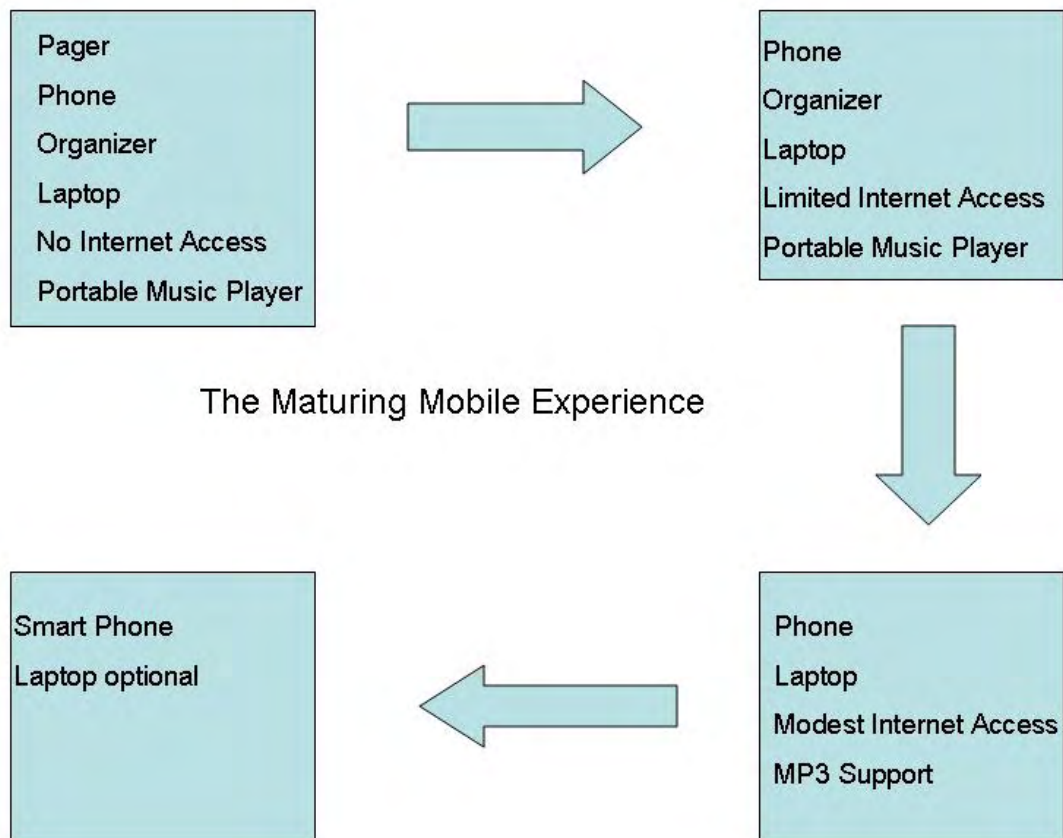


Figure 1.2: The mobile worker can be pleased with the reduction in devices that need to be toted. Mobile device functionality has converged at a very rapid pace. The laptop computer is becoming an optional piece of travel equipment.

With all of that as a back-drop, creating a successful mobile platform is clearly a non-trivial task involving numerous players, so Android is an ambitious undertaking, even for Google, a company of seemingly boundless resources and moxy. If anyone has the clout to move the mobile market, it is Google and its entrant to the mobile marketplace, Android.

The next section begins and ends the “Why and Where of Android” to provide some context and set the backdrop 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 devices typically associated with expensive “Smart Phones”. 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 can Android scale up and down to a variety of markets and gain market- and mind-share. This section provides some conjecture on Android from the perspective of a few existing “players” in the market

place. When talking about the cellular market, the place to start is “at the top”, with the carriers or as they are sometimes referred, Mobile Operators.

MOBILE OPERATORS

Mobile operators are in the business of selling subscriptions to their services, first and foremost. Share holders 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 all at once 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 Android 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, then 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 their market. The other threat from mobile operators is that they have 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 due to 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 for this book, let’s move on to a quick 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”, or the “I just want a phone” customer. This customer’s primary interest is in a phone for voice communications and perhaps an address book and maybe even a camera. Many of these phones have more capabilities such as mobile web browsing, but due to a relatively limited user experience, these features are not employed heavily. The one exception to that is text messaging which is a dominant application, no matter the classification of device. Another increasingly in-demand category is location based services.

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 towards penetrating this end of the market.



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

NOTE

The WebKit browser engine is an open source project which 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 is very high to get an application on this platform as 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 much 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 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 feature phones as described in this section, Android's capabilities will put Android-capable devices into the next market segment with the higher end devices as discussed in the next section.

ANDROID VS. THE SMART PHONES

The market leaders in the smart phone race are Windows Mobile, Windows SmartPhone, Blackberry, with Symbian (huge in non-US markets), iPhone and Palm rounding out the market. While we could focus on market share and pros vs. cons of each of the smart phone 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 respectable, largely due to 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 due to its intuitive email capabilities and the Microsoft platforms are compelling because of tight integration to the desktop experience and overall familiarity for Windows users.

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

ANDROID VS. ITSELF – THE CHALLENGE OF OPEN SOURCE IN THE CONSUMER MARKET

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 really a trite topic unto itself by now as the benefits of open source development are well documented. However, the other side of the open source equation is that without a centralized code base that has some stability to it, Android could splinter and not gain the critical mass it needs to penetrate the mobile market. For example, look at the Linux platform as an alternative to the "incumbent" Windows operating system. As a kernel, Linux has enjoyed tremendous success as it is found in many operating systems, appliances such as routers and switches, and a host of embedded and mobile platforms, such as Android. There are numerous "Linux distributions" available for the desktop – and ironically, the plethora of choice has held it back as a desktop alternative to Windows. Linux is arguably the most successful Open Source project, however 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, consider the abridged list of Linux distributions:

- Ubuntu
- openSUSE
- Fedora (Red Hat)

- Debian
- Mandriva (formerly Mandrake)
- PCLinuxOS
- MEPIS
- Slackware
- Gentoo
- Knoppix

The list contains just a sampling of the more popular Linux desktop software distributions. How many people do you know that 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, as 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, as is required for anyone licensing the open source operating system 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 to be more friendly towards commercial use. Some open source purists will find fault with anything but complete open-ness, source code sharing, and non-commercialization, however 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 launch Android.

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 comprise the environment, sometimes referred to as a "stack" because of the layer upon layer construction. The next section begins a high-level break down 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 WiFi and Bluetooth are found here. The Android stack is designed to be flexible, with many optional components. The optional components 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. It should be noted 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 3rd party applications (such as the ones built in this book) run in the Dalvik Virtual Machine, atop the components just introduced. The relationship between these layers can be seen in Figure 1.4.

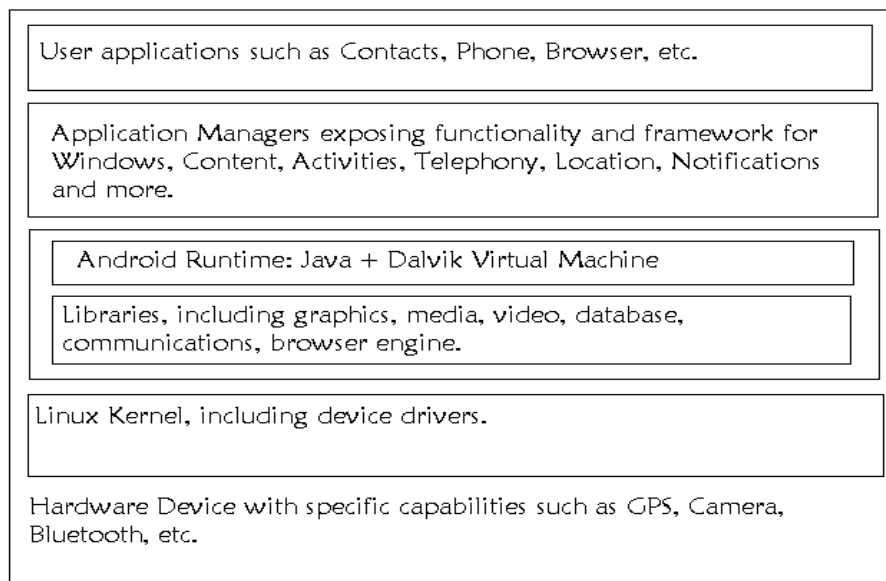


Figure 1.4: The Android Stack offers an impressive array of technologies and capabilities.

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 excellent Java books on the market. An excellent source of Java titles can be found at <http://mannning.com/catalog/java>.

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

1.2.1 Probing Android's Foundation

Android is built upon a Linux kernel and an advanced, optimized Virtual Machine for its Java applications. Both of these 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 because it makes the Android environment very accessible to programmers due to both the number of Java software developers and the rich environment that Java programming has to offer. Other mobile platforms that have relied on less accessible programming environments have seen stunted adoption due to a lack of applications as developers have shied away from the platform.

BUILDING ON THE LINUX KERNEL

Why 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 numerous talented individuals and companies to move the platform forward. This is particularly important in the world of mobile devices where the 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 also 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. Consumers and business customers want the cool data features and will purchase a device based on those features, but they demand voice reliability. 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 of a resource being unavailable, 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 perhaps even some Intellectual Property constraints colliding with creative innovation occurring as a result. Android's Java environment provides a rich application platform and is very accessible due to the popularity of the Java language. 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, which is compiled to Java byte codes and then subsequently translated to a similar but different representation called dex files. These files are logically equivalent to Java byte codes, but permit Android to run its applications in its own Virtual Machine that is both (arguably) free from Sun's licensing clutches and is an open platform upon which Google, and potentially the open source community, can improve as necessary. At the time of this writing the Dalvik Virtual Machine has not been released as an open source project.

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 thing to know about the Dalvik Virtual Machine is that Android applications run inside of it and that it relies on the Linux kernel for services such as process, memory and file system management.

After this discussion of the foundational technologies in Android, it is now time to focus on Android application development.

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 and then expands out 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 lookup 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 arguably the most important aspect of Android coding. *Understand the Intent, Understand Android*.

NOTE:

Instructions for setting up the Eclipse development environment may be found in the Appendix. This environment is used for all examples in this book. Chapter 2 goes into more detail on 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, and then goes on to describe how Intents work. Beyond the introduction of the Intent, the remainder of this chapter describes the major elements of Android application development leading up to and including the first complete application.

1.3.1 Android's good Intent-ions

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 simply has a powerful browser and is limited to clever Javascript and server-side resources, but rather it goes to the core of both 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 Locator, URLs, or alternatively, Uniform Resource Identifiers, or 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 to that question is a non-technical, but crucial response: *the way in which a mobile user navigates on the platform is crucial to its commercial success*. Platforms which 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 all of the cool features enumerated in the marketing materials, instruction manuals are almost never touched. The ease of use of the User Interface (UI) of a computing environment is highly correlated with its market penetration. User Interfaces 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. There are a number of built-in Intent actions 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 perhaps a reference to a media clip. Table 1.2 lists some example URI possibilities:

Table 1.2: Intents employ URIs, with some of the commonly employed URIs in Android listed here.

Type of information	URI data
Contact Lookup	<code>content://contacts/people</code>
Map Lookup/Search	<code>Geo:0,0?q=23+Route+206+Stanhope+NJ</code>
Browser Launch to a specific website	http://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/shell.

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 between Intents, IntentFilters and BroadcastReceivers.

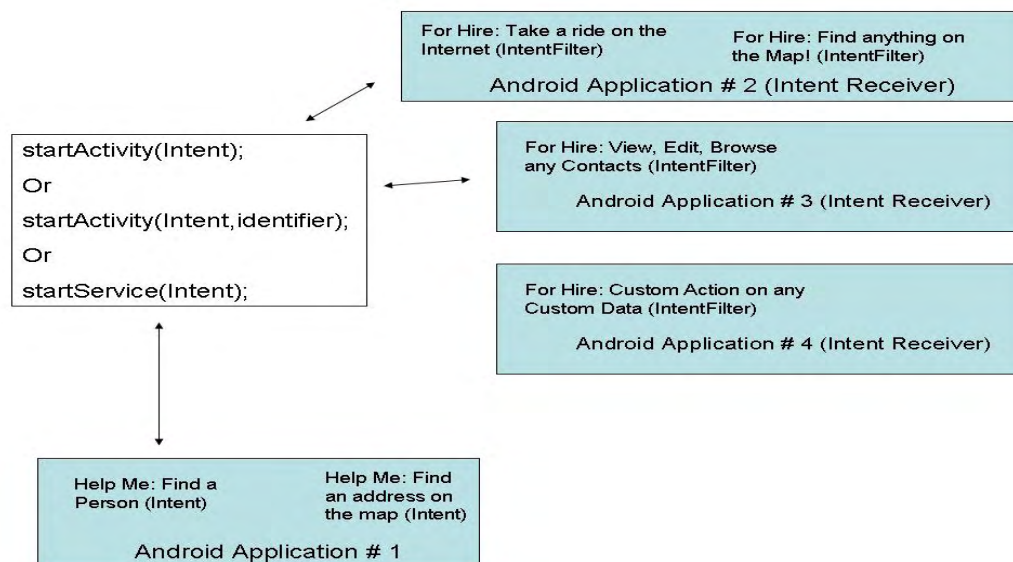


Figure 1.5: Intents are distributed to various Android applications which register themselves by way of the IntentFilter, typically in the AndroidManifest.xml file.

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, which is 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, or perhaps 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 a the action set to ACTION_VIEW and a URI which 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 (ie, the one you build!), a built-in application as in this case, or it may be a 3rd 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 user interfaces being used frequently, creating familiarity for the user. *This is particularly important for mobile platforms where the user is often non tech-savvy, nor interested in learning multiple ways to accomplish the same task such as looking up a contact on their 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 through a number of components and applications to troubleshoot a problem.

This discussion of Intents has focused on Intents which cause User Interface elements to be displayed for the user. There are also Intents which are more “event-driven” rather 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 in this book.

Now that Intents have been briefly introduced as the catalyst for navigation and event flow on Android, let’s jump back out to a broader view and discuss Android Application life cycle and the key components that make Android tick. The Intent will come into better focus as Android is further explored 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 User Interface. If it has a user interface, it will have one or more 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 user interface screen. An Android application will often contain more than one Activity. Each Activity displays a user interface and responds to system and user initiated events. The Activity employs one or more Views to present the actual User Interface elements to the user. The Activity class is extended by user classes, as seen in the code listing 1.1.

Listing 1.1: A very 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 icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.main);
    }
}
```

1. 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.
2. The class `activity1` extends the class `Activity`. For more examples of using an Activity, please see Chapter 3.
3. One of the primary tasks an Activity performs is the display of User Interface elements which are implemented as Views and described in XML layout files. 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 system-wide 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 introduced later in this book, 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 User Interface.

SERVICE

If an application is to have a long life cycle 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 which 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;
import android.os.IBinder;
import android.util.Log;

public class service1 extends Service implements Runnable
{
    public static final String tag = "service1";
    private int myiteration = 0;
    @Override
    protected void onCreate()
    {
        super.onCreate();
        Thread mythread = new Thread (this);
        mythread.start();
    }

    public void run()
    {
        while (true)
        {
            try
            {
                Log.i(tag,"service1 firing : # " + myiteration++);
            }
        }
    }
}
```

```

        Thread.sleep(10000);
    }
    catch(Exception ee)
    {
        Log.e(tag, ee.getMessage());
    }
}

@Override
public IBinder onBind(Intent intent)
{
    // TODO Auto-generated method stub
    return null;
}
}

```

5

1. This example requires that the package `android.app.Service` be imported. This package contains the `Service` class.
2. This example also demonstrates Android's Logging mechanism, which is useful for debugging purposes. This is discussed in further detail later in this book.
3. The `Service` class extends `Service`. It also implements `Runnable` to perform its main task on a separate Thread. This is a common paradigm, but it is not required to implement `Runnable`.
4. The `onCreate` method of the `Service` class permits the application to perform initialization type tasks.
5. The `onBind()` method is discussed in further detail in Chapter 4 when the topic of Binder and Interprocess 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 user interface 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 which is dedicated to processing `Intents`.

BROADCASTRECEIVER

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

- The application implements 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 desires 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 as the application is started automatically upon the triggering event. All of this house-keeping is managed by Android.
- An application registers itself at runtime via the `Context` class's `registerReceiver` method.

Like `Services`, `BroadcastReceivers` do not have a User Interface. 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`, however the use of these `Intents` are mutually exclusive from the `Intents` used to start an `Activity` or a `Service` as previously discussed.

An `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 there are a handful of methods 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 an 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 1
{
    public static final String tag = "MySMSMailBox"; 2

    @Override
    public void onReceive(Context context, Intent intent) 3
    {
        Log.i(tag, "onReceive");
        if (intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED")) 4
        {
            Log.i(tag, "Found our Event!"); 5
        }
    }
}
```

1. The class MySMSMailBox extends the BroadcastReceiver class. This is the most straight-forward way to employ an BroadcastReceiver. Note the class name of “MySMSMailBox”, as it will be used in the AndroidManifest.xml file, shown in code listing 1.4.
2. The tag variable is used in conjunction with the logging mechanism to assist in labeling messages sent to the console log on the emulator. Chapter 2 discusses the log mechanism.
3. The onReceive method is where all of the work takes place in an BroadcastReceiver. The BroadcastReceiver is required to implement this method. Note that a given BroadcastReceiver can register multiple IntentFilters and can therefore be instantiated for an arbitrary number of Intents.
4. It is important to make sure to handle the appropriate Intent by checking the action of the incoming Intent as shown.
5. 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 order for this BroadcastReceiver to “fire” and receive this Intent, it must be listed in the AndroidManifest.xml file as seen 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" /> 1
    <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" > 2
    </application>
</manifest>
```

```

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

```

1. Note that certain tasks on Android require the application to have a designated privilege. To give an application the required permissions, the `<uses-permission>` tag is used. This is discussed in further detail later in this chapter in the `AndroidManifest.xml` section.
2. The `<receiver>` tag contains the class name of the class implementing the `BroadcastReceiver`. Note that in this example, the class name is `"MySMSMailBox"` from the package named `"com.msi.manning.unlockingandroid"`.
3. The `IntentFilter` is defined in the `<intent-filter>` tag. The desired action is `"android.provider.Telephony.SMS_RECEIVED"`. 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.

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, placing phone calls, and as was demonstrated in this section's example, sending sms messages.

To send a sms message to the emulator, telnet to port 5554 (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>
```

Type help from the prompt to learn about other commands available.

These tools are discussed in more detail in Chapter 2.

Now that Intents and the various Android classes which process or handle Intents have been introduced, it is time to explore the next major Android application topic of this chapter, the Content Provider, 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 `ContentProvider` is used to access that data. The Content Provider implements a standard set of methods to permit applications access to 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 mechanisms 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 Content Provider 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 appropriate permissions.

Data stored in a Content Provider 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 that a filename is returned as

part of a Content Provider query, the file should not be accessed directly, but rather use the helper class ContentResolver's openInputStream method to access the binary data. This approach avoids Linux process/security hurdles as well as keeps all data access normalized through the Content Provider. Figure 1.6 outlines the relationship between ContentProviders, data stores, and their "clients".

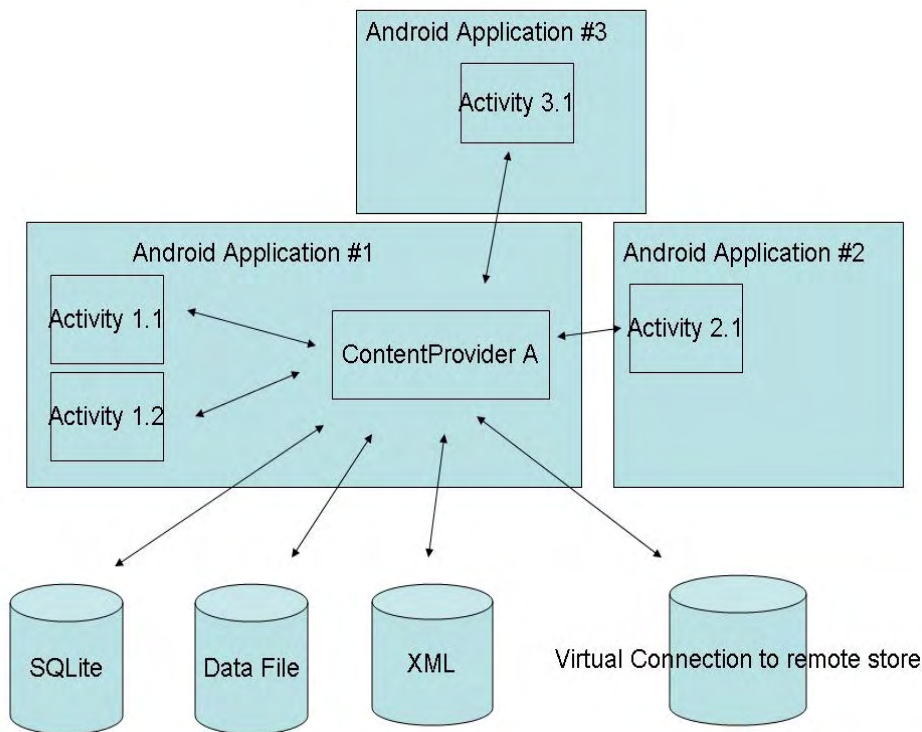


Figure 1.6 The Content Provider is the "data tier" for Android applications and is the prescribed manner in which data is accessed and shared on the device.

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 Content Provider might look like:

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

From this point, accessing a Content Provider is similar to using Structured Query Language in other platforms, though a complete SQL statement is not employed. A query is submitted to the Content Provider 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 Content Provider example is provided in Chapter 4.

NOTE

In many ways, a Content Provider acts like a database server. While an application could contain only a Content Provider and in essence be a database server, it is important to note that a Content Provider is typically a component of a larger Android application which hosts one or more Activity, Service and/or BroadcastReceiver as well.

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 at times. For example, have you ever had a piece of software that just didn't seem to 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 user name and password are used? What about the necessary libraries to run your application? These are all topics related to software deployment. An Android application requires a file named `AndroidManifest.xml`, which ties together the necessary pieces to run an Android application.

1.3.3 *AndroidManifest.xml*

The previous sections introduced the common elements of an Android application. To restate an important comment, an Android application will contain one or more Activity, Service, BroadcastReceiver, and 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 mechanism for this task 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. 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. `AndroidManifest.xml` files act as deployment descriptors for Android applications.

```
<?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=".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>
```

1
2
3

1. The manifest element contains the obligatory namespace as well as the Java package name containing this application.

2. This application contains a single Activity, with a class name of "chapter1". Note also the @string syntax. Any time an @ symbol is used in an `AndroidManifest.xml` file, it is referencing information stored in one of the resource files. For example, 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.

3. The Activity contains a single IntentFilter definition. The IntentFilter seen 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. Note that 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, as seen in Figure 1.7.



Figure 1.7 Applications are listed in the launcher based on their Intent Filter. In this example, the application titled “Where Do You Live” is available in the LAUNCHER category.

In addition to the elements seen 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" />
```

The AndroidManifest.xml file is revisited a number of times throughout the book as more detail needs to be added for certain elements.

Now that there is a basic understanding of the Android application and the AndroidManifest.xml file which describes its components, it is time to discuss how and where it actually executes. The next section discusses the relationship between Android applications and its Linux and Dalvik Virtual Machine runtime.

1.3.4 Mapping Applications to Processes

Android applications run each 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 operating system 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, non running 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 which are empty (loaded perhaps for performance caching purposes) or processes which have dormant Activities.

ps -a

The Linux environment is complete with 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 a real nice-to-have for troubleshooting application issues. It is unheard of on commercially available mobile phones to touch the “metal” in this fashion.

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

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”.

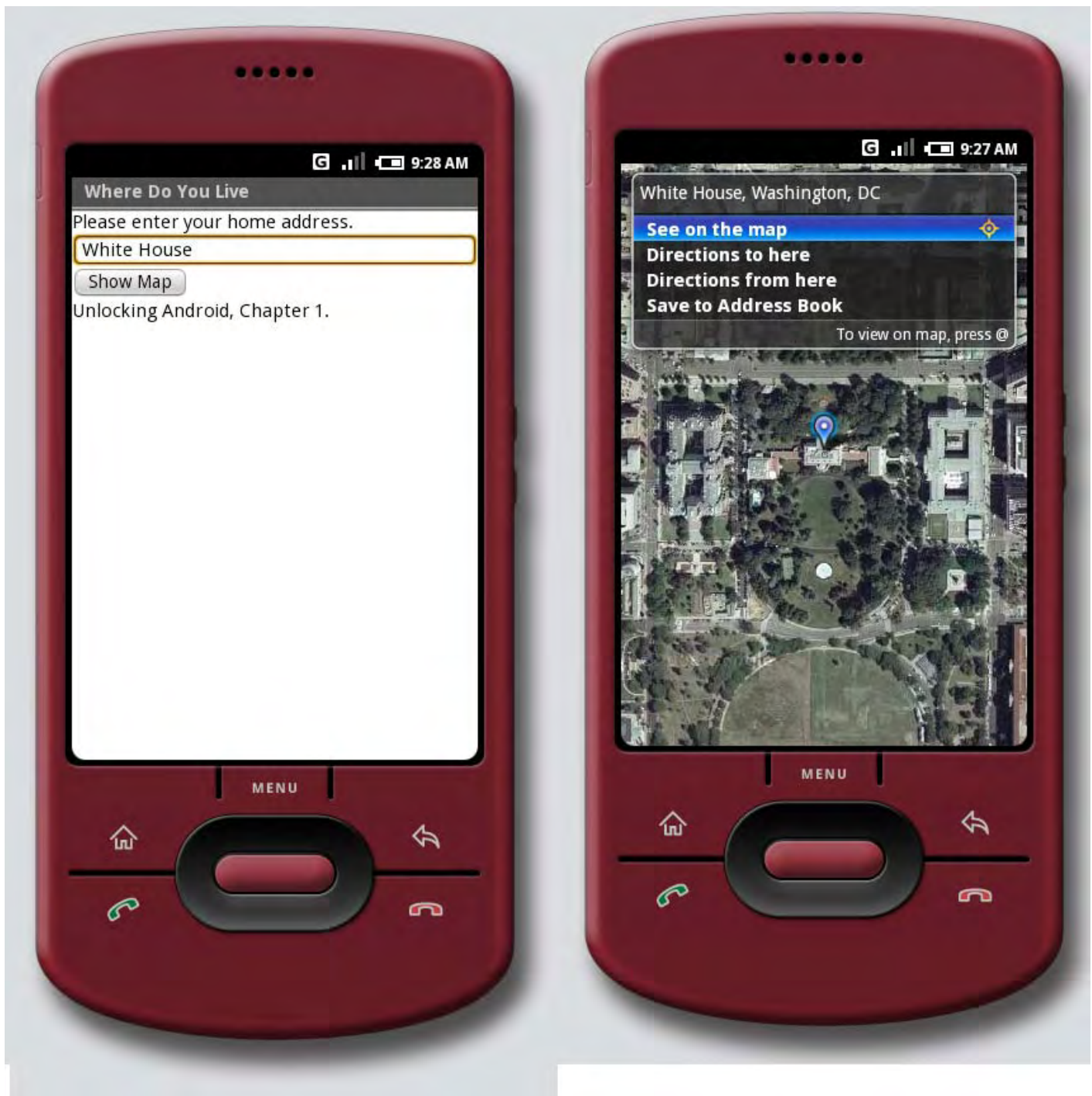


Figure 1.8 This Android application demonstrates a simple Activity and Intent.

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 seen 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 for this chapter's sample application with AWhereDoYouLive.java implements

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

import com.msi.manning.unlockingandroid.R;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;

public class AWhereDoYouLive extends Activity
{
    @Override
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.main);
1
    }

    final EditText addressfield = (EditText) findViewById(R.id.address);
2
    final Button button = (Button) findViewById(R.id.launchmap);
3
    button.setOnClickListener(new Button.OnClickListener()
    {
        public void onClick(View v)
        {
            try
            {
                String address = addressfield.getText().toString();
2
                address = address.replace(' ', '+');
                Intent myIntent = new Intent(android.content.Intent.ACTION_VIEW,
Uri.parse("geo:0,0?q=" + address));
5
                startActivity(myIntent);
6
            }
            catch (Exception e)
            {
                ...
            }
        }
    });
}
```

1. The setContentView method creates the primary User Interface, which is a layout defined in main.xml in the /res/layout directory

2. The EditText View collects information. It 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.
3. A Button object is connected to the launchmap user interface element.
4. The address is retrieved from the user interface element.
5. An Intent is created, requesting the ACTION_VIEW. The URI is a string representing a Geographic Search.
6. The Intent is initiated with a call to startActivity.

Resources are pre-compiled into a special class known as the “R” class, as seen in Listing 1.8. The final members of this class represent user interface 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 User Interface 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 seen 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 user interface elements for this chapter’s 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"
```

1

2

```

        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 & 2. Note the use of the '@' symbol in this resource's id attribute. This causes the appropriate entries to be made in to the R class via the automatically generated R.java file. These R class members are used in the calls to `findViewById()` as seen 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 as seen 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, briefly touched on market positioning and 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 the platform matures and actual 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 some devices sooner than later.

In this chapter the Android stack was examined and its relationship with Linux and Java was discussed. 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 big question 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, which code handles them, and provides a mechanism for delivering specific functionality to the platform, enabling 3rd party developers to deliver innovative solutions and products for Android. The main application classes of Activity, Service 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 user interface, 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.

2

Development Environment

In this chapter

- The Android SDK
- Using Eclipse for Android Development
- Command Line Tools
- The Android Emulator
- Running & Debugging an Android application
- The major Android SDK Java Packages

This chapter introduces the Android development tool chain and provides a hands-on guide to using the Android Development Tools as we walk through the creation, testing and debugging of a sample application. Upon completing this chapter, you will be familiar with using Eclipse and the Android Development Tools plug-in, navigating the Android SDK and its tools, running Android applications in the emulator, and debugging your application. With these skills in hand, we will take a quick look at the Java packages provided in the SDK to better equip you to embrace the development topics introduced later in this book as you prepare to develop your own Android applications.

The core task for a developer when embracing a new platform is getting an understanding of the Software Development Kit (SDK) with its various components. Let's start by examining the core components of the Android SDK and then transition into using the included tools to build and debug an application.

2.1 The Android Software Development Kit

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

TIP

The Android download page has instructions for installing the SDK, or you can refer to the Appendix of this book for detailed information on installing the required development tools.

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

2.1.1 The Application Programming Interface

The Java environment of Android can be broken down into a handful of key sections. Once you have an understanding of each of these areas, the Java-doc reference material that ships with the SDK becomes a real tool, and not just a pile of seemingly unrelated material. You may recall that Android is not strictly a J2ME software environment, however there is some commonality

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

between the Android platforms and other Java development platforms. The next few sections review some of the Java packages in the Android SDK and where they can be used. The remaining chapters of this book provide a deeper look into using many of these programming interfaces.

2.1.2 Core Android Packages

If you have developed in Java previously, you will recognize many familiar Java packages for core functionality. These include packages such as:

- `java.lang`: core Java language classes
- `java.io`: input/output capabilities
- `java.net`: network connections
- `java.util`: utility classes. This package includes the `Log` class used to write to the LogCat.
- `java.text`: text handling utilities
- `java.math`: math and number manipulation classes
- `javax.net`: network classes
- `javax.security`: security related classes
- `javax.xml`: DOM based XML classes
- `org.apache.*`: http related classes
- `org.xml`: SAX based XML classes

There are additional Java classes not listed here. Generally speaking there is minimal focus in this book on core packages listed here as our primary concern is Android development. With that in mind, let's look at the Android specific functionality found in the Android SDK.

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

- `android.app`: Android application model access
- `android.content`: accessing and publishing data in Android
- `android.net`: contains the `Uri` class, used for accessing various content
- `android.graphics`: graphics primitives
- `android.opengl`: OpenGL classes
- `android.os`: system level access to the Android environment
- `android.provider`: Content Provider related classes
- `android.telephony`: Telephony capability access
- `android.text`: Text layout
- `android.util`: collection of utilities for text manipulation, including XML
- `android.view`: User interface elements
- `android.webkit`: Browser functionality
- `android.widget`: more user interface elements

Some of these packages are absolutely core to Android application development including **`android.app`**, **`android.view`**, and **`android.content`**. Other packages are used to varying degrees depending on the type of applications being constructed.

2.1.3 Optional Packages

Not every Android device will have the same hardware and mobile connectivity capabilities, so there are some elements of the Android SDK which are considered "optional". Some devices will support these features, and others not. It is important that an application degrade gracefully if a feature is not available on a specific handset. Java packages to pay special attention to include those that rely on specific, underlying hardware and network characteristics, such as Location Based Services including Global Positioning System (GPS) and wireless technologies such as Bluetooth, IrDA, and WiFi (802.11).

This quick introduction to the Android SDK's programming interfaces is just that – quick and at-a-glance. The upcoming chapters of this book dig into the class libraries in further detail, so let's focus now on the tools required to build Android applications.

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

2.2 Fitting the pieces together

After installing the Android SDK along with the Android Development Tools plug-in for Eclipse, we're ready to explore the development environment. Figure 2.1 depicts the typical Android development environment including both real hardware and the useful Android Emulator. While not the exclusive tool required for Android development, Eclipse can play a big role in Android development as it provides not only a rich Java compilation and debugging environment, but also because with the Android Development Tools under Eclipse, we can manage and control virtually all aspects of testing our Android applications directly from the Eclipse IDE.

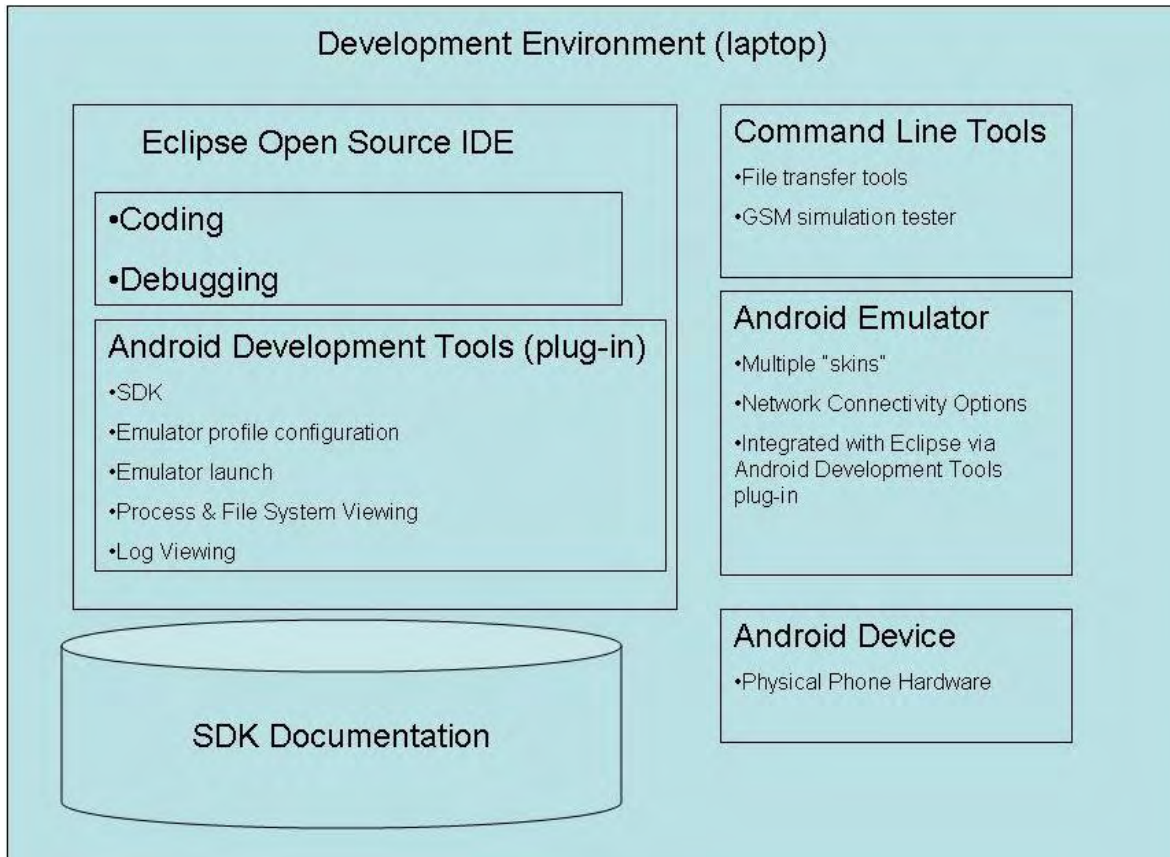


Figure 2.1: The development environment for building Android applications including the popular open source Eclipse IDE.

The key features of the Eclipse environment as it pertains to Android application development include:

▪

- Rich Java development environment including Java source compilation, class auto-completion and integrated JavaDoc.
- Source level debugging
- Android Emulator profile management & launch
- The Dalvik Debug Monitoring Service (DDMS)
 - Thread & Heap views
 - Emulator File System Management
 - Data and Voice Network control

- Emulator control
- System and Application Logging

Eclipse supports the concept of “Perspectives” where the layout of the screen has a set of related windows and tools. The windows and tools included in an Eclipse Perspective are known as “Views”. When developing Android applications, there are two Eclipse Perspectives which are of primary interest to us: the Java Perspective, and the Dalvik Debug Monitoring Service Perspective, or DDMS. Beyond those two Perspectives, the Debug Perspective is also available and useful when debugging an Android application. To switch between the available Perspectives in Eclipse, use the Open Perspective menu, found under the Window menu in the Eclipse IDE. Let’s examine the features of the Java and DDMS Perspectives and how they can be leveraged for Android development.

2.2.1 Java Perspective

The Java Perspective is where you will spend most of your time while developing Android applications. The Java Perspective boasts a number of convenient Views for assisting in the development process. The Package Explorer View allows us to see the Java projects in our Eclipse Workspace. Figure 2.2 shows the Package Explorer listing some of the sample projects for this book.



Figure 2.2: The Package Explorer allows us to browse the elements of our Android projects.

The Java Perspective is where you will edit your Java source code. Every time your source file is saved, it is automatically compiled by Eclipse’s Java Developer Tools (JDT) in the background. You need not worry about the specifics of the JDT, the important thing to know is that it is functioning in the background to make your Java experience as seamless as possible. If there is an error in your source code, the details will show up in the Problems View of the Java Perspective. Figure 2.3 has an intentional error in the source code to demonstrate the functionality of the Problems View. You can also put your mouse over the red “x” to the left of the line containing the error for a “tool-tip” explanation of the problem.

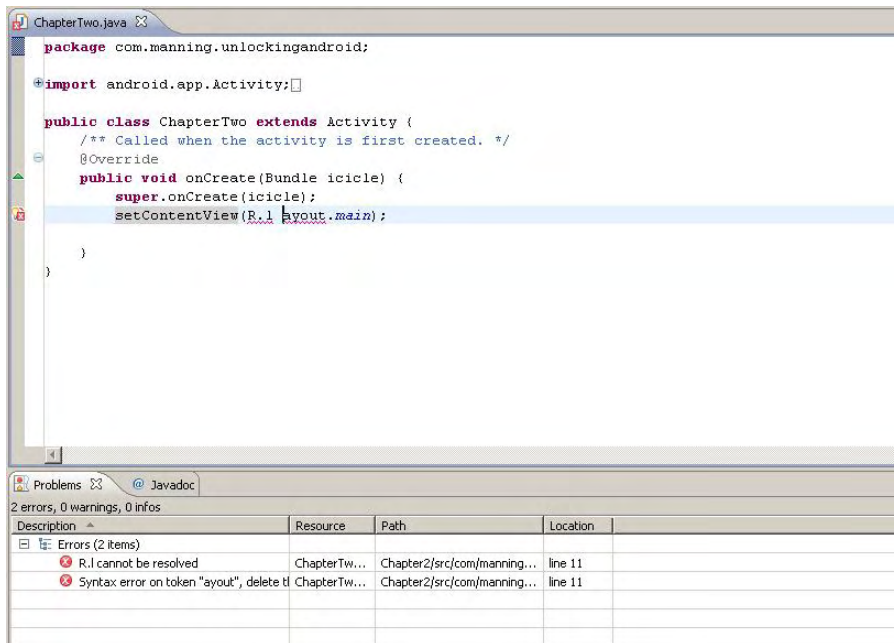


Figure 2.3: The Problems View shows any errors in your source code.

One of the very powerful features of the Java Perspective in Eclipse is the integration between the source code and the Javadoc View. The Javadoc View updates automatically to provide any available documentation about a currently selected Java class or method, as shown in Figure 2.4 where the Javadoc View displays information about the **Activity** class.

TIP

This chapter just scratches the surface in introducing the powerful Eclipse environment. If you want to learn more about Eclipse, you might consider *Eclipse in Action*, published by Manning and available online at <http://manning.com>.

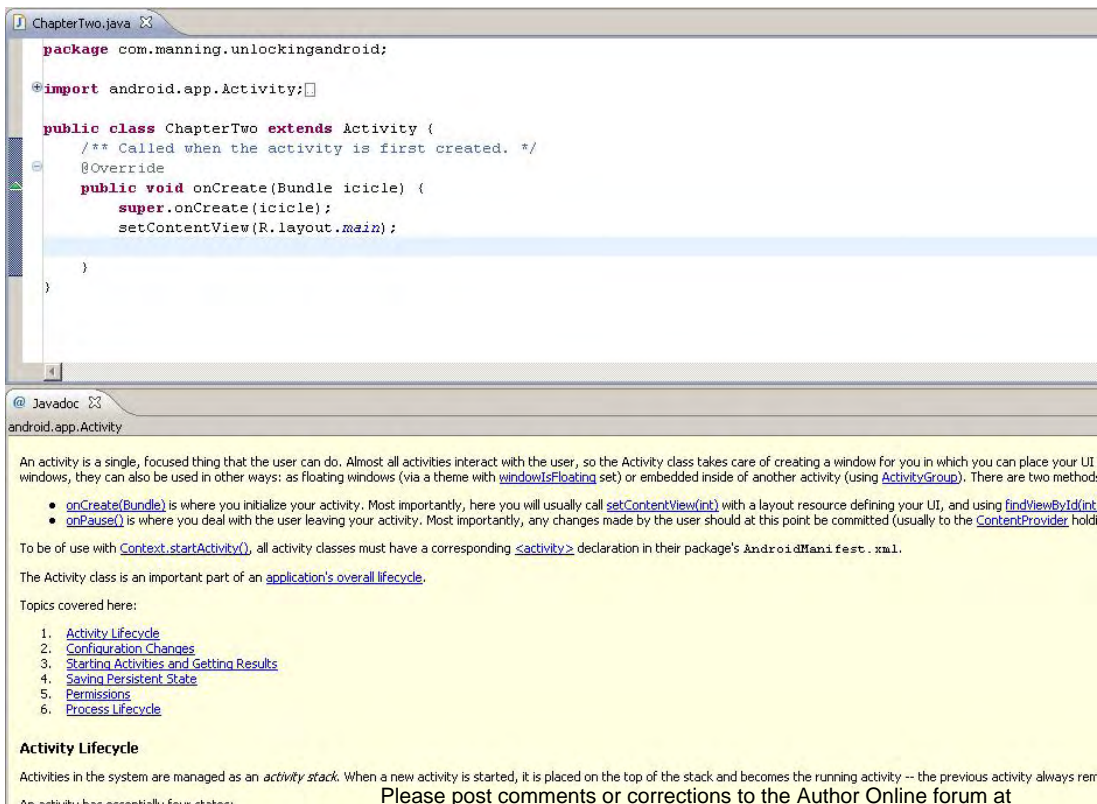


Figure 2.4: The JavaDoc View provides context-sensitive documentation, in this case for the Activity class.

TIP

It is easy to get the Views in the current Perspective into a layout that may not be desirable. If this occurs, you have a couple of choices to restore the Perspective to a more useful state. The first option is to use the Show View menu under the Window menu to display a specific View. Alternatively, you can select the Reset Perspective menu to restore the Perspective to its default settings.

In addition to the JDT which compiles java source files, the Android Development Tools automatically compile Android specific files such as layout and resource files. We'll learn more about the underlying tools later in this chapter, but now it is time to have a look at the Android-specific Perspective found in the Dalvik Debug Monitoring Service.

2.2.2 DDMS Perspective

The DDMS Perspective provides a dashboard-like view into the heart of a running Android device, or in our case, a running Android Emulator. Figure 2.5 shows the emulator running Chapter 2's sample application with the DDMS in the background.

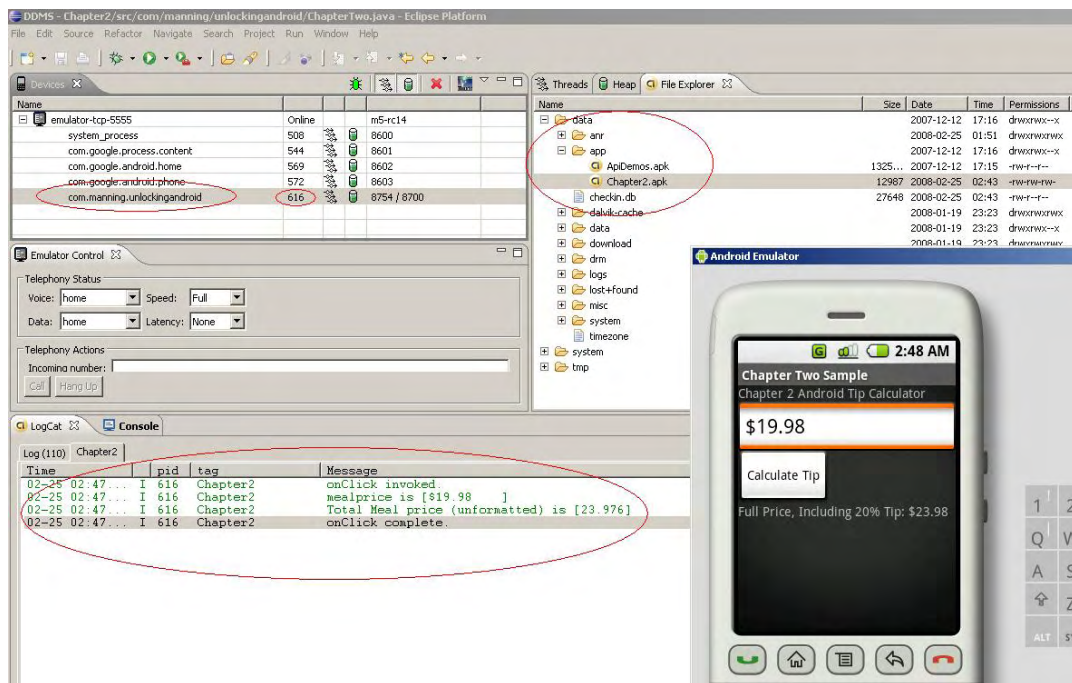


Figure 2.5: DDMS Perspective with an application running in the Android Emulator.

Don't worry, we'll walk through the details of the application, including how to build the application and how to start it running in the Android Emulator, but first, let's see what we can learn from the DDMS to continue the discussion of the tools available to us for Android development. The Devices View shows a single emulator session, entitled **emulator-tcp-5555**. This means that there is a connection to the Android Emulator at TCP/IP port 5555. Within this emulator session, there are five processes running. The one of interest to us is **com.manning.unlockingandroid**, with a process id of **616**.

TIP

Unless you are testing a peer to peer application, you will typically only have a single Android Emulator session running at once. It is possible to have multiple instances of the Android Emulator running concurrently.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
Licensed to Thow Way Chiam <ken.ctw@gmail.com>

Logging is an essential tool in software development, and that brings us to the LogCat View of the DDMS Perspective. This view provides a glimpse at system and application logging taking place in the Android Emulator. In Figure 2.5, a filter has been setup for looking at entries with a **tag** of "Chapter2"; using a filter on the LogCat is a helpful practice as it can reduce the noise of all the logging entries and allow us to focus on just our own application's entries. There are four entries in the list matching our filter criteria. We'll look at the source code soon to see how we get our messages into the log. Note that these log entries have a column showing the process id, or pid, of the application contributing the log entry. As expected, the pid for our log entries is 616, matching our running application instance in the emulator.

The File Explorer View is seen in the upper right of Figure 2.5. User applications, i.e., the ones you and I write, are deployed with a file extension of .apk and are stored in the /data/app directory of the Android device. The File Explorer View also permits file system operations such as copying files to and from the Android Emulator as well as removing files from the emulator's file system. Figure 2.6 shows the process of deleting a user application from the /data/app directory.

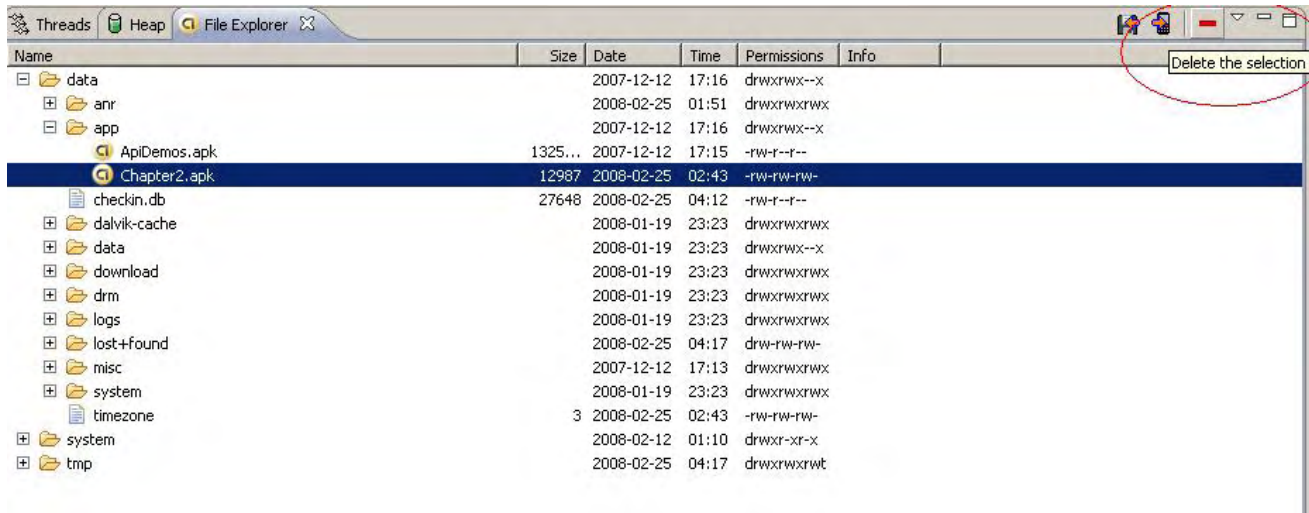


Figure 2.6: Deleting applications from the emulator by highlighting the application file and clicking the delete button.

Obviously being able to casually browse the file system of our "mobile phone" is a great convenience. This is a nice feature to have for mobile development where we are often relying on cryptic popup messages to help us along in the application development and debugging process. With easy access to the file system we can work with files and readily copy them to and from our development computer platform as necessary.

In addition to exploring a running application, the DDMS Perspective provides tools for controlling the emulated environment. For example, the Emulator Control View allows the testing of various connectivity characteristics for both Voice and Data networks such as simulating a phone call or receiving an incoming SMS. Figure 2.7 demonstrates sending an SMS message to the Android Emulator.

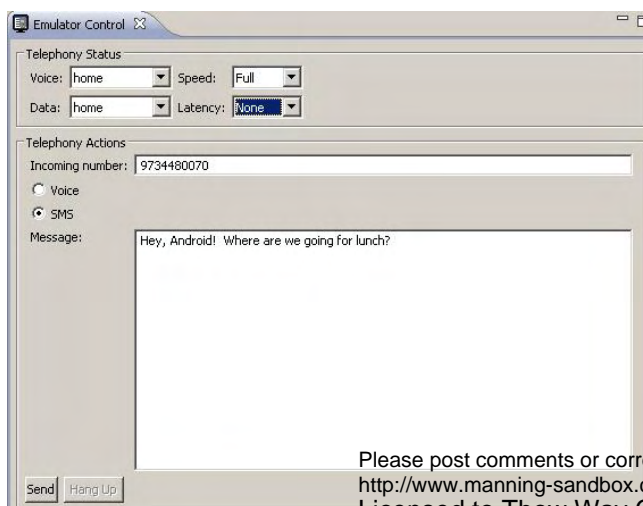


Figure 2.7: Sending a test SMS to the Android Emulator.

The DDMS provides quite a bit of visibility into, and control over, the Android Emulator and is a handy tool for evaluating our Android Applications. Before we move on to building and testing Android applications, it is helpful to understand what is happening behind the scenes and enabling the functionality of the DDMS.

2.2.3 Command Line Tools

The Android SDK ships with a collection of command line tools, which can be found in the tools subdirectory of your Android SDK installation. While Eclipse and the Android Development Tools provide a great deal of control over our Android development environment, sometimes it is nice to exercise greater control, particularly when considering the power and convenience that scripting can bring to a development platform. We are going to explore two of the command line tools found in the Android SDK.

TIP

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

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

or, use the following command for Mac OSX and Linux:

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

Android Asset Packaging Tool (aapt)

So you may be wondering just how files such as the layout file main.xml get processed and exactly where does the R.java file come from? Who ZIPs up the application file for us into the apk file? Well, you may have already guessed, but it is the Android Asset Package Tool, or as we call it from the command line, aapt. aapt is a versatile tool which combines the functionality of pkzip or jar along with an Android-specific resource compiler. Depending on the command line options provided to it, aapt wears a number of hats and assists with our design-time Android development tasks. To learn the functionality available in aapt, simply run it from the command line with no arguments. A detailed usage message is written to the screen.

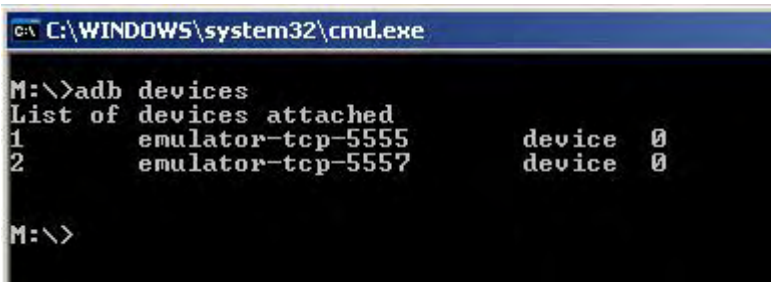
While aapt helps with design time tasks, another tool, the Android Debug Bridge assist us at runtime to interact with the Android Emulator.

Android Debug Bridge (adb)

The Android Debug Bridge utility permits us to interact with the Android Emulator directly from the command line or script. Have you ever wished you could navigate around the file system on your smart phone? Well, now you can with the adb! The adb works as a client/server tcp based application. While there are a couple of background processes that run on the development machine and the emulator to enable our functionality, the important thing to understand is that when we run adb, we get access to a running instance of the Android Emulator. Here are a couple of examples of using adb. First, let's look to see if we have any available Android Emulator sessions running:

```
adb devices<return>
```

This command will return a list of available Android Emulators, for example, Figure 2.8 shows adb locating two running emulator sessions.



```
C:\WINDOWS\system32\cmd.exe
M:\>adb devices
List of devices attached
1 emulator-tcp-5555 device 0
2 emulator-tcp-5557 device 0
M:\>
```

Figure 2.8: The adb tool provides interaction at “run time” with the Android Emulator.

Let's connect to the first Android Emulator session and see if our application is installed. We connect with the syntax **adb shell**. This is how we would connect if we had a single Android Emulator session active, but because there are two emulators running, we need to specify an identifier to connect to the appropriate session:

```
adb -d 1 shell
```

Figure 2.9 shows off the Android file system and demonstrates looking for a specific installed application, namely our Chapter 2 sample application, which we'll be building in a in the next section.



```
C:\WINDOWS\system32\cmd.exe - adb -d 1 shell

M:\>adb devices
List of devices attached
1      emulator-tcp-5555      device 0
2      emulator-tcp-5557      device 0

M:\>adb -d 1 shell
# cd /data/app
cd /data/app
# ls -l
ls -l
-rw-rw-rw- root    root    12982 2008-02-28 20:34 Chapter2.apk
-rw-r--r-- system system 1325833 2007-12-12 17:15 ApiDemos.apk
# _
```

Figure 2.9: Using the shell command, we can browse Android's file system.

This capability can be very handy when we want to remove a specific file from the emulator's file system, kill a process or generally interact with the operating environment of the Android Emulator. If you download an application from the Internet for example, you can use the adb command to install an application. For example, the command

```
adb shell install someapplication.apk
```

installs the application named *someapplication* to the Android Emulator. The file is copied to the `/data/app` directory and is accessible from the Android application launcher. Similarly, if you desire to remove an application, you can run adb to remove an application from the Android Emulator. For example, if you desire to remove the Chapter2.apk sample application from a running emulator's file system, you can execute the following command from a terminal or Windows command window:

```
adb shell rm /data/app/Chapter2.apk
```

Mastering the command line tools in the Android SDK is certainly not a requirement of Android application development, but having an understanding of what is available and where to look for capabilities is a good skill to have in your toolbox. If you need assistance with either the aapt or adb commands, simply enter the command at the terminal and a fairly verbose usage/help page is displayed. Additional information on the tools may be found in the Android SDK documentation.

TIP

The Android file system is a Linux file system. While the adb shell command does not provide a very rich shell programming environment as is found on a desktop Linux or Mac OSX system, the basic commands such as ls, ps, kill, rm and the like are available. If you are new to Linux, you may benefit from learning some very basic shell commands.

One other tool you will want to make sure you are familiar with is telnet. Telnet allows you to connect to a "remote system" with a character-based user interface. In this case, the remote system you connect to is the Android Emulator's console. You can accomplish this with the following command:

```
telnet localhost 5554
```

In this case, localhost represents your local development computer where the Android emulator has been started because the Android Emulator relies on your computer's loopback IP address of 127.0.0.1. Why port 5554? Recall when we employed `adb` to find running emulator instances that the output of that command included a name with a number at the end. The first Android Emulator can generally be found at IP port 5555. No matter which port number the Android Emulator is found at, the Android Emulator's console may be found at a port number equaling 1 less. For example, if the Android Emulator is running and listed at port 5555, the console is at 5554.

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

It is time to roll up our sleeves and write an Android application to exercise the development environment we have been learning about.

2.3 Building an Android Application in Eclipse

We are going to build a simple application which will provide us an opportunity to modify the user interface, provide a little application logic and then execute the application in the Android Emulator. More complex applications are left for later chapters – our focus is on the development tools. Building an Android application is not too much different from creating other types of Java applications in the Eclipse IDE. It all starts with File | New and selecting an Android application as the build target.

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

2.3.1 Android Project Wizard

The most straight-forward manner to create a new Android application is to utilize the services of the Android Project Wizard, which is part of the Android Development Tools plug-in. The wizard provides a simple means to define the Eclipse project name and location, the Activity name corresponding to the main user interface class as well as a name for the application. Of importance also is the Java package name under which the application is created. Once this application is created, it is easy to add new classes to the project.

NOTE

In this example, we are creating a brand new project in the Eclipse workspace. This same wizard may be used to import source code from another developer, such as the sample code for this book.

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

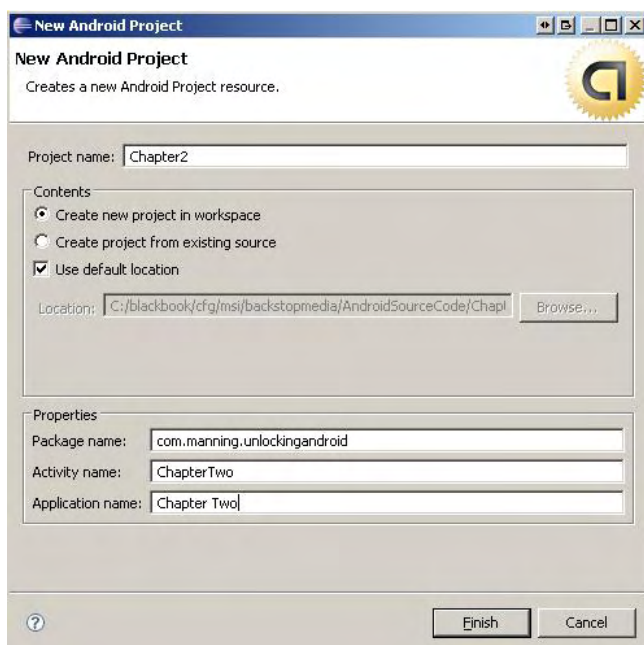


Figure2.10: Using the Android Project Wizard, it is easy to create an empty Android application, ready for customization.

TIP

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

Clicking on Finish creates our sample application. At this point the application compiles and is capable of running on the emulator – no further development steps are required. Of course, what fun would an empty project be? Let's flesh out this sample application, an Android Tip Calculator.

2.3.2 Android Sample Application Code

The Android Application Wizard takes care of a number of important elements in the Android application structure, including the Java source files, the default resource files and the AndroidManifest.xml file. Looking at the Package Explorer View in Eclipse we can see all of the elements of this application. Here is a quick description of the elements included in our sample application:

- The src folder contains two java source files automatically created by the wizard.
 - **ChapterTwo.java** contains the main Activity for the application. This file will be modified to add our sample application's tip calculator functionality.
 - **R.java** contains identifiers for each of the user interface resource elements in the application. It is important that you never modify this file directly, as it is automatically regenerated every time a resource is modified and any manual changes you make will be lost the next time the application is built.
- Android.jar contains the Android runtime Java classes. This is a reference to the **android.jar** file found in the Android SDK.
- The res folder contains all of the Android resource files, including:
 - Drawables - contains image files such as bitmaps and icons. The wizard includes a default Android icon named **icon.png**.
 - Layout - contains an xml file called **main.xml**. This file contains the User Interface elements for the primary View of our Activity. We will modify this file, however we will not be making any significant or special changes - just enough to accomplish our meager user interface goals for our Tip Calculator. User interface elements such as Views are covered in detail in Chapter 3. It is not uncommon for an Android application to have multiple xml files in the Layout section.
 - Values contains the **strings.xml** file. This file is used for localizing string values such as the application name and other strings used by your application.
- **AndroidManifest.xml** represents the deployment information for this project. While AndroidManifest.xml files can become somewhat complex, this chapter's manifest file can run without modification as no special permissions are required.

Now that we know what is "in" the project, let's review how we are going to modify the application. Our goal with the Android Tip Calculator is to permit our user to enter the price of a meal and then select a button to calculate the total cost of the meal, tip included. In order to accomplish this, we need to modify two files, namely ChapterTwo.java, and the user interface layout file, main.xml. Let's start with the user interface changes by adding a few new elements to the primary View, as seen in Listing 2.1.

Listing 2.1: Main.xml contains User Interface elements

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    Please post comments or corrections to the Author Online forum at
    http://www.manning-sandbox.com/forum.jspa?forumID=411
    Licensed to Thow Way Chiam <ken.ctw@gmail.com>
```



```

        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Chapter 2 Android Tip Calculator"
    />
    <EditText
        android:id="@+id/mealprice"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:autoText="true"
    />
    <Button
        android:id="@+id/calculate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Calculate Tip"
    />
    <TextView
        android:id="@+id/answer"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=""
    />

</LinearLayout>

```

1. Modify the text view (or label) at the top of the screen to be more descriptive of our application.
2. Add an EditText field, providing a means for the user to enter the price of their meal.
3. The EditText element has an attribute of type **android:id**, with a value of **mealprice**. This allows us to manipulate this element from our code. If a user interface element is static and does not need to be set or read from our application code, this android:id attribute is not required.
4. A Button named **calculate** is added to the View.
5. A TextView named **answer** is provided for displaying our total cost, including tip.

When we save the main.xml file, the file is processed by the Android Developer Tools plug-in, to compile the resource and generate an updated R.java file. Try it for yourself. Modify one of the id values in the main.xml file, save the file and then open R.java to have a look at the constants generated there. Remember not to modify the R.java file directly as all of your changes will be lost! If you conduct this experiment, be sure to change the values back as they are listed here to make sure the rest of the project will compile as-is. Provided we have not introduced any syntactical errors into our main.xml file, our user interface file is complete.

TIP

Through the maturation of the still very young Android Development Tools, the plugins for Eclipse have offered increasingly useful “resource editors” for manipulating the layout xml files. This means that you do not need to rely on editing the xml files directly “by hand”.

It is time to turn our attention to the file ChapterTwo.java to implement the desired Tip Calculator functionality. ChapterTwo.java is seen in Listing 2.2.

Listing 2.2: ChapterTwo.java implements the Tip Calculator logic.

```

package com.manning.unlockingandroid;
import com.manning.unlockingandroid.R;

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

import android.app.Activity; #3
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import java.text.NumberFormat; #4
import android.util.Log; #5

public class ChapterTwo extends Activity {

    public static final String tag = "Chapter2"; #6

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

        final EditText mealpricefield = (EditText) findViewById(R.id.mealprice); #7
        final TextView answerfield = (TextView) findViewById(R.id.answer); #8

        final Button button = (Button) findViewById(R.id.calculate); #9
        button.setOnClickListener(new Button.OnClickListener() #10
        {
            public void onClick(View v)
            {
                try
                {
                    //Perform action on click
                    Log.i(tag,"onClick invoked."); #11

                    // grab the meal price from the UI
                    String mealprice = mealpricefield.getText().toString(); #12

                    Log.i(tag,"mealprice is [" + mealprice + "]);
                    String answer = "";

                    // check to see if the meal price includes a "$"
                    if (mealprice.indexOf("$") == -1)
                    {
                        mealprice = "$" + mealprice;
                    }

                    float fmp = 0.0F;

                    // get currency formatter
                    NumberFormat nf =
                        java.text.NumberFormat.getCurrencyInstance(); #13

                    // grab the input meal price
                    fmp = nf.parse(mealprice).floatValue();

                    // let's give a nice tip -> 20%
                    fmp *= 1.2; #14

                    Log.i(tag,"Total Meal Price (unformatted) is [" + fmp + "]);
                    // format our result
                    answer =
                        "Full Price, Including 20% Tip: " + nf.format(fmp); #15

                    // display the answer
                    answerfield.setText(answer); #16

                    Log.i(tag,"onClick complete.");

                }
                catch (java.text.ParseException pe) #17
                {
                    Log.i(tag,"Parse exception caught");
                    answerfield.setText("Failed to parse amount?");
                }
                catch (Exception e)
                {

```

```

        Log.e(tag, "Failed to Calculate Tip:" + e.getMessage());
        e.printStackTrace();
        answerfield.setText(e.getMessage());
    }
}
};
}
}

```

1. This class is part of the **com.manning.unlockingandroid** package. This package value was brought over from the Application Wizard automatically.
2. We import the **com.manning.unlockingandroid.R** class to gain access to the definitions used by the User Interface. Note that this step is not actually required because the R class is part of the same application package, however it is helpful to include this import as it makes our code easier to follow. It should also be noted that there are some built in User Interface elements in the "R" class. Some of these built-in elements are introduced later in the book as part of sample applications.
3. A number of imports are necessary to resolve class names in use.
4. The **java.text.NumberFormat** class is used to handle currency values.
5. The **android.util.Log** class is employed to make entries to the log. Calling static methods of the Log class adds entries to the log viewed by the LogCat View of the DDMS Perspective.
6. The **tag** is a Java String value used to mark our log entries and differentiate them from other log entries. We can filter on this string value so we don't have to sift through the hundreds and thousands of LogCat entries to find our few debugging or informational messages.
7. We connect the User Interface element containing **mealprice** to a class level variable of type **EditText** by calling the **findViewById** method, passing in the identifier for the mealprice, as defined by our automatically generated R class, found in R.java. With this reference, we can access the user's input and manipulate the data.
8. We connect the User Interface element for displaying the calculated answer back to the user, again by calling the **findViewById** method.
9. Obtain a reference to the button so we can add an event listener.
10. The button is an essential user interface element to our sample application. We want to know when it has been "clicked". This is accomplished by adding a new **OnClickListener** method named **onClick**.
11. When the **onClick** method is invoked, we add the first of a few log entries using the **i** method of the Log class. This adds an entry to the log with an "Information" classification. The Log class contains methods for adding entries to the log for different levels, including Verbose, Debug, Information, Warning and Error.
12. Now that we have a reference to the mealprice User Interface element, we can obtain the text entered by our user with the **getText()** method of the EditText class.
13. Get the static currency formatter.
14. Add a nice tip to the base meal price, $.2 = 20\%$
15. Format the full meal cost, including tip.
16. Using the **setText()** method of the **TextView** user interface element named **answerfield**, we update the User Interface to tell the user the total meal cost.
17. It is a good practice to put code logic into Try/Catch blocks to keep our applications behaving when the "unexpected" occurs.

There are of course additional files in this sample project, but we are really only concerned with modifying the application enough to get some custom functionality working in this chapter. You will notice that as soon as we save our source files, the Eclipse IDE compiles the project source files in the background. If there are any errors, they are listed in the Problems View of the Java Perspective as well as marked in the left hand margin with a small red "x" to draw our attention to them.

TIP

Using the command line tools found in the Android SDK, you can create batch builds of your applications without the use of the IDE. This approach is useful for software shops with a specific configuration management function and a desire to conduct automated builds. In addition to the Android-specific build tools found under the tools subdirectory of your Android SDK installation, you will also require a Java Developer Kit (JDK, version 5 or better) in order to complete command line application builds. Automating builds of Android applications is beyond the scope of this book, however you can learn more about the topic of build scripts by reading two Manning titles on the topic: Java Development with Ant found at <http://www.manning.com/hatcher/> and Ant in Action found at <http://www.manning.com/loughran/>.

Assuming there are no errors in the source files, our classes and user interface files will compile properly. But what actually needs to happen before our project can be run and tested in the Android Emulator?

2.3.3 Building the Application

Now that the application compiles, the runtime package must be built. Recall that despite the compile-time reliance upon Java, Android applications do not run in a Java virtual machine. Instead, the Android SDK employs the Dalvik VM. This means that Java byte codes created by the Eclipse compiler must be converted to the .dex file format for use in the Android runtime. The Android SDK has tools to perform these steps, but the ADT takes care of all of this for us transparently.

The Android SDK contains tools which convert the project files into a file ready to run on the Android Emulator. Figure 2.11 depicts the generalized flow of source files in the Android build process. If you recall from our earlier discussion of Android SDK tools, the tool used at “design time” is aapt.exe. Application resource xml files are processed by aapt with the R.java file created as a result – remember we need to refer to the R class for user interface identifiers when connecting our code to the UI. Java source files are first compiled to class files by our Java environment, typically Eclipse and the JDT. Once compiled, they are then converted to .dex files to be compatible with Android’s use of the Dalvik Virtual Machine. The project’s xml files are converted to a binary representation, though on the device, they retain the xml file extension.

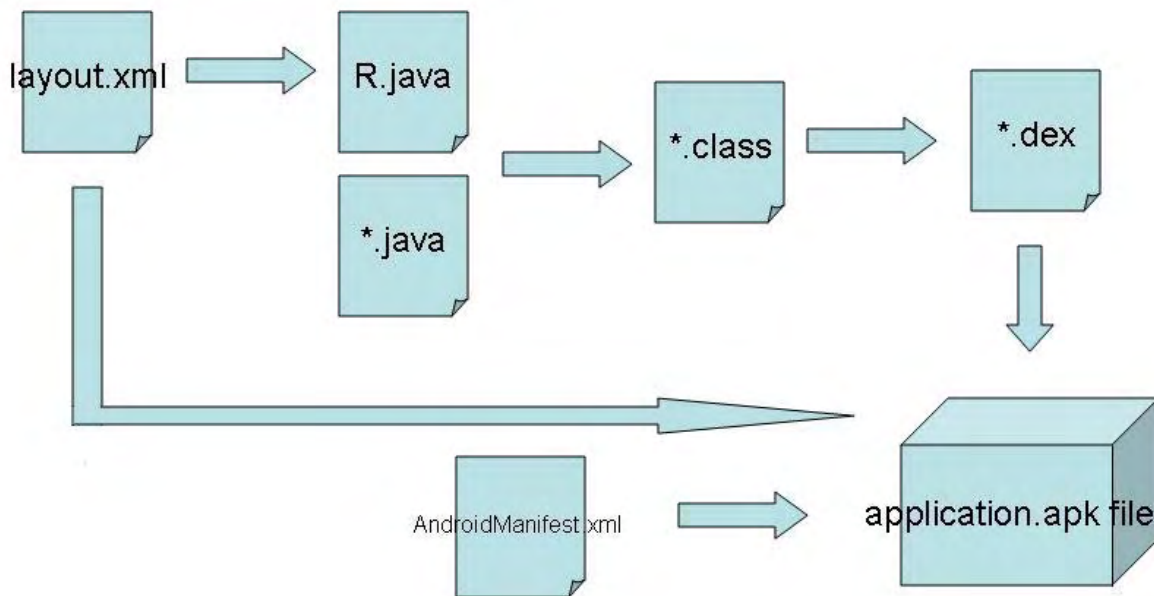


Figure 2.11: The ADT employs tools from the Android SDK to convert source files to a package ready to run on an Android device or emulator.

The converted xml files, a compiled form of the non-layout resources including the Drawables and Values, and the dex file (classes.dex) are packaged by the aapt tool into a file with a naming structure of projectname.apk. The resulting file can be read

with a PKZIP compatible reader such as WinRAR, WinZip, or the Java archiver, jar. Figure 2.12 shows this chapter's sample application in WinRAR.

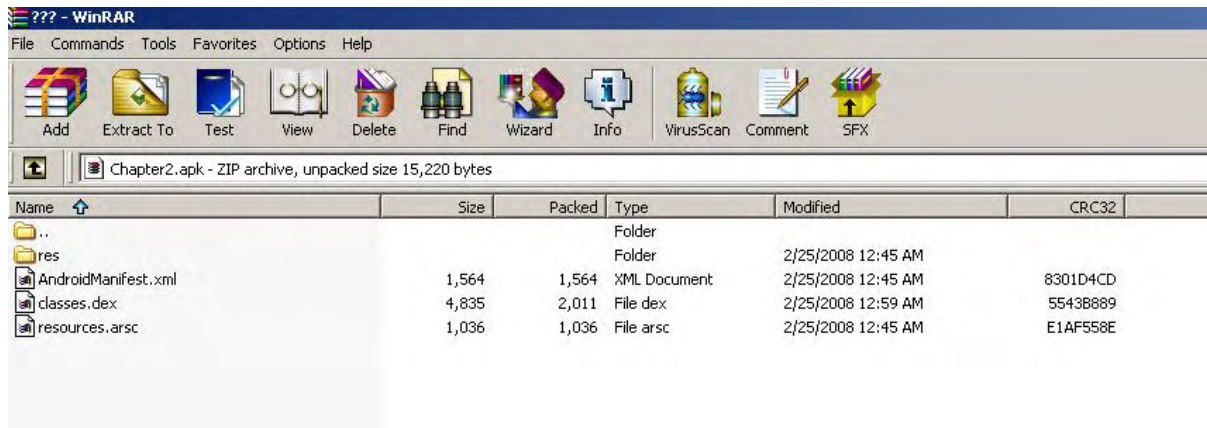


Figure 2.12: The Android application file format is ZIP compatible.

We're finally ready to run our application on the Android Emulator. It is important to become comfortable with working in an emulated environment when doing any serious mobile software development. There are many good reasons to have a quality emulator available for development and testing. One simple reason is that having multiple real devices with requisite data plans is a very expensive proposition. A single device may be hundreds of dollars alone. If the Open Handset Alliance has its way, Android will find its way onto multiple carriers with numerous devices, often with varying capabilities. Having one of every device is not practical for all but the development shops with the largest of budgets. For most of us, a device or two and the emulator will have to suffice. Let's focus on the strengths of emulator based mobile development.

2.4 The Android Emulator

While the best test of an application is running it on the actual hardware for which it was designed, an emulator often makes the job of the developer much easier. Working in an emulated environment permits a more rapid compile, run, and debug iterative cycle than is typically available when testing on a real hardware device. Taking the time to sync, or copy, an application to a real device typically takes longer than starting an emulator session. Also, it is easier to clean the "file system" of an emulator than performing the same maintenance operation on a real device. When you add in the capability of "scripting" commands to/from the emulator, it becomes an option worthy of investigation.

Beyond being a "faster tool" than working with a real device, there are physical characteristics of a device which the emulator tool must consider, primarily the screen dimensions, input devices and network connectivity.

2.4.1 Skins

Not all mobile devices are created equally, so it is important to be able to accommodate and test varying device characteristics in an emulated environment. The Android SDK comes with an emulator with distinct "skins". The skins represent different hardware layouts as well as portrait and landscape orientations. Figure 2.13 shows two emulator views, one in portrait with a hidden QWERTY keypad, the other in landscape mode with a visible keyboard.



Figure 2.13: The Android SDK includes 006Multiple emulator skins for testing a variety of device configurations.

Not only is it important to understand and accommodate how the device looks, it is important to understand what connectivity options a device is able to offer. Have you ever tested a mobile application in an area where there is excellent data coverage only to find out that where the application is going to be used in the field often has only marginal data service? The ability to test this condition in the confines of our development environment gives a real advantage to the application developer. Fortunately, the Android Emulator permits this kind of testing.

2.4.2 Network Speed

Network speed simulation is a key element of mobile software development. This feature is important because the actual user experience will vary during real-world use and it is important that mobile applications degrade gracefully in the absence of a reliable network connection. The Android emulator provides for a rich set of emulation tools for testing various network conditions and speeds. Table 2.1 lists the available network speed and latency conditions available in the Android Emulator.

Table 2.1: The Android emulator supports a variety of Network Speed options.

Network Speed	Full Speed (Use the development environment's full Internet connection)
	GSM
	HSCSD
	GPRS
	EDGE
	UMTS
	HSDPA
Network Latency	None – no latency introduced at all
	GPRS
	EDGE
	UMTS

The higher speed network environment found in the Android Emulator is welcome when testing core features of our applications. This is because functional test cases are often run hundreds or even thousands of times before releasing a product. If we had to compile the application, sync or copy the application to the device, and then run our application over a wireless data network, the testing time adds up quickly, reducing the number of tests performed in a given amount of time and elevating the associated costs. Worse yet, the challenges of mobile data connectivity testing may entice us to minimize application testing! Considering that most software development time frames are aggressive, every moment counts, so a quality emulator environment is valuable for rapid and cost-effective mobile application development activities. Also, it is important to consider the fact that there may be usage charges for voice and data consumption on a mobile communications plan. Imagine paying by the kilobyte for every downloaded data packet when testing a new streaming audio player!

Emulator vs. Simulator

You may hear the words emulator and simulator thrown about interchangeably. While they have a similar purpose – testing applications without the requirement of real hardware, those words should be used with care. A simulator tool works by creating a testing environment which behaves as close to 100% of the same manner as the “real” environment, however it is just an approximation of the real platform. However, this does not mean that the code targeted for a simulator will run on a real device because it is compatible only at the “source code” level. Simulator code is often written to be run as a software program running on our desktop computer with Windows DLLs or Linux libraries that mimic the Application Programming Interfaces (APIs) available on the real device. In the build environment, you typically select the CPU type for a target, and that is often x86/Simulator. However, in an emulated environment, the target of our projects is compatible at the binary level. The code we

write works on an emulator as well as the real device. Of course, some aspects of the environment differ in terms of how certain functions are implemented on an emulator. For example, a network connection on an emulator will run through your development machine's network interface card, whereas the network connection on a real phone runs over the wireless connection such as GPRS or Edge networks. Emulators are preferred as they more reliably prepare us for running our code on real devices. Fortunately, the environment available to Android developers is an emulator, not a simulator.

The Android SDK contains a command line program named `emulator.exe` which runs the Android Emulator. There are many command line switches available in the Android Emulator permitting us to customize the emulator's environment, how it looks and how it behaves. Some of these options are exposed in the Eclipse IDE via the Android Developer Tools plug-in. The majority of our focus is on employing the Android Emulator from Eclipse, but you are encouraged to examine the command line options available in the `emulator.exe` tool as they will undoubtedly be of value as you progress to building more complex Android applications and your application testing requirements grow.

2.4.3 Emulator Profiles

At this point, our sample application, the Android Tip Calculator has compiled successfully. We now want to run our application in the Android Emulator.

TIP

If you have had any trouble building the sample application, now would be a good time to go back and clear up any syntax errors preventing the application from building. In Eclipse you can easily see errors as they are marked with a red "x" next to the project source file and on the offending line(s). If you continue to have errors, make sure that your build environment is setup correct. You can refer to the Appendix of this book for details on configuring the build environment.

Our approach is to create a new Android Emulator profile so we can easily re-use our test environment settings. Our starting place is the Open Run Dialog menu in the Eclipse IDE, as seen in Figure 2.14.

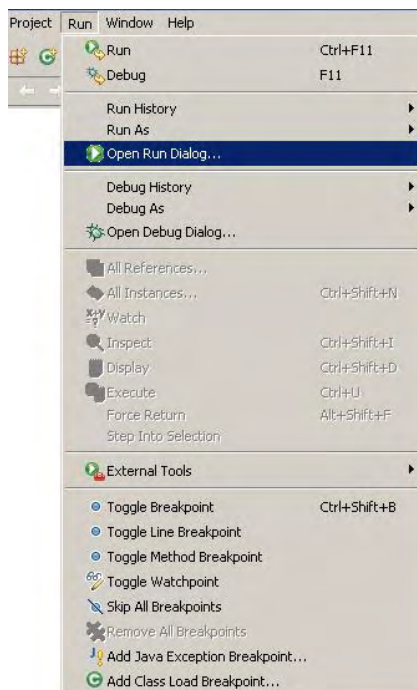


Figure 2.14: Creating a new Launch configuration for testing our Android application

We want to create a new Launch Configuration, as shown in Figure 2.15. To begin this process, highlight the "Android Application" entry in the list to the left and click on the "New Launch Configuration" button, shown encircled in red in Figure 2.15.

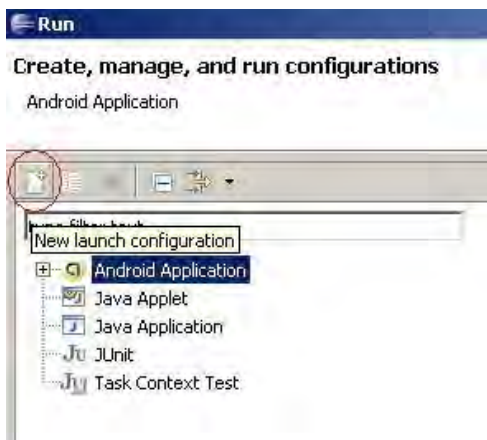


Figure 2.15: Select the Android Application Run template

We now want to give our launch configuration a name that we can readily recognize. Keep in mind that we are going to have quite a few of these launch configurations on the menu, so give the name something unique and easy to identify. The sample is entitled “Android Tip Calculator”, as seen in Figure 2.16. There are three tabs with options to configure, the first allowing the selection of the project and the first Activity in the project to launch.

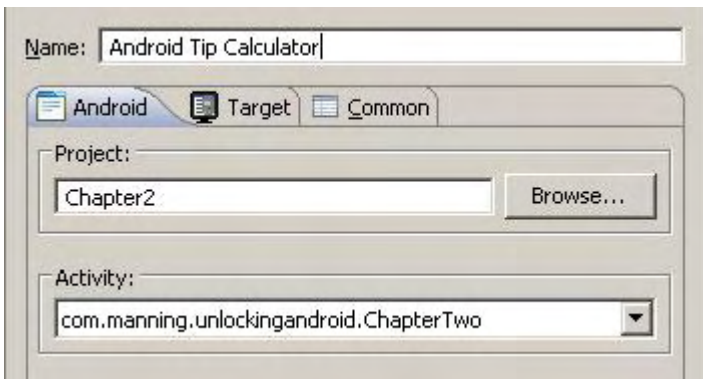


Figure 2.16. Setting up the Android Emulator launch configuration.

The next tab permits the selection of the desired “skin” which includes the screen layout, the network speed and the network latency. In addition, any command line parameters desired can be passed through to the emulator, as shown in Figure 2.17. When writing Android applications, keep in mind that the application may be run on different size screens, as not all devices will have the same physical characteristics. This setting in the Android Emulator launch configuration is a great way to test an application’s handling of different screen sizes and layouts.

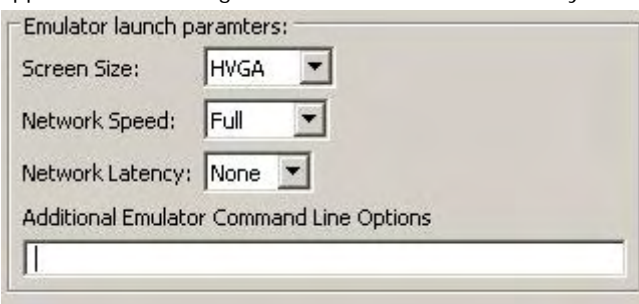


Figure 2.17. Selecting the operating characteristics of the Android Emulator

The third tab permits us to put this configuration on the favorites menu in the Eclipse IDE for easy access, as seen in Figure 2.18. We can select Run and/or Debug. Let’s make both selections, as it makes for easier launching when we want to test or debug the application.

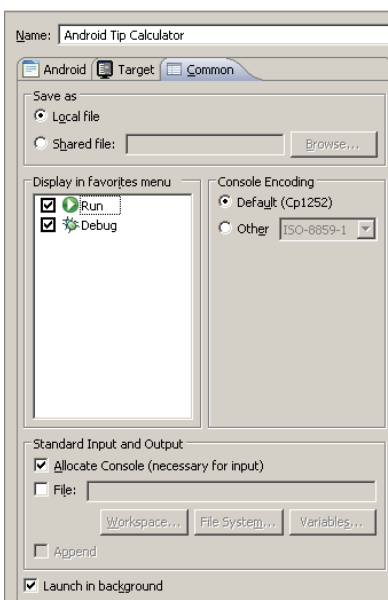


Figure 2.18. Adding this launch configuration to the toolbar menu

We're now ready to start the Android Emulator to test our Tip Calculator application, so we select our new Launch Configuration from the favorites menu as seen in Figure 2.19:

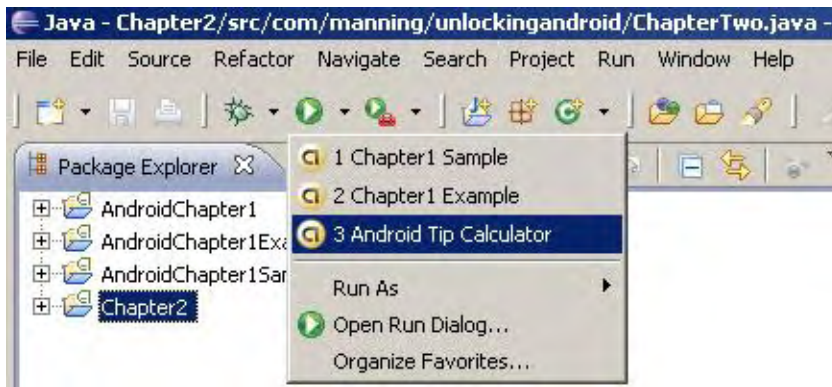


Figure 2.19. Starting this chapter's sample application, Android Tip Calculator

The Android Tip Calculator should now be running in the Android Emulator! Go ahead, test it out. But wait, what if there is a problem with the code but we're not sure where? It is time to have a brief look at debugging an Android application.

2.5 Debugging

Debugging an application is a skill no programmer can survive without, and fortunately, it is a straight-forward task to debug an Android application under Eclipse. The first step to take is to switch to the Debug Perspective in the Eclipse IDE. Remember, switching from one perspective to another takes place by using the Open Perspective menu found under the Window menu. Starting an Android application for debugging is just as simple as "Running" the application. Instead of selecting the application from the Favorites Run menu, use the Favorites Debug menu instead. This is the menu item with a picture of an insect (i.e. "bug"). Remember, when we setup the launch configuration, we added this configuration to both the Run and the Debug favorites menu.

The Debug Perspective gives us debugging capabilities similar to other development environments including the ability to single step into, or over, method calls and peer into values of variables. Breakpoints can be set by double-clicking in the left margin on the line of interest. Figure 2.20 demonstrates stepping through the Android Tip Calculator project and the resulting values showing up in the LogCat View. Note that full meal price, including tip, has not yet been displayed on the Android Emulator as that line has not yet been reached.

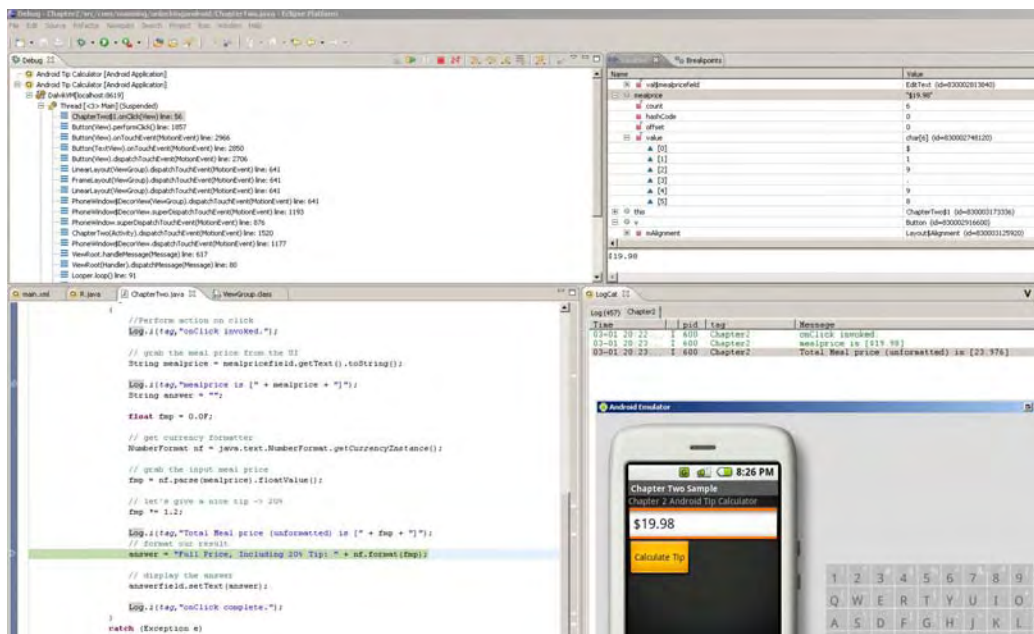


Figure 2.20. The Debug Perspective permits line by line stepping through an Android application.

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

2.6 Summary

This chapter introduced the Android SDK and a quick glance at the Android SDK's Java Packages in order to get familiar with the contents of the SDK from a class library perspective. The key development tools for Android application development including the Eclipse IDE and the Android Development Tools plug-in were introduced as well as some of the behind-the-scenes tools available in the SDK.

While building out the Android Tip Calculator, this chapter's sample application, we had the opportunity to navigate between the available perspectives in the Eclipse IDE. We used the Java Perspective to develop our application and both the DDMS Perspective and the Debug Perspective to interact with the Android Emulator while our application was running. A working knowledge of the Eclipse IDE's perspectives will be very helpful as you progress to build out the sample applications and study the development topics in the remainder of this book.

The Android Emulator and some of its fundamental permutations and characteristics were discussed. Due to the limited supply of real Android capable devices on the market, it is essential that the Android Emulator is a part of your toolbox today, however, even more than the "out of necessity" use of the Android Emulator, employing the Android Emulator is a good practice because of the benefits of using emulation for testing and validating mobile software applications.

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

In this chapter,

- Understanding Activities and Views
- Exploring Activity Life-Cycle
- Working with Resources
- Defining the `AndroidManifest.xml`

▪

With our introductory tour of the main components of the Android platform and development environment complete, we will now focus on some more depth with regard to user interface components. In this chapter we will look more closely at the fundamental Android concepts surrounding activities, views, and resources.

Activities are essential because, as we learned in chapter 1, they make up the screens of your application, and play a key role in the all important Android application life-cycle. Rather than allowing any one application to wrest control of the device away from the user, and other applications, Android introduces a well-defined life-cycle to swap in and out processes as needed. This means it is essential to understand not only how to start and stop an Android `Activity`, but also how to suspend and resume one. Activities themselves are made up of sub-components called views.

Views are what your users will see and interact with. Views handle layout, provide text elements for labels and feedback, provide buttons and forms for user input, and draw graphics to the screen. Views are also used to register interface event listeners, such as those for touch screen controls. A hierarchical collection of views is used to “compose” an `Activity`. You are the conductor, an `Activity` is your symphony, and `View` objects are your musicians.

Musicians need instruments, so we will stretch this analogy just a bit further to bring Android resources into the mix. Views, and other Android components, make use of strings, colors, styles, graphics, and the like, which are compiled into a binary form and made available to applications as resources. A dynamically generated class, `R.java`, which you will use heavily as you develop Android applications, is the bridge between Android binary references and source resources. This class is used, for example, to grab a string or a color, and add it to a `View`. The relationship between activities, views, and resources is depicted in figure 3.1.

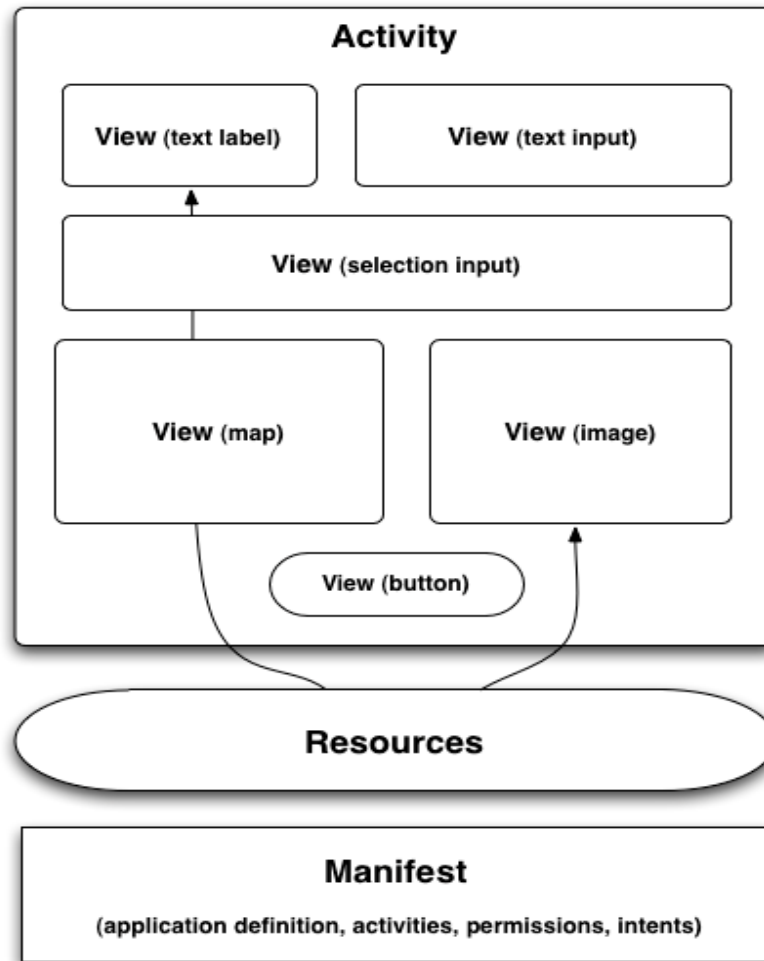


Figure 3.1. High level diagram of Activity, View, resources, and manifest relationship showing that activities are made up of views, and views use resources.. TODO update diagram (use bridge analogy)

Along with the components you use to build an application, views, resources, and activities, Android also includes a file that describes your application and provides metadata. This file, `AndroidManifest.xml`, which we also first met in chapter 1, is an XML (Extensible Markup Language) file that provides references for the various parts of your application to the platform. Where your application begins, what its permissions are, and what activities it includes, are all defined (along with other information that we will come to) in the manifest. The manifest is the one stop shop for the platform to boot and manage your application.

Overall, if you have done any development involving user interfaces of any kind on any platform, the concepts activities, views, and resources represent may be somewhat familiar or intuitive, at least on a fundamental level. The way these concepts are implemented in Android is, nevertheless, somewhat unique – and this is where we hope to shed some light in this chapter. Here we will be introducing a sample application that we will use to walk through these items, beginning with getting past the theory and into the code, with a form input screen, our first “Activity.”

3.1 Introducing The Activity

Over the course of this chapter and the next we will be building a sample application that allows the user to search for restaurant reviews based on location and cuisine. From there, this application will also allow the user to optionally call, visit the web site of, or map directions to, a selected restaurant. We chose this application as a starting point because it has a very clear and simple use case, and because it involves many different parts of the Android platform, and that allows us to cover a lot of ground fast – as well as, we hope, having the side benefit of being actually useful on your phone!

To create this application we will need 3 basic screens to begin with:

- A criteria screen where a user enters some parameters to search for restaurant reviews.
- A list of reviews screen that shows paged results that match the specified criteria.
- A detail page that shows the review details.

When complete our “RestaurantFinder” application will have the three screens (activities) shown in figure 1.2.



Figure 3.2 RestaurantFinder application screen shots, showing three Activities: **ReviewCriteria**, **ReviewList**, and **ReviewDetail**.

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

3.1.1 Creating an Activity class

Recall that an Activity is basically analogous to a screen. So to create a screen we will be extending the `android.app.Activity` base class and implementing several key methods it defines. Listing 3.1 shows the first portion of the RestaurantFinder ReviewCriteria class.

Listing 3.1 The first half of the ReviewCriteria Activity class


```

public class ReviewCriteria extends Activity {    #1

    private static final int MENU_GET_REVIEWS = Menu.FIRST;

    private EditText location;    #2
    private Spinner cuisine;    #2
    private Button getReviews;    #2

    @Override
    public void onCreate(final Bundle savedInstanceState) {    #3
        super.onCreate(savedInstanceState);

        this setContentView(R.layout.review_criteria);    #4

        this.location = (EditText) this.findViewById(R.id.location);    #5
        this.cuisine = (Spinner) this.findViewById(R.id.cuisine);    #5
        this.getReviews = (Button) this.findViewById(R.id.get_reviews_button);    #5

        ArrayAdapter<String> cuisines =
            new ArrayAdapter<String>(this, R.layout.spinner_view,
                this.getResources().getStringArray(R.array.cuisines));    #6
        cuisines.setDropDownViewResource(R.layout.spinner_view_dropdown);    #7
        this.cuisine.setAdapter(cuisines);    #8

        getReviews.setOnClickListener(new OnClickListener() {    #9
            public void onClick(View v) {
                ReviewCriteria.this.handleGetReviews();
            }
        });
    }
}

```

1. Extend android.app.Activity
2. Define Views to use within Activity
3. Override the Activity onCreate() method
4. Define the layout with setContentView
5. Inflate views from XML
6. Define ArrayAdapter instances to bind data, with Context, View, and Array
7. Set the View for the dropdown
8. Associate View and Data using Adapter
9. Add an OnClickListener for our Button

The `ReviewCriteria` class extends `android.app.Activity` #1, which does a number of very important things: first, it gives our application a “context,” because `Activity` itself extends `android.app.ApplicationContext`; second it brings the Android life-cycle methods into play; third it gives the framework a hook to start and run your application; and lastly it provides a container into which `View` elements can be placed.

Because an `Activity` represents an interaction with the user, in the form of a screen, it needs to provide components on the screen. This is where views come into play. In our `ReviewCriteria` class we have referenced three views in the code: `location`, `rating`, and `getReviews` #2. `location` is a type of `View` known as an `EditText`, a basic text entry component. Next, `rating` is a fancy select list component, known in Android terms as a `Spinner`. And, `getReviews` is a `Button`.

View elements such as these are placed within an `Activity` using a particular layout to create a screen. Layout and views can be defined directly in code, or in a layout XML

“resource” file. We will learn more about views as we progress through this section, and we will focus specifically on layout in section TODO-XXX.

Location as an EditText View

You may be wondering why we are using an EditText View for the “location” field in the ReviewCriteria Activity when Android includes technology that could be used to derive this value from the current physical location of the device (or allow the user to select it using a Map, rather than type it in). Good eye, but we are doing this intentionally here. We want this early example to be complete, and non trivial, but not too complicated. We will learn more about using the location support Android provides, and MapViews, in later chapters.

After an `Activity`, complete with necessary views, is started, then the life-cycle takes over and the `onCreate()` method is invoked **#3**. This is one of a series of important life-cycle methods the `Activity` class provides access to. Every `Activity` will override `onCreate()`, where component initialization steps are invoked, though not every `Activity` will need to override other life-cycle methods. The `Activity` life-cycle is worthy of an in depth discussion of its own, and for that reason we will explore these methods further, in section 3.1.2.

Once inside the `onCreate()` method, the `setContentView()` method is where you will normally associate an XML layout view file **#4**. We say normally, because you do not have to use an XML file at all, you can instead define all of your layout and `View` configuration in code, as Java objects. Nevertheless, it is often easier, and better practice by de-coupling, to use an XML layout resource for each `Activity`. An XML layout file defines `View` objects, which are laid out in a tree, and can then be “set” into the `Activity` for use.

Once again, we will come back to layout and views, both defined directly in code, and in XML, in later sections of this chapter. Here we simply need to stress that views are typically defined in XML, and then are set into the `Activity` and “inflated.” Views that need some run time manipulation, such as binding to data, can then be referenced in code and cast to their respective sub-types **#5**. Views that are static, those you don't need to interact with or update at run time, like labels, do not need to be referenced in code at all (they show up on the screen, because they are part of the `View` tree as defined in the XML, but need no explicit setup in code). Going back to the screen shots in Figure 3.1, you will notice that the `ReviewCriteria` screen has two labels, as well as the three inputs we have already discussed. These labels are not present in the code, they are defined in the “review_criteria.xml” file that we will see coming up when we discuss XML defined resources.

The next area of our `ReviewCriteria` Activity is where we bind data to our select list views, the `Spinner` objects. Android employs a handy “adapter” concept to link views that contain collections with data. Basically an `Adapter` is a collection handler that returns each item in the collection as a `View`. Android provides many basic adapters: `ListAdapter`, `ArrayAdapter`, `GalleryAdapter`, `CursorAdapter`, and more. And, you can also easily create your own, a technique we will use when we discuss creating custom views in section 3.2. Here we are using an `ArrayAdapter` that is populated with our `Context` (`this`), a `View` element defined in an XML resource file, and an array representing the data (also defined as a resource in XML – again, we will learn more about resources defined in XML in section 3.3) **#6**. When we create the `ArrayAdapter` we define the `View`

to be used for the element shown in the `Spinner` before it is selected, after it is selected it uses the `View` defined in the “dropdown” #7. After our `Adapter`, and its `View` elements, are defined, we then set it into the `Spinner` object #8.

The last thing this initial Activity demonstrates is our first explicit use of event handling. UI elements in general support many times of events, which we will learn more about in section TODO XXX. In this case we are using an `OnClickListener` with our `Button`, in order to respond when the button is clicked #9.

After the `onCreate()` method is complete, with the binding of data to our `Spinner` views, we next have some menu buttons (which are different than on screen `Button` views, as we shall see) and associated actions, we see how these are implemented in the last part of `ReviewCriteria` in listing 3.2.

Listing 3.2 The second half of the `ReviewCriteria` Activity class

```

. . .

@Override
public boolean onCreateOptionsMenu(final Menu menu) {           #1
    super.onCreateOptionsMenu(menu);
    menu.add(0, ReviewCriteria.MENU_GET_REVIEWS, 0, R.string.menu_get_reviews).
        setIcon(android.R.drawable.ic_menu_more);
    return true;
}

@Override
public boolean onOptionsItemSelected(final int featureId, final MenuItem item) {   #2
    switch (item.getItemId()) {
        case MENU_GET_REVIEWS:
            this.handleGetReviews();
            return true;
    }
    return super.onOptionsItemSelected(featureId, item);
}

private void handleGetReviews() {           #3
    if (!this.validate()) {
        return;
    }

    RestaurantFinderApplication application = (RestaurantFinderApplication)
this.getApplication(); #4
    application.setReviewCriteriaCuisine(this.cuisine.getSelectedItem().toString());
    application.setReviewCriteriaLocation(this.location.getText().toString());

    Intent intent = new Intent(Constants.INTENT_ACTION_VIEW_LIST); #5
    this.startActivity(intent);             #6
}

private boolean validate() {
    boolean valid = true;
    StringBuffer validationText = new StringBuffer();
    if ((this.location.getText() == null) ||
this.location.getText().toString().equals("")) {

validationText.append(this.getResources().getString(R.string.location_not_supplied_message));
        valid = false;
    }
    if (!valid) {
        new AlertDialog.Builder(this).setTitle(this.getResources()
            .getString(R.string.alert_label)).setMessage(
            validationText.toString()).setPositiveButton("Continue",
new android.content.DialogInterface.OnClickListener() {           #8

```

```

        public void onClick(final DialogInterface dialog, final int arg1) {
            }
        }).show();
        validationText = null;
    }
    return valid;
}
}
}

```

1. Create options menu
2. Respond when menu item is selected
3. Create method to process getting reviews and moving to next screen
4. Use the `android.app.Application` object for state
5. Create an Intent
6. Start an Activity
7. Use an AlertDialog – again notice builder pattern
8. Respond to button click with anonymous ClickListener

The menu items seen at the bottom of the `Activity` screens in figure 3.2 are all created using the `onCreateOptionsMenu()` method **#1**. Here we are using the `Menu` class `add()` method to create a single `MenuItem` element **#1**. We are passing a group ID, an ID, order, and a text resource reference to create the menu item. We are also then assigning an icon to the menu item an icon with the `setIcon` method. The text and the image are externalized from the code, again using Android's concept of resources. The `MenuItem` we have added duplicates the on screen `Button` with the same label for the "Get reviews" purpose.

Using the Menu versus on screen Buttons

We have chosen to use the `Menu` here, in addition to the on screen `Button` Views. Though either (or both) can work in many scenarios, you need to consider whether the menu, which is invoked by pressing the menu button on the device and then tapping a selection (button and a tap) is appropriate for what you are doing, or if an on screen `Button` (single tap) is a better choice. Generally on screen `Button`'s should be tied to UI elements (a search button for a search form input, for example), and menu items should be used for screen wide actions (submitting a form, performing an action like "create," "save," "edit," or "delete"). Yet, because all rules need an exception, if you have the screen real estate, it may be more convenient for users to have on screen buttons for actions as well (as we have done here). The most important thing to keep in mind with these type UI decisions is to be consistent. If you do it one way on one screen, use that same approach on other screens.

In addition to creating the menu item, we also add support to react and perform an action when the item is selected. This is done in the `onOptionsItemSelected()` event method **#2**, where we parse the ID of the multiple possible menu items with a `case/switch` statement. When the `MENU_GET_REVIEWS` item is determined to have been selected, then we then call the `handleGetReviews` method **#3**. This method puts the logic to save the users selection state in the `Application` object **#4**, and then sets up to call the next screen. We have moved this logic into its own method because we are using it from multiple places, from our on screen `Button`, and again from our `MenuItem`.

The `Application` object is used internally by Android for many purposes, and can be extended, as we have done with `RestaurantFinderApplication` (which simply includes few member variables in JavaBean style), to store global state information. We will reference this object again on other activities to retrieve the information we are storing

here. There are several ways to pass objects back and forth between activities, using `Application` is one of them. You can also use public static members, and `Intent` extras with `Bundle` objects. Additionally, you can use the provided SQLite database, or you can implement your own `ContentProvider` and store data there. We will cover more about state, and data persistence in general, including all these concepts, in chapter 5. The important thing to take away here is that at this point we are using the `Application` object to pass state between activities.

After we store the criteria state we then fire off an action in the form of an Android `Intent` #5. We touched on intents in chapter 1, and we will delve into them further in the next chapter, but basically we are asking another `Activity` to respond to the user's selection of a menu item by calling `startActivity(Intent intent)` #6.

Using `startActivity` versus `startActivityForResult`

The most common way to invoke an `Activity` is by using the simple `startActivity()` method, but there is also another method you will see used in specific instances – `startActivityForResult()`. Both of these pass control to a different `Activity`. The difference with regard to “forResult” is that it returns a value to the current `Activity` when the `Activity` being invoked is complete. It in effect allows you to chain `Activities` and expect callback responses (you get the response by implementing the `onActivityResult()` method).

Also of note within the `ReviewCriteria` example is that we are using an `AlertDialog` #7. Before we allow the next `Activity` to be invoked, we call a simple `validate()` method that we have created, where we display a pop-up style alert dialog to the user if the location has not been specified. Along with generally demonstrating the use of `AlertDialog`, this also again demonstrates how a button can be made to respond to a click event with an `OnClickListener()` #8.

The Builder Pattern

You may have noticed the usage of the Builder pattern when we added parameters to the `AlertDialog` we created in the `ReviewCriteria` class. If you are not familiar with this approach, basically each of the methods invoked, such as `AlertDialog.setMessage()`, and `AlertDialog.setTitle()` returns a reference to itself (`this`), which means we can continue chaining method calls. This avoids either an extra long constructor with many parameters, or the repetition of the class reference throughout the code. Intents make use of this handy pattern too, it is something you will seem time and time again in Android.

Though we made several references here to sections where we will talk more about underlying details later, we still covered a good deal of material and have completed our first `Activity` – `ReviewCriteria`! With this class now fully implemented, we next need to take a much closer look at the all-important Android `Activity` life-cycle and how it relates to processes on the platform.

3.1.2 Exploring Activity Life-Cycle

Every process on a running Android platform is placed on a stack. When you use an `Activity` in the foreground, the process that hosts that activity is placed at the top of the stack, and the previous process (the one hosting whatever `Activity` was previously in the foreground) is moved down one notch. This is a key point to understand. Android tries to keep processes running as long as it can, but it can't keep every process running forever, system memory is after all finite. So what happens when memory starts to run low?

UNDERSTANDING HOW PROCESSES AND ACTIVITIES RELATE

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

1. The process hosting the foreground `Activity` is the most important.
2. Any process hosting a visible but not foreground `Activity` is next in line.
3. Any process hosting a background `Activity` is next in line.
4. Any process not hosting any `Activity` (or `Service` or `BroadcastReceiver`), known as an "empty" process, is last in line.

Due to the fact that a user can elect to change directions at just about any time – make a phone call, change the screen orientation, respond to an SMS message, just decide to stop using your wonderful stock market analysis application and start playing Android Poker – all `Activity` classes have to be able to handle being stopped and shut down at any time. If the process your `Activity` is in falls out of the foreground, it is eligible to be killed (it's not up to you, it's up to the user, and the platform).

To manage this environment, Android applications, and the `Activity` classes they host, have to be designed a bit differently than what you may be used to. Using a series of event related callback type methods the `Activity` class defines, you can setup and tear down state gracefully. The `Activity` sub-classes that you implement (as we saw a bit of with `ReviewCriteria` in the previous section) override a set of "life-cycle" methods to make this happen. As we discussed in section 3.1.1, every `Activity` has to implement the `onCreate()` method. This is the starting point of the life-cycle. In addition to `onCreate()`, most activities will also want to implement the `onPause()` method, where data and state can be persisted before the hosting process potentially falls out of scope.

The life-cycle methods that the `Activity` class provides are all called in a specific order by the platform as it decides to create and kill processes. Because you, as an application developer, cannot control the actual processes, you have to rely on your use of the callback life-cycle methods to control state in your `Activity` classes as they come into the foreground, and as they move into the background, and fall away altogether. This is a very significant, and clever, part of the overall Android platform. As the user makes choices, activities are created and paused in a defined order, by the system as it starts and stops processes.

ACTIVITY LIFE-CYCLE

Beyond `onCreate()` and `onPause()`, Android also provides other distinct stages, each of which is a part of a particular "phase" of the life of an `Activity` class. All of the methods, and the phases, for each part of the life-cycle are shown in figure 3.3.

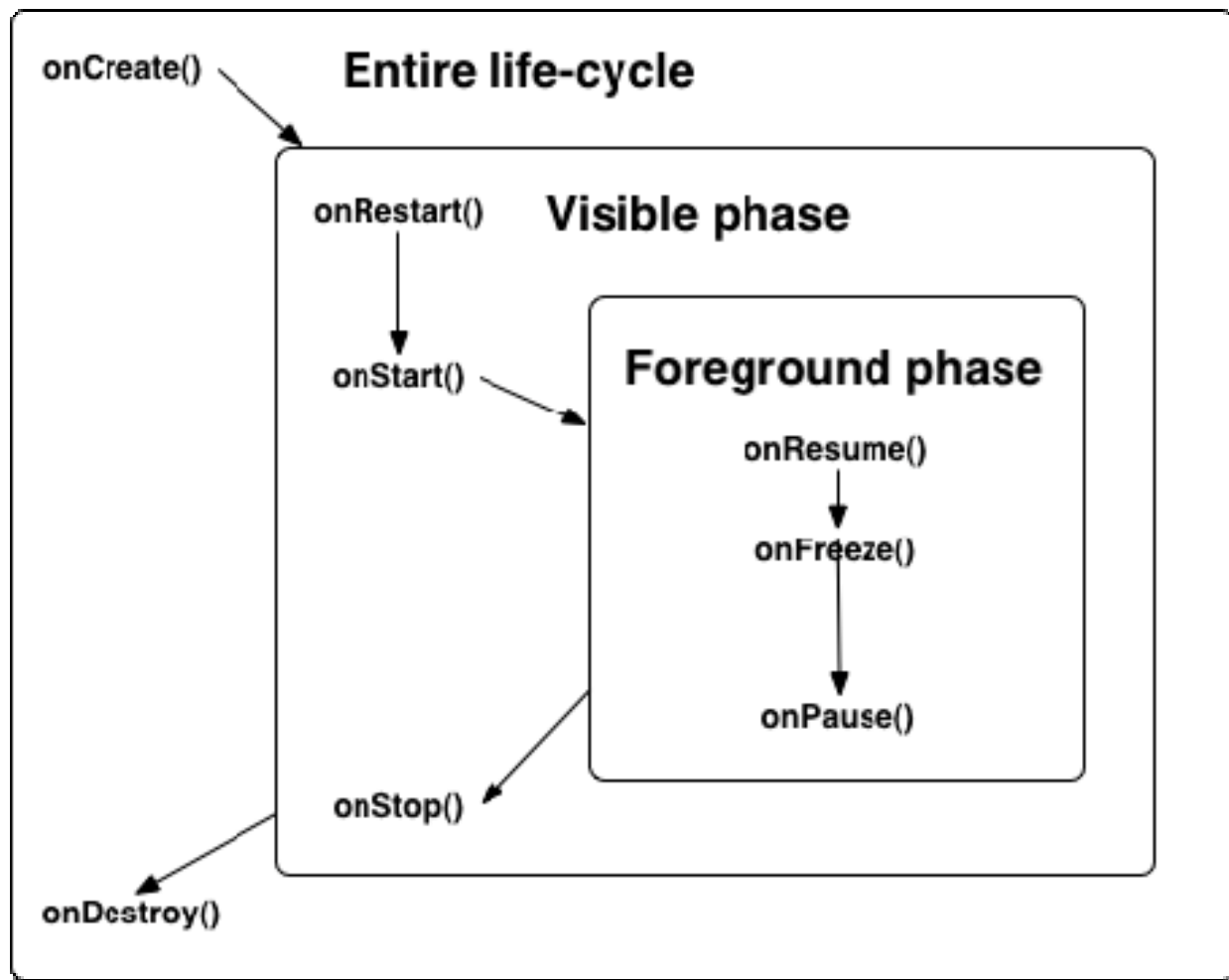


Figure 3.3 Android Activity life-cycle diagram, showing the methods involved in the foreground and background phases.

Each of the life-cycle methods Android provides has a distinct purpose, and each happens during part of either the foreground, visible, or entire life-cycle phases.

- The foreground phase refers to when the `Activity` is viewable on the screen and on top of everything else (when the user is interacting with the `Activity` to perform a task).
- The visible phase refers to when the `Activity` is on the screen, but may not be on top and interacting with the user (when a dialog or floating window is on top of the `Activity`, for example).
- And, the entire-life cycle phase refers to the methods that may be called when the application is not on the screen at all, before it is created, and after it is gone prior to being shut down.

Table 3.1 provides further information about the life-cycle phases and outlines each of the related methods on the `Activity` class.

Table 3.1 Android Activity life-cycle methods and purpose.

Method	Purpose
<code>onCreate()</code>	Called when the Activity is created. Setup is done here, Also provided is access to any previously stored state in the form of a <code>Bundle</code> .
<code>onRestart()</code>	Called if the Activity is being restarted, if it is still in the stack, rather than starting new.
<code>onStart()</code>	Called when the Activity is “becoming” visible on the screen to the user.
<code>onResume()</code>	Called when the Activity starts interacting with the user (note this is always called, whether starting or restarting).
<code>onPause()</code>	Called when the Activity is pausing, reclaiming CPU and other resources. This method is where state should be saved so that when an Activity is restarted it can start from the same state it had when it quit.
<code>onStop()</code>	Called after Activity is no longer visible to user.
<code>onDestroy()</code>	Called when an Activity is being completely removed from system memory. Happens either because “ <code>onFinish()</code> ” is directly invoked, or the system decides to stop the Activity to free up resources.

In terms of life-cycle methods it is very important to be aware that `onPause()` is the last opportunity you have to clean up and save state information. The processes that host your `Activity` classes will not be killed by the platform until after the `onPause()` method has completed, but may be killed thereafter. This means, the system will attempt to run through all of the life-cycle methods every time, but if resources are spiraling out of control (as determined by the platform) a fire alarm may be sounded and the processes that are hosting activities that are beyond the `onPause()` method may be killed at any point. Any time your `Activity` is moved to the background, `onPause()` is called, but before your `Activity` is completely removed `onDestroy()` is not guaranteed to have been called (it probably will be called, under normal circumstances, but not always).

The `onPause()` method is therefore paramount. That is where you need to save persistent state. Whether that persistent state is specific to your application, such as user preferences, or global shared information, such as the contacts database, `onPause()` is where you need to make sure all the loose ends are tied up – every time. We will discuss how to save data in Chapter 5, but here the important thing is to know when and where that needs to happen.

In addition to persistent state there is one more aspect you should also be familiar with, and that is “instance state.” Instance state refers to the state of the user interface itself. The `onSaveInstanceState()` `Activity` method is called when an `Activity` may be destroyed, so that at a future time the interface state can be restored. This method is used by the platform to handle the view state processing in the vast majority of cases. This means you normally don't have to mess with it. Nevertheless, it is important to know that it is there, and that the `Bundle` it saves is what is handed back to the `onCreate()` method when an `Activity` is restored. If you need to customize the view

state, you can, by overriding this method, but don't confuse this with the larger life-cycle methods.

Managing activities with life-cycle events in this way, through parent processes the platform controls, allows Android to do the heavy lifting, deciding when things come into and out of scope, relieving applications of the burden themselves, and ensuring a level playing field. This is a key aspect of the platform that varies somewhat from many other application development environments. In order to build robust and responsive Android applications you have to pay careful attention to the life-cycle.

Now that we have some background in place concerning the `Activity` life-cycle, and have created our first screen, we will next further investigate views and fill in some more detail.

3.2 Working with Views

Though it is a bit cliché, it is true that views are the “building blocks” of the user interface of an Android application. Activities, as we have seen, contain views, and `View` objects represent elements on the screen and are responsible for interacting with users through events.

Every Android screen that you can see contains a hierarchical tree of `View` elements on it. These views come in a variety of shapes and sizes. Many of the views you will need on a day to day basis are provided for you as part of the platform – basic text elements, input elements, images, buttons, and the like. In addition, you can create your own composite and or custom views when the need arises. Views can be placed into an `Activity` (and thus on the screen) either directly in code, or through the use of an XML resource that is later “inflated” at runtime.

In this section we will discuss many of the fundamental aspects of views: the common views that Android provides, custom views that can be created as needed, layout in relation to views, and event handling. We specifically won't address views defined in XML here though, because that will be covered explicitly in section 3.3 as part of a larger resources section. Here we begin with the common `View` elements Android provides by taking a short tour of the API.

3.2.2 Exploring Common Views

Android provides a healthy set of `View` objects in the `android.view` package. These objects range from familiar constructs like the `EditText`, `Spinner`, and `TextView` that we have already seen in action, to more specialized widgets such as `AnalogClock`, `Gallery`, `DatePicker`, `TimePicker`, and `VideoView` – to name a few. For a quick glance at some of the more eye-catching views check out the sample page in the Android documentation: <http://code.google.com/android/reference/view-gallery.html>.

For a high level snapshot of what the overall `View` API looks like you can refer to the class diagram in figure 3.4. This diagram shows how the specializations fan out and includes many, but not all, of the `View` derived classes.

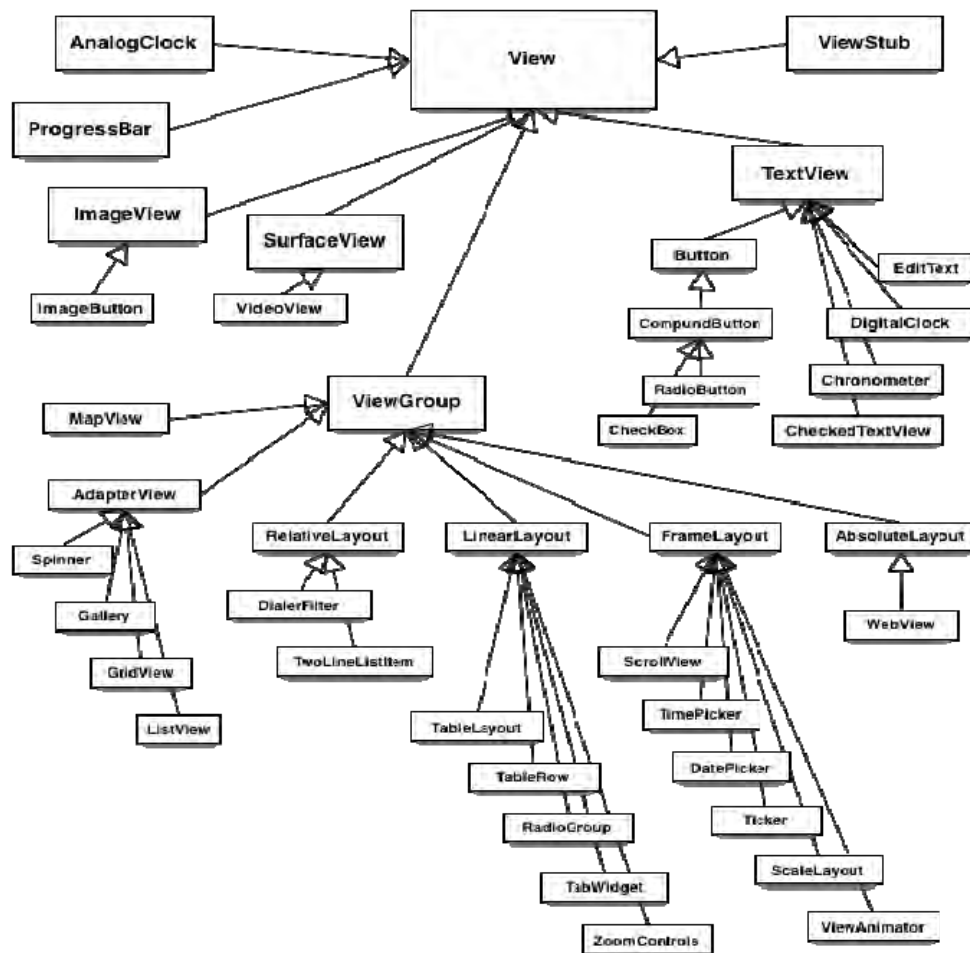


Figure 3.4 A Class diagram of the Android View API, showing the root View class, and specializations from there, notice that ViewGroup classes, such as layouts, are also a type of View.

As is evident from the diagram in figure 3.4 (which is not comprehensive) the `View` API has quite a few classes. Importantly you should note that `ViewGroup` is itself a subclass of `View`, as are other elements such as `TextView`. Everything is ultimately a `View`, even the layout classes (which extend `ViewGroup`).

Of course, everything that extends `View` has access to the base class methods. These methods allow you to perform a variety of important UI related operations, such as setting the background, setting the minimum height and width, setting padding, setting and enabling events (such as clickable and focusable), setting layout parameters, and more. Table 3.2 includes an example of some of the methods available on the root `View` class.

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

Method	Purpose
--------	---------

<code>setBackgroundColor(int color)</code>	Set the background color.
<code>setBackgroundDrawable(Drawable d)</code>	Set the background drawable (image)
<code>setMinimumHeight(int minHeight)</code>	Set the minimum height (parent may override).
<code>setMinimumWidth(int minWidth)</code>	Set the minimum width (parent may override).
<code>setPadding(int left, int right, int top, int bottom)</code>	Set the padding.
<code>setClickable(boolean c)</code>	Set whether or not element is clickable.
<code>setFocusable(boolean f)</code>	Set whether or not element is focusable.
<code>setOnClickListener(OnClickListener l)</code>	Set listener to fire when click event occurs.
<code>setOnFocusChangeListener(OnFocusChangeListener l)</code>	Set listener to fire when focus event occurs.
<code>setLayoutParams(ViewGroup.LayoutParams l)</code>	Set the LayoutParams (position, size, and more).

Beyond the base class each `View` subclass also typically also adds a host of refined methods to further manipulate its respective state, such as what is shown for `TextView` in Table 3.3.

Table 3.3 Further View methods for the TextView subclass.

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

Using the combination of base class methods, with the subtype methods, you can see that you can set layout, padding, focus, events, gravity, height, width, colors, and basically everything you might need. Using these methods in code, or their counterpart attributes in the `android:` namespace in XML, when defining views in XML (something we will see done in section 3.3), is how you manipulate a `View` element. Each `View` element you use has its own path through the API and therefore particular set of methods available, for details on all the methods see the Android JavaDocs: <http://code.google.com/android/reference/android/view/View.html>.

When you couple the wide array of classes, with the rich set of methods available from the base `View` class on down, the Android View API can quickly seem intimidating. Thankfully, despite the initial impression, many of the concepts involved fast become evident and usage becomes more intuitive as you move from `View` to `View` (because they all are specializations on the same object at the core). So, even though the "747

cockpit" analogy could be applied, once you start working with Android you should be able to earn your wings fairly quickly.

Though our RestaurantFinder application will not use many of the views listed in our whirlwind tour here, these are still useful to know about, and many of them will be used in later examples throughout the book. The next thing we will focus on is a bit more detail concerning one of the most common non trivial `View` elements, specifically the `ListView` component.

3.2.2 Using a *ListView*

On the `ReviewList` Activity of the RestaurantFinder application, shown in figure 3.2, we see a different type of `View` than the simple user inputs and labels we have used up to this point – this screen presents a scrollable list of choices for the user to choose from.

This Activity is using a `ListView` component to display a list of review data that is obtained from calling the Google Base Atom API using HTTP (we will refer to this as a “web service,” even though it is not technically SOAP or any other formal standard). After we make the HTTP call, by appending the user's criteria to the required Google Base URL, we will then parse the results with the Simple API for XML (SAX) and create a `List` of reviews. The details of this parsing will be covered in chapter 5, when we discuss using the network as a data source on another sample application. The resulting `List` will be used to populate our screen's list of items to choose from – and that is our focus here.

The code in listing 3.3 shows how we create and use a `ListView` to represent this list of items to choose from on an Activity screen.

Listing 3.3 The first half of the `ReviewList` Activity class, showing the usage of a `ListView`

```
public class ReviewList extends ListActivity {    #1

    private static final int NUM_RESULTS_PER_PAGE = 8;
    private static final int MENU_GET_NEXT_PAGE = Menu.FIRST;
    private static final int MENU_CHANGE_CRITERIA = Menu.FIRST + 1;
    private ProgressDialog progressDialog;
    private ReviewAdapter reviewAdapter;    #2
    private List<Review> reviews;    #3
    private TextView empty;

    private final Handler handler = new Handler() {    #4
        @Override
        public void handleMessage(final Message msg) {
            ReviewList.this.progressDialog.dismiss();
            if ((reviews == null) || (reviews.size() == 0)) {
                ReviewList.this.empty.setText("No Data");
            } else {
                ReviewList.this.reviewAdapter = new ReviewAdapter(ReviewList.this, reviews);
                ReviewList.this.setListAdapter(reviewAdapter);
            }
        }
    };

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

        this setContentView(R.layout.review_list);    #5

        this.empty = (TextView) findViewById(R.id.empty);    #6

        final ListView listView = this.getListView();    #7
        listView.setItemsCanFocus(false);    #7
    }
}
```

```

        listView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);    #|7
        listView.setEmptyView(this.emptyView);                  #|7
    }

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

        RestaurantFinderApplication application = (RestaurantFinderApplication)
this.getApplication();    #8
        String criteriaCuisine = application.getReviewCriteriaCuisine();
        String criteriaLocation = application.getReviewCriteriaLocation();

        int startFrom = this.getIntent().getIntExtra(Constants.STARTFROM_EXTRA, 1);    #9

        this.loadReviews(criteriaLocation, criteriaCuisine, "ALL", startFrom);    #10
    }

    // onCreateOptionsMenu omitted for brevity
    . . .

```

1. Extend a ListActivity
2. Use a ReviewAdapter adapter
3. Back the Adapter with a List of Review objects
4. Include a Handler to respond to Messages and update UI
5. Use a layout defined in resources
6. Define a TextView to use when the list is empty
7. Set specific properties for the ListView
8. Again use the Application for global state
9. Use an Intent extra for simple properties
10. Load review data

The ReviewList Activity extends ListActivity **#1**, which is used to host a ListView. The default layout of a ListActivity is a full screen, centered, list of choices for the user to select from. A ListView is similar in concept to a Spinner, in fact they are both subclasses of AdapterView as we saw in the class diagram in figure 3.4. This means that ListView, like Spinner, also uses an Adapter to bind to data. In this case we are using a custom ReviewAdapter class **#2**. We will learn more about ReviewAdapter in the next section, when we discuss custom views. The important part here is that we are using an Adapter for our ListView (even though it's a custom adapter), and we use a List of Review objects to populate the Adapter **#3**.

Because we don't yet have the data to populate the list, which we will get from a web service call in another Thread, we need to include a Handler to allow for fetching data and updating the UI to occur in separate steps **#4**. Don't worry too much about these concepts here, as they will make more sense when we discuss them while looking at the second half of ReviewList in listing 3.4 shortly.

After our ListView, and its data, are declared, we then move on to the typical onCreate() tasks we have already seen, including using a layout defined in a resources XML file **#5**. This is significant with respect to ListActivity because a ListView with the ID name "list" is required if you want to customize the layout, as we have done (the ID name is in the layout XML file, which we will see in section 3.3.3). If you don't provide a layout, you can still use ListActivity and ListView, you just get the system default. We are also defining an element that will be used to display the message "No Data" if our List backing our View has no elements **#6**. Finally, we also set several specific properties on the ListView, using its customization methods, such as whether or not the list items themselves are focusable, how many elements can be selected at a time, and what View to use when the list is empty **#7**.

After we setup the View elements needed on the Activity we then get the criteria to make our web service call from the Review object we placed in the Application from the ReviewCriteria Activity **#8**. Here we also use an Intent "extra" to store a primitive int for page number **#9**. We pass all the criteria data (criteriaLocation, criteriaCuisine, and startFrom) into the loadReviews() method **#9**, which eventually makes our web service call to get populate our data list. This method, and several other methods that show how we deal with items in the list being clicked on, are shown in the second half of the ReviewList class, in listing 3.4.

Listing 3.4 The second half of the ReviewList Activity class, showing a response to a list item being selected

```

@Override
public boolean onOptionsItemSelected(final int featureId, final MenuItem item) {           #1
    Intent intent = null;
    switch (item.getItemId()) {
    case MENU_GET_NEXT_PAGE:
        intent = new Intent(Constants.INTENT_ACTION_VIEW_LIST);
        intent.putExtra(Constants.STARTFROM_EXTRA,
            getIntent().getIntExtra(Constants.STARTFROM_EXTRA, 1) + NUM_RESULTS_PER_PAGE);
#2
        this.startActivity(intent);
        return true;
    case MENU_CHANGE_CRITERIA:
        intent = new Intent(this, ReviewCriteria.class);
        this.startActivity(intent);
        return true;
    }
    return super.onOptionsItemSelected(featureId, item);
}

@Override
protected void onListItemClick(final ListView l, final View v,                         #3
    final int position, final long id) {
    RestaurantFinderApplication application =
        (RestaurantFinderApplication) this.getApplication();                         #4
    application.setCurrentReview(reviews.get(position));                             #4

    Intent intent = new Intent(Constants.INTENT_ACTION_VIEW_DETAIL);                 #5
    intent.putExtra(Constants.STARTFROM_EXTRA,
        this.getIntent().getIntExtra(Constants.STARTFROM_EXTRA, 1));                 #5
    this.startActivity(intent);
}

private void loadReviews(final String location, final String cuisine,
    final String rating, final int startFrom) {                                     #6

    final ReviewFetcher rf = new ReviewFetcher(location, cuisine,
        rating, startFrom, NUM_RESULTS_PER_PAGE);                                   #7

    this.progressDialog = ProgressDialog.show(this, " Working...",
        " Retrieving reviews", true, false);                                       #8

    new Thread() {
        @Override
        public void run() {
            ReviewList.this.reviews = rf.getReviews();                             #9
            ReviewList.this.handler.sendMessage(0);                                #10
        }
    }.start();
}
}

```

1. Implement the menu handling onOptionsItemSelected method
2. If the menu selection is next page, increment the startFrom Intent extra value

3. Implement the `onListItemClick` method
4. Get the `Application` object and set the selected `Review` into it
5. Pass the `startFrom` extra value along through the `Intent` for the next screen
6. Create the `loadReviews` method
7. Instantiate a `ReviewFetcher` instance which will be used for the web service call
8. Show a `ProgressDialog` while the network call is taking place
9. Inside a separate `Thread` make the web service call
10. Update the handler when the call is complete

This `Activity` has a menu item which allows the user to get the next “page” of results, or change the list criteria. To support this we have to implement the `onMenuItemSelected` method **#1**. If the `MENU_GET_NEXT_PAGE`, menu item is selected **#2**, we then define a new intent to reload the screen with an incremented `startFrom` value (and we use the `getExtras()` and `putExtras()` intent methods to do this) **#2**.

Next we see a special `onListItemClick()` method **#3**. This method is used to respond when one of the list items in a `ListView` is clicked. Here we use the position of the clicked item to reference the particular `Review` item the user chose, and we set this into the `Application` for later usage in the `ReviewDetail` `Activity` (which we will begin to implement in section 3.3) **#4**. After we have the data set, we then call the next `Activity` (including the `startFrom` extra) **#5**.

Lastly in the `ReviewList` class we have the `loadReviews()` method which, strangely enough, loads reviews **#6**. This method is significant for several reasons. First it sets up the `ReviewFetcher` class instance, which will be used to call out to the Google Base API and return a `List` of `Review` objects **#7**. Then it invokes the `ProgressDialog.show()` method to show the user we are retrieving data **#8**. And, finally it sets up a new `Thread` **#9**, within which the `ReviewFetcher` is used, and the earlier `Handler` we saw in the first half of `ReviewList` is sent an empty message **#10**. If you refer back to when the `Handler` was established, in listing 3.3, you can see that is where, when the message is received, we dismiss the `ProgressDialog`, populate the `Adapter` our `ListView` is using, and call `setListAdapter()` to update the UI. The `setListAdapter()` method will iterate the `Adapter` it is handed and display a returned `View` for every item.

With the `Activity` created and setup, and the `Handler` being used to update the `Adapter` with data, we now have a second screen in our application. The next thing we need to do is fill in some of the gaps surrounding working with handlers and different threads. These concepts are not view specific, but are worth a small detour at this point because you will want to use these classes when trying to perform various tasks related to retrieving and manipulating data needed for the UI.

3.2.3 Multitasking with Handler and Message

The `Handler` is the swiss army knife of messaging and scheduling operations for Android. This class allows you to queue tasks to be run on different threads, and allows you schedule tasks using `Message` and `Runnable` objects.

The Android platform monitors the responsiveness of applications, and kills those that are considered non-responsive. An “Application Not Responding” (ANR) event is defined as no response to an user input for 5 seconds (a user touches the screen, or presses a key, etc. - your application must respond within 5 seconds). So does this mean your code always has to complete within 5 seconds? No, of course not, but the

main UI thread does have to **respond** within 5 seconds. To keep the main UI thread snappy, any long running tasks, such as retrieving data over the network, or getting a large amount of data from a database, or complicated calculations, should be performed in a separate thread.

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

In the `Handler` usage we have already seen in listings 3.3 and 3.4 we created a `Handler` with a no argument constructor. With this approach, the `Handler` is automatically associated with the `Looper` of the current running thread, typically the main UI thread. The main UI thread, which is created by the process of the running application, is an instance of a `HandlerThread`, which is basically an Android `Thread` specialization that provides a `Looper`. The key parts involved in this arrangement are depicted in the diagram in figure 3.5.



Figure 3.5 Usage of the Handler class with separate threads, and the relationship of HandlerThread, Looper, and MessageQueue.

When implementing a `Handler` you will have to provide a `handleMessage(Message m)` method. This method is the hook that lets you pass messages. When you create a new `Thread`, you can then call one of several `sendMessage` methods on `Handler` from within that thread's run method, as our examples and diagram demonstrate. Calling `sendMessage` puts your message on the `MessageQueue`, which the `Looper` maintains.

Along with sending messages into handlers, you can also send `Runnable` objects directly, and you can schedule things to be run at different times in the future. You “send” messages and “post” runnables. Each of these concepts support various methods such as `sendMessage()`, which we have already used, and the counterparts `sendMessageAtTime(int what, long time)` and `sendMessageDelayed(int what, long delay)`. Once in the queue your message is processed as soon as possible (unless you schedule or delay it using the respective send or post method).

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

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

3.2.4 Creating Custom Views

Though you can often get away with simply using the views that are provided with Android, there may also be situations, like the one we are now facing, where you need a custom view to display your own object in some unique way.

In the `ReviewList` screen we used an `Adapter` of type `ReviewAdapter` to back our `ListView`. This is a custom `Adapter` that within it contains a custom `View` object, `ReviewListView`. A `ReviewListView` is what our `ReviewList` Activity displays for every row of data it contains. The `Adapter` and `View` are shown in listing 3.5.

Listing 3.5 The `ReviewAdapter` and Inner `ReviewListView` classes which show a custom `Adapter` and `View`

```
public class ReviewAdapter extends BaseAdapter {           #1

    private final Context context;                         #|2
    private final List<Review> reviews;                    #|2

    public ReviewAdapter(final Context context, final List<Review> reviews) {
        this.context = context;
        this.reviews = reviews;
    }

    public int getCount() {                                #3
        return this.reviews.size();
    }
```

```

public Object getItem(final int position) { #3
    return this.reviews.get(position);
}

public long getItemId(final int position) { #3
    return position;
}

public View getView(final int position, final View convertView, final ViewGroup parent) {
#4
    Review review = this.reviews.get(position);
    return new ReviewListView(this.context, review.name, review.rating);
}

private final class ReviewListView extends LinearLayout {
    private final TextView name;
    private final TextView rating;

    public ReviewListView(final Context context,
        final String name, final String rating) { #5

        super(context);
        this.setOrientation(LinearLayout.VERTICAL);

        LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(
            ViewGroup.LayoutParams.WRAP_CONTENT,
            ViewGroup.LayoutParams.WRAP_CONTENT); #6
        params.setMargins(5, 3, 5, 0);

        this.name = new TextView(context); #7
        this.name.setText(name);
        this.name.setTextSize(16f);
        this.name.setTextColor(Color.WHITE);
        this.addView(this.name, params);

        this.rating = new TextView(context); #7
        this.rating.setText(rating);
        this.rating.setTextSize(16f);
        this.rating.setTextColor(Color.GRAY);
        this.addView(this.rating, params); #8
    }
}
}

```

1. Extend BaseAdapter (which itself implements Adapter)
2. Include Context and List<Review> as members
3. Implement the basic Adapter methods
4. Implement the Adapter getView method
5. Define a custom inner View class ReviewView
6. Set layout in code
7. Instantiate TextView members and set properties
8. Add TextView to the tree ReviewListView provides

The first thing to note in ReviewAdapter is that it extends BaseAdapter #1. BaseAdapter is an Adapter implementation that provides some basic event handling support. Adapter itself is an interface in the android.Widget package that provides a way to bind data to a View with some common methods. This is often used with collections of data, such as we saw with Spinner and ArrayAdapter in listing 3.1. Another common usage is with a CursorAdapter, which returns results from a database (something we will see in chapter 5). Here we are creating our own Adapter, because we want it to return a custom View.

Our ReviewAdapter class accepts two parameters in the constructor and sets those values to two simple member objects: Context and List<Review> #2. Then this class goes on to implement the straightforward

required `Adapter` interface methods that return a count, an item, and an ID (we just use the position in the collection as the ID) **#3**. The next `Adapter` method we have to implement is the important one, `getView()`. This is where the `Adapter` will return any `View` you create for a particular item in the collection of data it is supporting. Within this method we get a particular `Review` object based on the position/ID, and then we create an instance of a custom `ReviewListView` object to return as the `View` **#4**.

`ReviewListView` itself, which extends `LinearLayout` (something we will learn more about in section 3.2.4), is an inner class inside `ReviewAdapter` (since we will never use it outside of returning a view from `ReviewAdapter`) **#5**. Within it we see an example of setting layout and `View` details in code, rather than in XML. Here we set the orientation, parameters, and margin for our layout **#6**. Then we populate the simple `TextView` objects that will be children of our new `View` and represent data **#7**. Once these are setup via code when then add them to the parent container (in this case the parent is our custom class `ReviewListView`) **#8**. This is where the data binding happens – the bridge to the `View` from data. Another important thing to note about this is that we have created not only a custom `View`, but a composite one as well. That is, we are using simple existing `View` objects in a particular layout to construct a new type of re-usable `View` which shows the detail of a selected `Review` object on screen, as seen in figure 3.2.

Our `ReviewListView` object, while custom, is admittedly (and intentionally) fairly trivial. In many cases you will be able to create custom views by combining existing views in this manner. Nevertheless, you should also be aware that you can go deeper and extend the `View` class itself. Then you can implement core methods as needed. Using this approach you have access to the life-cycle methods of a `View` (not an `Activity` as we have already covered, but an individual `View`). These include `onMeasure()`, `onLayout()`, `onDraw()`, `onVisibilityChanged()`, and others. Though we don't need that level of control here, you should be aware that extending `View` gives you a great deal of power to create custom components.

Now that we have seen how we get the data for our reviews, and what the `Adapter` and custom `View` we are using look like, the last thing we need to do is implement our final `RestaurantFinder` `Activity`, the `ReviewDetail` screen. We will begin this in section 3.3. when we look at resources, but before we can get there we need to stop and take a closer look at a few more aspects of views, including layout.

3.2.5 Understanding Layout

One of the most significant aspects of creating your UI and designing your screens is understanding layout. In Android terms screen layout is defined in terms of `ViewGroup` and `LayoutParams` objects. `ViewGroup` is a `View` that contains other views (has children), and also defines and provides access to the layout.

On every screen all the views are placed in a hierarchical tree, so every element has “children,” and somewhere at the root is a `ViewGroup`. All the views on the screen support a host of attributes that pertain to background color, color, and so on. We touched on many of these attributes in section 3.2.2 when we discussed the methods on the `View` class. Dimensions though, width and height, and other properties such as relative or absolute placement, and margins, are based on the `LayoutParams` a view requests, and what the parent – based on its type, its own dimensions, and the dimensions of all of its children – can accommodate.

The main `ViewGroup` classes are shown in the class diagram we saw previously in figure 3.4. The diagram in figure 3.6 expands on this class structure to show the

specific `LayoutParams` inner classes of the view groups, and layout properties each type provides.

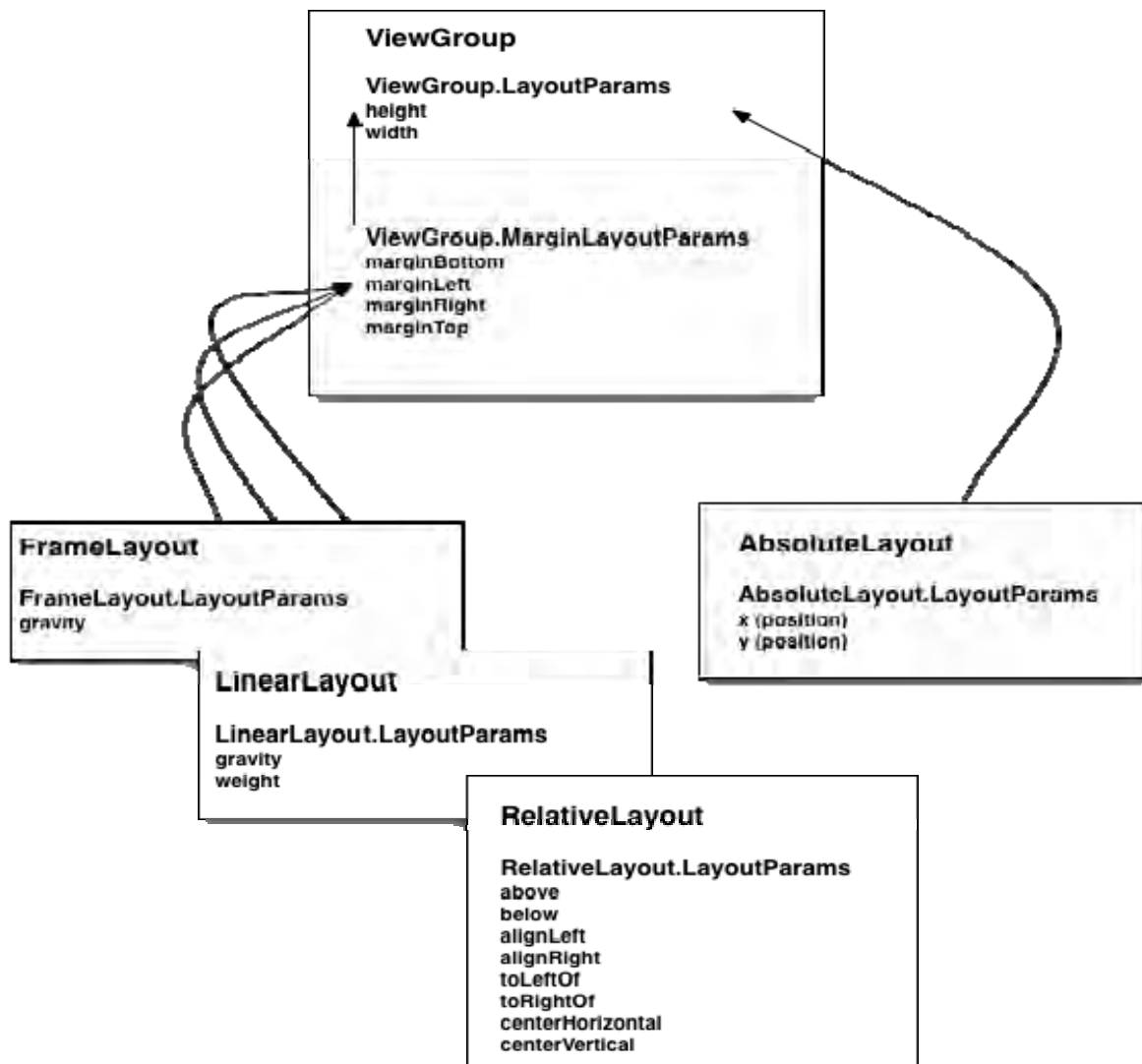


Figure 3.6 Common `ViewGroup` classes with `LayoutParams` and properties provided.

As figure 3.6 shows the base `ViewGroup.LayoutParams` class, which is an inner class of `ViewGroup`, are `height` and `width`. From there an `AbsoluteLayout` type, with `AbsoluteLayout.LayoutParams` allows you to specify the exact X and Y coordinates of the child `View` objects placed within.

As an alternative to absolute layout, you can also use the `FrameLayout`, `LinearLayout`, and `RelativeLayout` subtypes which all support variations of `LayoutParams` that are derived from `ViewGroup.MarginLayoutParams`. A `FrameLayout` is intended to simply frame one child element, such as an image. A `FrameLayout` does support multiple children, but all the items are pinned to the top left – meaning they will overlap each other in a stack. A `LinearLayout` aligns child elements in either a horizontal or vertical line. Recall that we used a `LinearLayout` in code in our `ReviewListView` in listing 3.5. There we created our `View`, and its `LayoutParams` directly in code. And, in our previous `Activity` examples, we used a `RelativeLayout` in our XML layout files that was inflated into our code (again, we will cover XML resources in detail in section 3.3). A `RelativeLayout` specifies child elements relative the each other (above, below, `toLeftOf`, etc.).

So the container is a `ViewGroup`, and a `ViewGroup` supports a particular type of `LayoutParams`. Child `View` elements are then added to the container and must fit into the layout specified by their parents. A key concept to grasp is that even though a child `View` has to lay itself out based on its parents `LayoutParams`, it can also specify a different layout for its own children. This design creates a very flexible palette upon which you can construct just about any type of screen you desire.

For each dimension of the layout a view needs to provide, based on the `LayoutParams` of its parents, it specifies one of three values:

- An exact number
- `FILL_PARENT`
- `WRAP_CONTENT`

The `FILL_PARENT` constant means the take up as much space in that dimension as the parent does (subtracting padding). And, `WRAP_CONTENT` means take up only as much space as is needed for the content within (adding padding). A child `View` therefore requests a size, and the parent makes a decision. In this case, unlike what happens sometimes with actual kids, the children have to listen, they have no choice, and they can't talk back.

Child elements do keep track of what size they initially asked to be, in case layout is recalculated when things are added or removed, but they cannot force a particular size. Because of this `View` elements have two sets of dimensions, the size and width they want to take up (`getMeasuredWidth()` and `getMeasuredHeight()`), and the actual size they end up after a parent's decision (`getWidth()` and `getHeight()`).

Layout takes place in a two step process, first measurements are taken, using the `LayoutParams`, and then items are actually placed on the screen. Components are drawn to the screen in the order they are found in the layout tree, parents first, then children (parents end up behind children, if they overlap in positioning).

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

3.2.6 Handling Focus

Focus is like a game of tag, one and only one component on the screen is always “it.” All devices with user interfaces support this concept. When you are turning the pages of a book, your “focus” is on one particular page (or even word, or letter) at a time. Computer interfaces are no different. Though there may be many different windows and widgets on a particular screen, only one has the current “focus” and can respond to user input. An

event, such as movement of the mouse, or a mouse click, or keyboard press, may trigger the focus to shift to another component.

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

- `nextFocusDown`
- `nextFocusLeft`
- `nextFocusRight`
- `nextFocusUp`

Views can also indicate a particular focus type, `DEFAULT_FOCUS`, or `WEAK_FOCUS`, to set the priority of focus they desire, themselves (default) versus their descendants (weak). In addition to hints, such as `UP`, `DOWN`, and `WEAK`, you can also use the `View.requestFocus()` method directly, if need be, to indicate that focus should be set to a particular `View` at a given time. Manipulating the focus manually though, should be the exception, rather than the rule (the platform logic generally does what you would expect).

Focus gets changed based on event handling logic using the `OnFocusChangeListener` object, and related `setOnFocusChangeListener()` method. This takes us into the world of event handling in general.

3.2.7 Grasping Events

Events are used for changing the focus, and for many other actions as well. We have already implemented several `onClickListener()` methods for buttons in listing 3.2. Those `onClickListener` instances were connected to button presses. The events they were indicating were “hey, somebody pressed me.” This is exactly the same process that focus events go through when announcing or responding to `OnFocusChange` events.

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

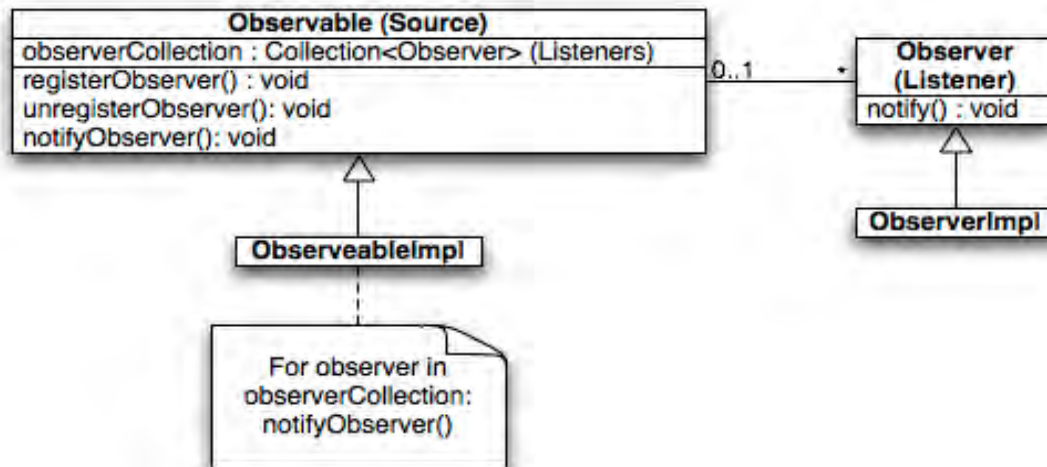


Figure 3.7 A class diagram depicting the Observer design pattern. Each Observable component has zero to many Observers, which can be notified of changes when necessary.

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

With regard to an Android `Button` the two halves are represented as follows:

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

This pattern comes into play in terms of Android `View` items in that many things are Observable and allow other components to attach and listen for events. For example, most of the `View` class methods that begin with “on” are related to events, `onFocusChanged()`, `onSizeChanged()`, `onLayout()`, `onTouchEvent()`, and the like. Similarly, the `Activity` life-cycle methods we have already discussed, `onCreate()`, `onFreeze()` and such, are also event related (on a different level).

Events happen in UI, and all over the platform. For example when an incoming phone call comes in, or a GPS based location changes based on physical movement, many different reactions may occur down the line – many components may want to be notified when the phone rings, or when the location changes (not just one, and not just the UI). Views support events on many levels. When an interface event comes in (a user pressed a button, or scrolled, or selected a portion of a window) it is dispatched to the appropriate view. In general click events, keyboard events, touch events, and focus events are the main types of events you will deal with in the UI.

One very important aspect of the `View` in Android is that the interface is single threaded. If you are calling a method on a `View`, you have to be on the “UI thread.” This

is, again, why we used a `Handler` in listing 3.3, to get data outside of the UI thread, and then notify the UI thread to update the `View` via the `setMessage()` event.

We are admittedly discussing events here on a fairly broad level, to make sure that the overarching concepts are clear. We do this because we cannot cover all of the event methods in the Android APIs in one chapter. Yet, you will see events in examples throughout the book, and in your day to day experiences with the platform. We will call out event examples when necessary, and we will cover them in more detail as we come to specific examples.

Our coverage of events in general, and how they relate to layout, rounds out the majority of our discussion of views – yet we still have one notable related concept to tackle, resources. Views are closely related to resources, but they also go beyond the UI. In the next section we will address all the aspects of resources, including XML defined views.

3.3 Using Resources

We have mentioned Android resources in several areas up to now, and they were initially introduced in chapter 1. Here we will revisit resources with more depth in order to expand on this important topic, and to begin completing the third and final `Activity` `RestaurantFinder` – the `ReviewDetail` screen.

When you begin working with Android you will quickly notice many references to a class named `R`. This was also introduced in chapter 1, and indeed we have used it in our previous `Activity` examples in this chapter. This is the Android resources reference class. Resources are non-code items that are included with your project automatically by the platform.

To begin looking at resources we will first discuss how they are classified into types in Android, and then we will work on examples of each type.

3.3.1 Supported Resource Types

In source, resources are kept in the `res` directory and can be one of several types:

- `res/anim` – XML representations of frame by frame animations
- `res/drawable` – `.png`, `.9.png`, and `.jpg` images
- `res/layout` – XML representations of `View` objects
- `res/values` – XML representations of strings, colors, styles, dimensions, and arrays
- `res/xml` – User defined XML files (that are also compiled into a binary form)
- `res/raw` – Arbitrary and uncompiled files that can be added

Resources are treated specially in Android because they are typically compiled into an efficient binary type (with the noted exception of items that are already binary, and the `raw` type, which is not compiled). Animations, layouts and views, string and color values, and arrays, can all be defined in an XML format on the platform. These XML resources are then processed by the Android Asset Processing Tool (AAPT), which we met in chapter 2, and compiled. Once resources are in compiled form they are accessible in Java through the automatically generated `R` class.

3.3.2 Referencing Resources in Java

The first portion of the `ReviewDetail` Activity, shown in listing 3.6, re-uses many of the `Activity` tenets we have already learned, and uses several sub-components that come from `R.java`, the Android resources class.

Listing 3.6 The first portion of the `ReviewDetail` Activity showing multiple uses of the Android `R` (Resources) class

```
public class ReviewDetail extends Activity {

    private static final int MENU_WEB_REVIEW = Menu.FIRST;
    private static final int MENU_MAP_REVIEW = Menu.FIRST + 1;
    private static final int MENU_CALL_REVIEW = Menu.FIRST + 2;

    private TextView name;           #|1
    private TextView rating;         #|1
    private TextView review;         #|1
    private TextView location;       #|1
    private TextView phone;          #|1
    private ImageView reviewImage;   #|1

    private String link;
    private String imageLink;

    private Handler handler = new Handler() { #2
        @Override
        public void handleMessage(final Message msg) {
            if ((ReviewDetail.this.imageLink != null) && !ReviewDetail.this.imageLink.equals(""))
            {
                try {
                    URL url = new URL(ReviewDetail.this.imageLink);
                    URLConnection conn = url.openConnection();
                    conn.connect();
                    BufferedInputStream bis = new BufferedInputStream(conn.getInputStream());
                    Bitmap bm = BitmapFactory.decodeStream(bis);
                    bis.close();
                    ReviewDetail.this.reviewImage.setImageBitmap(bm);
                }
                catch (IOException e) {
                    Log.e(Constants.LOGTAG, " error", e);
                }
            }
            else {
                ReviewDetail.this.reviewImage.setImageResource(R.drawable.no_review_image);
            }
        }
    };

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

        this setContentView(R.layout.review_detail); #3

        this.name = (TextView) this.findViewById(R.id.name_detail); #|4
        this.rating = (TextView) this.findViewById(R.id.rating_detail); #|4
        this.location = (TextView) this.findViewById(R.id.location_detail); #|4
        this.phone = (TextView) this.findViewById(R.id.phone_detail); #|4
        this.review = (TextView) this.findViewById(R.id.review_detail); #|4
        this.reviewImage = (ImageView) this.findViewById(R.id.review_image); #|4

        RestaurantFinderApplication application = (RestaurantFinderApplication)
        this.getApplication();
        Review currentReview = application.getCurrentReview();
    }
}
```

```

        this.link = currentReview.link;
        this.imageLink = currentReview.imageLink;
        this.name.setText(currentReview.name);
        this.rating.setText(currentReview.rating);
        this.location.setText(currentReview.location);
        this.review.setText(currentReview.content);
        if ((currentReview.phone != null) && !currentReview.phone.equals("")) {
            this.phone.setText(currentReview.phone);
        }
        else {
            this.phone.setText("NA");
        }
    }

    @Override
    protected void onResume() {
        super.onResume();
        this.handler.sendEmptyMessage(1);
    }

    @Override
    public boolean onCreateOptionsMenu(final Menu menu) {
        super.onCreateOptionsMenu(menu);
        menu.add(0, ReviewDetail.MENU_WEB_REVIEW,
            0, R.string.menu_web_review)
            .setIcon(android.R.drawable.ic_menu_info_details);          #5
        menu.add(0, ReviewDetail.MENU_MAP_REVIEW,
            1, R.string.menu_map_review)
            .setIcon(android.R.drawable.ic_menu_mapmode);              #5
        menu.add(0, ReviewDetail.MENU_CALL_REVIEW,
            2, R.string.menu_call_review)
            .setIcon(android.R.drawable.ic_menu_call);                  #5
        return true;
    }
}

. . . remainder of this class is in Chapter 4, when we discuss Intents

```

1. Define View items that will be inflated from XML resources
2. Use a Handler to get an image from the network
3. Set the layout using setContentView() from XML resources
4. Inflate particular views using findViewById() and R.id
5. Refer to String and Drawable resources

In the `ReviewDetail` class we are first defining `View` components that we will later reference from resources **#1**. From there we see a `Handler` that is used to perform a network call to populate an `ImageView` based on a URL. This doesn't relate to resources, but is included here for completeness. Don't worry too much about the details of this here, as it will be covered more when we specifically discuss networking in Chapter 5 **#2**. After the `Handler`, we then set the layout and `View` tree using `setContentView(R.layout.review_detail)` **#3**. This maps to an XML layout file at the path `src/res/layout/review_detail.xml`. Next we see that we are also referencing some of the `View` objects in the layout file directly through resources and corresponding IDs **#4**.

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

You reference resources in code, such as we are here, through the automatically generated `R` class. The `R` class is made up of static inner classes (one for each resource type) that hold references to all of your resources in the form of an `int` value. This value is a constant pointer to an object file through a resource table (which is contained in a special file the APPT tool creates, and the `R` file utilizes).

The last reference to resources we see in listing 3.6 is for the creation of our menu items **#5**. For each of these we are referencing a `String` for text from our own local resources, and we are also assigning an icon from the `android.R.drawable` resources namespace. You can qualify resources in this way and re-use the platform drawables: icons, images, borders, backgrounds, etc. Of course you will likely want to customize much of your own applications and provide your own drawable resources, which you can do, but the platform resources are also available if you need them (and they are arguably the better choice in terms of consistency for the user, if you are calling out to well defined actions like we are here: map, phone call, web page).

We will cover how all the different resource types are handled, and where they are placed in source, in the next several sections. This first type of resources we will look at more closely are those of layouts and views.

3.3.3 Defining Views and Layout through XML Resources

As we have noted in several earlier sections, views and layout can be, and often are, defined in XML rather than in Java code. Defining views and layout as resources in this way, makes them easier to work with, decoupled from the code, and in some cases re-usable in different contexts.

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

An important thing to understand here though is that resources in the `res/layout` directory **don't** have to be layouts. You can define a single `TextView` in a "layout" file the same way you might define an entire tree starting from an `AbsoluteLayout`. Yes, this makes the "layout" name and path potentially confusing, but that is how it is setup. (It might make more sense to have separate `res/layout` and `res/view` directories, but that arguably might be confusing too, so just keep in mind that `res/layout` is useful for more than layout.)

You can have as many XML layout/view files as needed, all defined in the `res/layout` directory. Each View is then referenced in code based on the **type and ID**. Our layout file for the `ReviewDetail` screen, `review_detail.xml`, which is shown in listing 3.7, is referenced in the Activity code as `R.layout.review_detail` – which is a pointer to the `RelativeLayout` parent View object in the file.

Listing 3.7 XML Layout Resource file for the ReviewDetail Activity, review_detail.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout                                #1
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"          #|2
    android:layout_height="fill_parent"        #|2
    android:gravity="center_horizontal"
    android:padding="10px"                     #3
    android.setVerticalScrollBarEnabled="true"
>
```

```

<ImageView android:id="@+id/review_image"                                #4
    android:layout_width="100px"
    android:layout_height="100px"
    android:layout_marginLeft="10px"
    android:layout_marginBottom="5px" />

<TextView android:id="@+id/name_detail"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_below="@id/review_image"                             #5
    android:layout_marginLeft="10px"
    android:layout_marginBottom="5px"
    style="@style/intro_blurb" />                                     #6

<TextView android:id="@+id/rating_label_detail"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_below="@id/name_detail"
    android:layout_marginLeft="10px"
    android:layout_marginBottom="5px"
    style="@style/label"
    android:text="@string/rating_label" />                             #7

. . . remainder of file omitted for brevity

</RelativeLayout>

```

1. Defining the root View element, typically *Layout
2. Defining LayoutParams through XML
3. Defining other View parameters in XML, such as padding
4. Including a child element with ID
5. Referencing another resource from within this resource
6. Referencing a style for a View in XML

In this file we see that we are using a RelativeLayout **#1**. This is the ViewGroup at the root of the view tree. LayoutParams are then also defined in XML using the `android:layout_[attribute]` convention (where [attribute] refers to a layout attribute) **#2**. Along with layout, other View related attributes can also be defined in XML with counterpart XML attributes to the methods available in code, such as `android:padding` which is analogous to `setPadding()` **#3**.

After the RelativeLayout parent itself is defined then the child view elements are added. Here we are using an ImageView and multiple TextView components. Each of the components is given an ID using the form `android:id="@+id/[name]"` **#4**. When an ID is established in this manner, an int reference is defined in the resource table, and named with the specified name. This allows other components to reference the ID by the friendly textual name.

Once views are defined as resources, the Activity method `findViewById()` can be used to obtain a reference to a particular view using the name. That view can then be manipulated in code. For example, in listing 3.6 we grabbed the rating TextView as follows:

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

This inflates and hands off the `rating_detail` element we saw in listing 3.7. Note that child views of "layout" files end up as "id" type in R.java (they are not `R.layout.name`, rather they are `R.id.name`, even though they are required to be placed in the `res/layout` directory).

The R.java “id” type

Each child View element in any of your layout files should be uniquely named. Each application will compile all the layout files into one namespace. If you have “button1” in one layout file, and “button1” in another file in the same application, you will get unpredictable results (but curiously, not an error – still only one item ends up in the R.id static class named “button1”).

The properties for the `View` object are all defined in XML, this includes the layout. Because we are using a `RelativeLayout` we use attributes that place one `View` relative to another, such as `below` or `toRightOf`. This is done with the `android:layout_below="@id/[name]` syntax **#5**. The `@id` syntax is a way to reference other resources items from within a current resources file. Using this approach you can reference other elements defined in the current file you are working on, or other elements defined in other resource files.

Some of our views represent labels, which are shown on the screen as is and are not manipulated in code, such as `rating_label_detail`. Others we will populate at run time, these don't have a text value set, such as `name_detail`. The elements that we do know the values of, the labels, are defined with references to externalized strings that we have defined in another resource file (`strings.xml`).

The same approach is applied with regard to styles, using the syntax `style="@style/[stylename]` **#7**. Strings, styles, and colors are themselves defined as resources in another type of resource file.

3.3.4 Externalizing Values

It is fairly common practice in the programming world to externalize string literals from code. In Java this is done with a `ResourceBundle`, and or a properties file. Externalizing references to strings in this way allows the value of a component to be stored updated separately from the component itself, away from code.

Android includes support for “values” resources that are subdivided into several groups: animations, arrays, styles, strings, dimensions, and colors. Each of these items is defined in a specific XML format, and then made available in code as references from the `R` class, just like layouts, views, and drawables. For the `RestaurantFinder` application we are using externalized strings as shown in listing 3.8, `strings.xml`.

Listing 3.8 The externalized strings for the `RestaurantFinder` sample application, `strings.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name_criteria">RestaurantFinder - Criteria</string>          #1
  <string name="app_name_reviews">RestaurantFinder - Reviews</string>
  <string name="app_name_review">RestaurantFinder - Review</string>
  <string name="app_short_name">Restaurants</string>

  <string name="menu_get_reviews">Get reviews</string>
  <string name="menu_web_review">Get full review</string>
  <string name="menu_map_review">Map location</string>
  <string name="menu_call_review">Call restaurant</string>
  <string name="menu_change_criteria">Change review criteria</string>
```

```

<string name="menu_get_next_page">Get next page of results</string>

<string name="intro_blurb_criteria">Enter review criteria</string>
<string name="intro_blurb_detail">Review details</string>

. . . remainder omitted for brevity

</resources>

```

1. Using a string element with a name attribute

As is evident from the `strings.xml` example, this is very straightforward. This file uses a `<string>` element with a `name` attribute **#1** for each string value you need. We have used this file for the application name, menu buttons, labels, and alert validation messages. This format is known as “simple value” in Android terms. This file is placed in source at the `res/values/strings.xml` location. In addition to strings, colors and dimensions can be defined in the same way.

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

- pixels (px)
- inches (in)
- millimeters (mm)
- points (pt)
- density-independent pixels (dp)
- scale-independent pixels (sp)

Colors can be defined in `colors.xml` and are defined with the `<color>` element: `<color name=color_name>#color_value</color>`. Colors values are expressed in RGB codes. Color, and dimension files are also placed in the `res/values` source location.

Though we haven't defined separate colors and dimensions for the RestaurantFinder application, we are using several styles, which we referenced in listing 3.7. The style definitions are shown in listing 3.9. This is where we move beyond a simple value layout to a specific “style” XML structure (though styles are still placed in source in the `res/values` directory, which can be confusing).

Listing 3.9 Values resource defining re-usable styles, `styles.xml`.

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="intro_blurb">    #1
        <item name="android:textSize">22sp</item>        #|2
        <item name="android:textColor">#ee7620</item>    #|2
        <item name="android:textStyle">bold</item>        #|2
    </style>

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

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

    . . . rest of file omitted for brevity

```

```
</resources>
```

1. Using A `<style>` element with name attribute
2. Using `<item>` element children to define style

The Android styles approach is a similar concept to using Cascading Style Sheets (CSS) with HTML. Styles are defined in `styles.xml`, and then referenced from other resources or code. Each `<style>` element **#1** has one or more `<item>` children that defines a single setting **#2**. Styles are made up of the various `View` settings: sizes, colors, margins, and such. Styles are very helpful because they facilitate easy re-use, and the ability to make changes in one place. Styles are applied in layout XML files by associating a style name with a particular `View` component, such as: `style="@style/intro_blurb"` (note that in this case style is not prefixed with the `android:` namespace, it is a custom local style and not one provided by the platform).

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

Along with styles and themes, Android also supports a specific XML structure for defining arrays as a resource as well. Arrays are placed in source in `res/values/arrays.xml` and are helpful for defining collections of constant values, such as the cuisines we used to pass to our `ArrayAdapter` back in listing 3.1. Listing 3.10 shows how these arrays are defined in XML.

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="cuisines">           #1
        <item>ANY</item>              #2
        <item>American</item>
        <item>Barbeque</item>
        <item>Chinese</item>
        <item>French</item>
        <item>German</item>
        <item>Indian</item>
        <item>Italian</item>
        <item>Mexican</item>
        <item>Thai</item>
        <item>Vegetarian</item>
        <item>Kosher</item>
    </array>
</resources>
```

1. Each `<array>` element has a name attribute
2. Each item in an array is defined with an `<item>` element

Arrays are defined as resources using an `<array>` element with a name attribute **#1**, and include any number of `<item>` child elements to define each array member. You can access arrays in code using the syntax we saw in listing 3.1: `String[] ratings = getResources().getStringArray(R.array.ratings)`.

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

Going past simple values for strings, colors, and dimensions, and more involved but still straightforward structures for styles, arrays, raw files, and raw XML, the next type of resources we need to explore are animations.

3.3.5 Providing Animations

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

Animation XML files are placed in the `res/anim` source directory. There can be more than one anim file, and like layouts you reference the respective animation you want by name/id. Android supports four types of animations:

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

In addition to the types of animations supported, Android also provides five attributes that can be used with any animation type:

- `duration` – duration in milliseconds
- `startOffset` – offset start time in milliseconds
- `fillBefore`
- `fillAfter`
- `interpolator` – used to define a velocity curve for speed of animation

Listing 3.11 shows a very simple animation that can be used to scale a View.

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

```
<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android" #1
    android:fromXScale="0.5"
    android:toXScale="2.0"
    android:fromYScale="0.5"
    android:toYScale="2.0"
    android:pivotX="50%"
    android:pivotY="50%"
    android:startOffset="700"
    android:duration="400"
    android:fillBefore="false" />
```

1. Using the `<scale>` animation

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

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

This will scale the view element up in size on both the X and Y axes. Though we do not have any animations in the RestaurantFinder sample application by default, to see this work you can simply add the `startAnimation` method() to any view element in the code and reload the application. Animations can come in handy so you should be aware of them. We will cover animations, and other graphics topics, in detail in chapter 9.

With our journey through Android resources now complete we next need to address the final aspect of RestaurantFinder we have yet to cover, the `AndroidManifest.xml` manifest file, which is required for every Android application.

GH

3.4 Understanding the AndroidManifest file

As we learned in chapter 1, Android requires a manifest file for every application – `AndroidManifest.xml`. This file, which is placed in the root directory of the project source, describes the application context, and any supported activities, services, intent receivers, and or content providers, as well as permissions. We will learn more about services, intents, and intent receivers in the next chapter, and about content providers in chapter 5, for now the manifest for our RestaurantFinder sample application, as shown in listing 3.11, contains only the `<application>` itself, and an `<activity>` element for each screen, and several `<uses-permission>` elements.

Listing 3.11 The RestaurantFinder AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"          #1
    package="com.msi.manning.restaurant">

    <application android:icon="@drawable/restaurant_icon_trans"
        android:label="@string/app_short_name" android:name="RestaurantFinderApplication"
        android:allowClearUserData="true" android:theme="@android:style/Theme.Black"> #2

        <activity android:name="ReviewCriteria"
            android:label="@string/app_short_name"> #3
            <intent-filter> #4
                <action android:name="android.intent.action.MAIN" />
                <category
                    android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name="ReviewList"
            android:label="@string/app_name_reviews"> #5
            <intent-filter> #6
                <category
                    android:name="android.intent.category.DEFAULT" />
                <action
                    android:name="com.msi.manning.restaurant.VIEW_LIST" />
            </intent-filter>
        </activity>

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

    </application>

    <uses-permission android:name="android.permission.CALL_PHONE" /> #7
```

```
<uses-permission android:name="android.permission.INTERNET" />    #|7
</manifest>
```

1. The <manifest> declaration with android namespace
2. The RestaurantFinderApplication declaration with class and properties
3. The ReviewCriteria Activity screen
4. The MAIN LAUNCHER Intent filter
5. The ReviewList Activity screen
6. Defining a custom Intent filter for the ReviewList Activity
7. Adding a permissions for the application

In the RestaurantFinder descriptor file we first see the root <manifest> element declaration, which includes the application's package declaration, and the Android namespace **#1**. Then we see the <application> element with both the name and icon attributes defined **#2**. You don't have to include the name attribute here unless you want to extend the default Android Application object to provide some global state to your application (which we did to store the Review object each screen is operating on). The icon is also optional, if not specified a system default is used to represent your application on the main menu.

After the application itself is defined we see the child <activity> elements within. These, obviously, define each Activity the application supports **#3** (and note that the manifest file can use Android resources as well, such as with @string/app_name). As was noted when discussing activities in general, one Activity in every application is the starting point, this Activity has the <intent-filter> action MAIN and category LAUNCHER designation **#4**. This tells the Android platform how to start an application from the Launcher, meaning this activity will be placed in the main menu on the device.

Past the ReviewCriteria Activity we see another <activity> designation for ReviewList **#5**. This Activity also includes an <intent-filter>, but for our own action, com.msi.manning.chapter3.VIEW_LIST **#6**. This tells the platform that this Activity should be invoked for this "intent." We will learn more about exactly how this works in the next chapter. Lastly in our manifest we have a <uses-permission> element. This also relates to intents, and tells the platform that this application needs the CALL_PHONE permission (recall we discussed several aspects of security in chapter 2, and we will touch on this in various contexts throughout the book).

The RestaurantFinder sample application uses a fairly basic manifest file with three activities, and a series of intents. This is not a comprehensive example of course, but all of the elements an Android manifest supports are shown in table 3.5 for reference.

Table 3.5 Supported AndroidManifest.xml elements and their descriptions

Element	Position	Description
<manifest>	root	Define application package and Android namespace
<uses-permission>	root	Requests a security permission
<permission>	root	Declares a security permission
<instrumentation>	root	Declares a test instrumentation component
<application>	root	Defines an application, class name, label, icon, theme (one per manifest)

<activity>	child of <application>	Define an Activity class
<intent-filter>	child of <activity>	Declares the Intents an Activity supports
<action>	child of <intent-filter>	Intent action
<category>	child of <intent-filter>	Intent category
<data>	child of <intent-filter>	Intent MIME type, URI scheme, URI authority, or URI path
<meta-data>	child of <activity>	General meta data, accessible via ComponentInfo.metaData
<receiver>	root	Define an IntentReceiver, respond to Intents (also supports <intent-filter> children)
<service>	root	Define a background Service (also supports <intent-filter> children)
<provider>	root	Define a ContentProvider to manage persistent data for access by other applications

Wrapping up the description of the manifest file completes our discussion of views, activities, resources, and in general working with user interfaces in Android.

3.5 Summary

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

In addition, here we looked at how Android handles various types of resources, from simple types, to more involved layouts, arrays, and animations – and how these related to, and are used within, views and activities. Lastly, we also explored the AndroidManifest.xml application descriptor and how it brings all these pieces together to define an Android “application.”

This chapter has provided a good foundation for general Android UI development, next we need to go deeper into the concepts of Intent and IntentReceiver classes, the communication layer that Android activities, and other components, make use of. We will cover these items, along with longer running Service processes, and the Android Inter-Process Communication (IPC) system involving the Binder, next in chapter 4.

Intents and Services

In this chapter,

- Intents and IntentFilters
- BroadcastReceivers
- Services
- Inter-Process Communication and AIDL

The canonical Android application is made of `Activity` and `View` objects on the front end, and `Intent` and `Service` objects on the back end. As we covered in chapter 3, activities are roughly comparable to UI screens, and views are UI components. When a user interacts with a screen, that screen usually represents a task, such as: display a list of choices and allow selection, gather information through form input, or display graphics and data. Once each screen is done with its individual job it usually hands off to another component to perform the next task.

In Android terms “hand off to another component” is done with an `Intent`. We were introduced to this concept and term in chapter 1, and we saw some limited amounts of `Intent` related code in our examples in chapter 3. In this chapter we are going to expand on the details, including looking more closely at what exactly an `Intent` is, and how they are resolved and matched with an `IntentFilter`. Along the way we will complete the `RestaurantFinder` application we started in chapter 3, finishing up the code and elaborating on the intents involved. `RestaurantFinder` uses `Intent` objects internally, to go from `Activity` to `Activity`, and also calls on intents from Android built in applications – to phone a restaurant, map directions to a restaurant, and visit a restaurant review web page.

After we complete the `RestaurantFinder` application we will move on to another sample application in this chapter – `WeatherReporter`. `WeatherReporter` will make use of the Yahoo! Weather API to retrieve weather data and display it, along with weather alerts, to the user on the Android platform. Through the course of the `WeatherReporter` application we will exercise intents in a new way, using an `BroadcastReceiver` and a `Service`.

An `BroadcastReceiver`, as the name implies, also deals with intents, but is used to catch broadcasts to any number of interested receivers, rather than to signal a particular action from an `Activity`. `Services` are background processes, rather than UI screens – but they are also invoked with a call to action, an `Intent`.

Lastly in this chapter, in relation to services, we will examine the Android mechanism for making Inter-Process Communication (IPC) possible using `Binder` objects and the Android Interface Definition Language (AIDL). Android provides a high performance way for different processes to pass messages between one another. This is important because every application runs within its own isolated process (for security and performance purposes, this is owed to the Linux heritage of the platform). To enable communication between components in different processes, the platform provides a path via a specified IPC approach.

The first thing we need to cover is the basic means to call an “action” from within any component, this means we need to begin with a focus on `Intent` details.

4.1 Working with Intents

`Intents` are the communications network of the applications on the Android platform. In many ways the Android architecture is similar to larger service oriented architecture (SOA) approaches in that each `Activity` makes a type of `Intent` call to get something done, without knowing exactly what the receiver of the `Intent` may be.

In an ideal situation you don't care how a particular task gets performed, rather you care that it gets done, and is completed to your requirements. That way, you can divide up what you need to get done at a particular time – your intent – and concentrate on the problem you are trying to solve, rather than worrying about specific underlying implementation details.

Intents are “late binding,” and this is one of the things that makes them a bit different from what you might be used to. This means they are mapped and routed to a component that can handle a specified task at run time, rather than at build or compile time. One `Activity` tells the platform “I need a map to Langtry, TX, US,” and another component, one the platforms determines is capable, handles the request and returns the result. With this approach individual components are decoupled, and can be modified, enhanced, and maintained, without requiring changes to a larger application or system.

With that concept, and the advantages the design intends in mind (no pun, well, intended), here we will look at exactly how an `Intent` is defined in code, how an `Intent` is invoked by an `Activity`, how intent resolution takes place using `IntentFilters`, and some activities that are built into the platform ready for you to take advantage of.

4.1.1 Defining Intents

Intents are made up of three primary pieces of information, action, categories, and data, and also include an additional set of optional elements. An “action” is simply a `String`, as is a “category,” and “data” is defined in the form of a `Uri` object. A `Uri` is a generic Uniform Resource Identifier (URI), as defined by RFC 2396, which includes a scheme, an authority, and optionally a path (we will find out more about these parts in the next section). Table 4.1 lays out all of the components of an `Intent` object.

Table 4.1 Intent elements and description.

Intent Element	Description
Action	Fully qualified String indicating action (for example: android.intent.action.MAIN).
Data	Data to work with expressed as a URI (for example content://contacts/1).
Category	Additional meta data about Intent (for example: android.intent.category.LAUNCHER).
Type	Specify an explicit MIME type (as opposed to being parsed from a URI).
Component	Specify an explicit package and class to use for Intent, optional, normally inferred from action, type, and categories.
Extras	Extra data to pass to the Intent that is in the form of a Bundle

Intents therefore, typically, express a combination of action, data, and attributes such as category. This designation is used by the system as a sort of language to resolve exactly what class should be used to fill the request.

When a component such as an `Activity` wants to call upon an `Intent`, it can do so in one of two ways:

- Implicit Intent invocation
- Explicit Intent invocation

An implicit `Intent` invocation is one in which the platform determines which component is the best to run the `Intent`. This happens through a process of intent resolution using the action, data,

and categories. We will explore this resolution process in detail in the next section. On the other hand, an explicit `Intent` invocation is one in which the code directly specifies which component should handle the `Intent`. Explicit invocation is done by specifying either the `Class` or `ComponentName` of the receiver (where `ComponentName` is a `String` for the package, and a `String` for the class).

To explicitly invoke an `Intent`, you can simply use the following form: `Intent(Context ctx, Class cls)`. With this approach you can short circuit all the Android intent resolution wiring, and directly pass in an `Activity` or `Service` class reference to handle the `Intent`. While this approach is convenient, and fast, and therefore sometimes arguably appropriate, it does also introduce some tight-coupling that may be a disadvantage later.

In listing 4.1 we see the final portion of the `ReviewDetail` `Activity` from the `RestaurantFinder` sample application. This listing shows several implicit `Intent` invocations. (Recall, we began this application in chapter 3, where the first half of this class is shown in listing 3.6.)

Listing 4.1 The second portion of the `ReviewDetail` `Activity`, demonstrating `Intent` invocation

```
@Override
public boolean onOptionsItemSelected(final int featureId, final MenuItem item) {
    Intent intent = null;          #1
    switch (item.getItemId()) {
    case MENU_WEB_REVIEW:
        if ((this.link != null) && !this.link.equals("")) {
            intent = new Intent(Intent.ACTION_VIEW, Uri.parse(this.link));    #2
            this.startActivity(intent);    #3
        }
        else {
            new AlertDialog.Builder(this).setTitle(this.getResources()
                .getString(R.string.alert_label))
                .setMessage(R.string.no_link_message)
                .setPositiveButton("Continue", new OnClickListener() {
                    public void onClick(final DialogInterface dialog, final int arg1) {
                    }
                })
                .show();
        }

        return true;
    case MENU_MAP_REVIEW:
        if ((this.location.getText() != null) && !this.location.getText().equals("")) {
            intent = new Intent(Intent.ACTION_VIEW,
                Uri.parse("geo:0,0?q=" + this.location.getText().toString()));    #4
            this.startActivity(intent);
        }
        else {
            new AlertDialog.Builder(this).setTitle(this.getResources()
                .getString(R.string.alert_label))
                .setMessage(R.string.no_location_message)
                .setPositiveButton("Continue", new OnClickListener() {
                    public void onClick(final DialogInterface dialog, final int arg1) {
                    }
                })
                .show();
        }

        return true;
    case MENU_CALL_REVIEW:
        if ((this.phone.getText() != null) &&
            !this.phone.getText().equals("") && !this.phone.getText().equals("NA")) {
            String phoneString = ReviewDetail.parsePhone(this.phone.getText().toString());
            intent = new Intent(Intent.ACTION_CALL, Uri.parse("tel:" + phoneString));    #5
            this.startActivity(intent);
        }
        else {
            new AlertDialog.Builder(this).setTitle(this.getResources()
                .getString(R.string.alert_label))
                .setMessage(R.string.no_phone_message)
                .setPositiveButton("Continue", new OnClickListener() {
                    public void onClick(final DialogInterface dialog, final int arg1) {
                    }
                })
                .show();
        }

        return true;
    }
    return super.onOptionsItemSelected(featureId, item);
}
```

```
    . . . parsePhone omitted for brevity  
}
```

1. Declare an Intent
2. Setting the Intent for the web menu item
3. Using StartActivity(intent)
4. Setting the Intent for the map menu item
5. Setting the Intent for the call menu item

The `Review` object that the `ReviewDetail` Activity displays to the user contains the address and phone number for a restaurant, and a link to the full online review. Using this Activity the user can choose, through the menu, to display a map with directions to the restaurant, call the restaurant, or view the full review in a web browser. To allow all of these actions to take place, `ReviewDetail` uses built in Android applications, through implicit Intent calls.

First an Intent class instance is initialized to null **#1**, so it can later be used by the various menu cases. Then, if the `MENU_WEB_REVIEW` menu button is selected by the user, we create a new instance of the intent variable by passing in an action and some data **#2**. For the action we are using the String constant `Intent.VIEW_ACTION`. The value of this constant is `android.app.action.VIEW`, a fully qualified String including the package so as to be unique. The Intent class has a host of constants like this that represent common actions, for example: `Intent.EDIT_ACTION`, `Intent.INSERT_ACTION`, and `Intent.DELETE_ACTION`, to name a few. Various activities and services use these same values when they declare they support a particular Intent (and you can re-use these constants too, where applicable, see the Android JavaDocs for a complete list of what is available: <http://code.google.com/android/reference/android/content/Intent.html>).

After the action is declared we then come to the data, in this case we are using `Uri.parse(link)` to specify a Uri (where `link` is an HTTP URL). The `parse(String s)` method simply parses the parts of a URI and creates a Uri object. This Uri is used in the resolution process we will cover next, basically the type can be derived from the Uri, or the scheme, authority, and path themselves can be used. This allows the correct component to answer the `startActivity(Intent i)` request **#3**, and render the resource identified by the Uri. As you can see, we haven't directly declared any particular Activity or Service for the Intent, we are simply saying we want to VIEW `http://somehost/somepath`. This is the late binding aspect in action. When it comes to a web URL it is pretty obvious how this works, but the same concept is applied in Android with many other built in data types (and you can define your own when necessary, as we shall see).

The next menu item `ReviewDetail` handles is for the `MENU_MAP_REVIEW` case, where we see the Intent re-initialized to use the `Intent.VIEW_ACTION` again, but this time with a different type of Uri being parsed - `"geo:0,0?q=" + street_address` **#4**. This combination of VIEW and "geo" scheme invokes a different Intent, this time within the built in Maps application. And, lastly, we see the `MENU_MAP_CALL` case, where the Intent is re-initialized again, this time to make a phone call using the `Intent.CALL_ACTION` and the "tel:" Uri scheme **#5**.

Through those simple statements our `RestaurantFinder` application is using implicit Intent invocation to allow the user to phone or map the restaurant they have selected, or to view the full review web page. These menu buttons are seen in the screen show shown in figure 4.1.

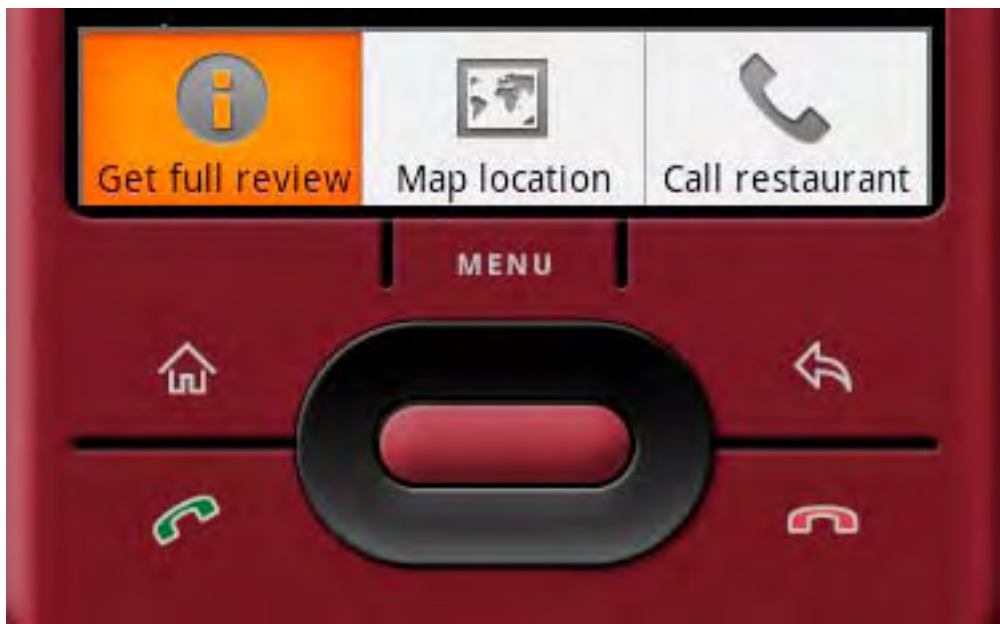


Figure 4.1 The menu buttons on the RestaurantFinder sample application, used for invoking respective intents.

To get the menu buttons on the `ReviewDetail` activity of the RestaurantFinder sample application to work, we did not have to code all the functionality ourselves, we simply had to leverage the existing applications Android provides, by telling the platform our intentions. Those last steps actually complete the RestaurantFinder application, which can now search for reviews, allow the user to select a particular review from a list, display a detailed review, and allow the user to use those built in additional applications to find out more about a selected restaurant.

We will find out more about all the of built in apps and action-data pairs in section 4.1.3. Now we will turn our focus to more detail on the intent resolution process, where we will uncover a bit more about `Intent` action and data.

4.1.2 Intent Resolution

Three types of Android components can register to be `Intent` handlers: `Activity`, `BroadcastReceiver`, and `Service`. These components typically register with the platform to be the destination for particular intent types using the `<intent-filter>` element in the `AndroidManifest.xml` file, as we have seen.

Each `<intent-filter>` element is parsed into an `IntentFilter` object. When a package is installed on the platform, the components within are registered, including the intent filters. Once the platform has a registry of intent filters, it basically knows how to map any `Intent` requests that come in to the correct installed `Activity`, `BroadcastReceiver`, or `Service`.

When an `Intent` is requested, resolution takes place through the registered filters, using the action, data, and categories of the `Intent`. There are a few basic rules about `Intent` to `IntentFilter` matching you should be aware of:

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

In the next few sections we will explore these aspects in greater detail, as they are paramount to understanding how intents work.

ACTION AND CATEGORIES

The action and category parts are pretty simple. These boil down to `String` objects, one for the action, multiple possible for the categories. If the action is not specified in the `IntentFilter`, it will then match any action coming from an `Intent` (all actions work). With categories, the `IntentFilter` is a superset. An `IntentFilter` can have additional categories beyond what an `Intent` specifies to match, but must have at least what the `Intent` specifies. Also, unlike with action, an `IntentFilter` with no categories will **only** match an `Intent` with no categories (it is not treated as a wildcard). So first, action and category specifications have to be a match.

Before we move on to the next matching component, data, it's important to understand that data is optional. You can work with action and category alone, and in many cases that suffices. This is, for example, the technique we used in the `ReviewList` Activity we built in chapter 3. There the `IntentFilter` was defined (in the manifest XML) as shown in listing 4.2.

Listing 4.2 Declaring the `ReviewList` Activity with `intent-filter` in the manifest XML file.

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

To match the filter declared in listing 4.2, we used the following `Intent` in code (where `Constants.INTENT_ACTION_VEW_LIST` is the `String` `"com.msi.manning.restaurant.VIEW_LIST"`):

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

NOTE

The `DEFAULT` category designation on an Activity means that said Activity should be present as an option for the “default” action – center button press – for a particular type of data. This is usually specified in an `IntentFilter`, but does not typically need to be present in an `Intent` (the filter will still match, categories are a superset).

DATA

After the action and categories are resolved, then `Intent` data comes into play. The data can be either an explicit MIME type, or a combination of scheme, authority, and path – either of these can be derived from a `Uri`. The `Uri` seen in figure 4.2 is an example of using scheme, authority, and path.



Figure 4.2 The portions of a `Uri` that are used in Android, showing scheme, authority, and path.

As opposed to scheme, authority, and path, using an explicit MIME type within a `Uri` looks like the following:

```
content://com.google.provider.NotePad/notes
```

You might reasonably ask how this is differentiated from scheme/authority/path, because those elements are really still there? The answer is the "content://" scheme. That indicates a type override to the platform. The type itself is defined in the manifest of the package supplying the content "provider." We will look at more details concerning content providers later in this chapter.

When `IntentFilters` are defined they set the boundaries for what they will match in terms of type, scheme, authority, and path. There is a somewhat convoluted resolution path that follows:

1. If scheme is present and type is NOT present, Intents with any type will match
2. If type is present and scheme is NOT present, Intents with any scheme will match
3. If neither scheme or type are present, only Intents with neither scheme or type will match
4. If an authority is specified, a scheme must also be specified
5. If a path is specified, a scheme and authority must also be specified

The majority of times what you are matching will be fairly straightforward, but as you can see, with these rules, and multiple levels of authorities and and schemes it can get complicated. To boil down intent resolution, think of `Intent` and `IntentFilter` as separate pieces of the same puzzle. When you call an `Intent` in an Android application, the system resolves the `Activity` or `Service` (or `BroadcastReceiver`) to handle your request through this resolution process using the action, categories, and data (type or scheme, authority, and path) provided. The system searches all the pieces of the puzzle it has until it finds one that meshes with the one you have just handed it, and then it snaps those pieces together to make the "late binding" connection.

A more involved example of this matching is shown in figure 4.3. There we can see that an `IntentFilter` is defined with an action, the default category, and a combination of scheme and authority (leaving out the path so that any path will match). An example of an `Intent` that would match this filter is also shown, in this case using a `Uri` that is passed in by the next sample application we will build, `WeatherReporter`.

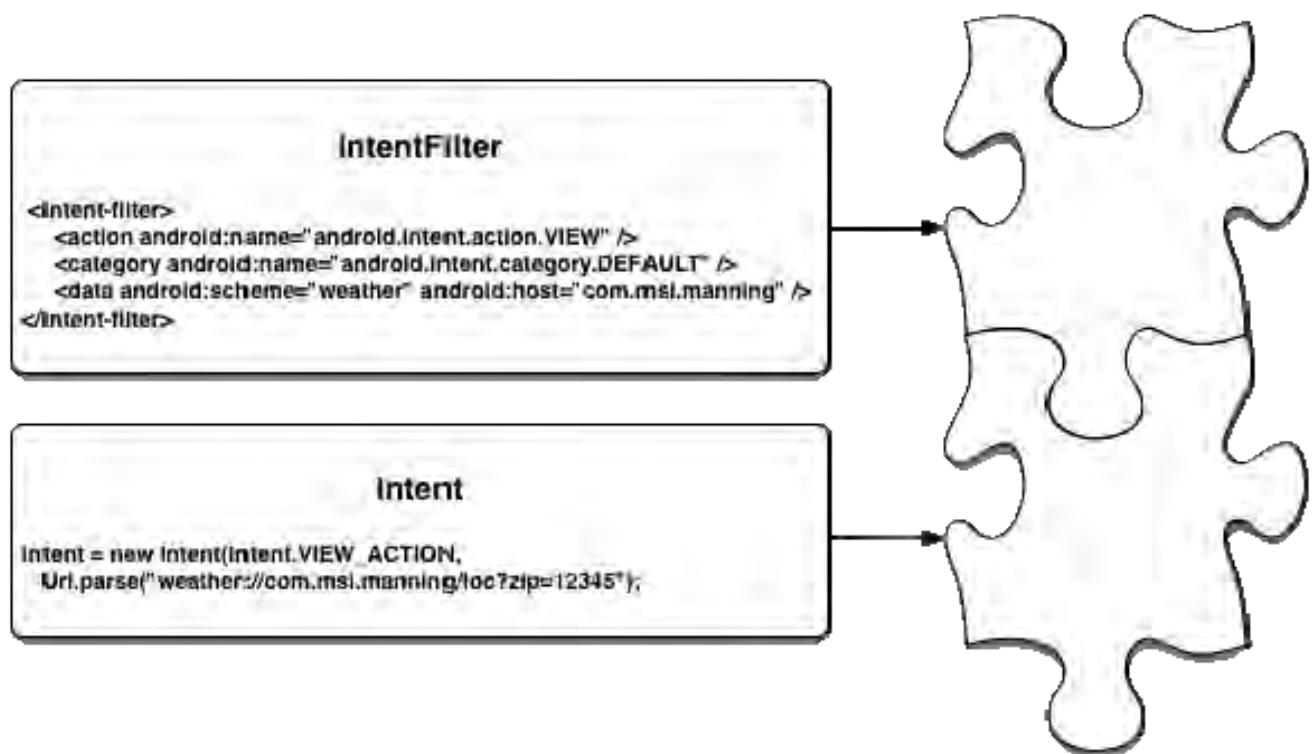


Figure 4.3 Example Intent and IntentFilter matching using a filter defined in XML.

The `IntentFilter` shown in figure 4.3 matches with the action, category, and data (extracted from the `Uri` passed in) to the `Intent` being used. This intent and filter come from the next sample application we are going to begin working on, a weather reporting and alerting application. This application will carry us through most of the remaining concepts in this chapter, and into the next.

4.1.3 Matching a custom URI

The concept behind `WeatherReporter`, the next sample application we will build, is that it will make use of the Yahoo! Weather API to retrieve weather data and display it to the user on the Android platform. Optionally this application will also alert users of severe weather for locations they have indicated they are interested in (based on either the current location of the device, or specified postal code).

Within this project we will see how a custom URI can be defined and registered with a matching intent filter to allow any other application to invoke a weather report through an intent. When complete the main screen of the `WeatherReporter` application will look like what is shown in figure 4.4.



Figure 4.4 A screen shot of the main screen in the sample WeatherReporter application showing the weather forecast for the current location and a checkbox to indicate whether or not alerts should be enabled.

To begin this application we have to cover some basics first, such as the manifest file. We have already explored manifest files in general in chapter 3, here we are filling in some details for this application, and we are further reinforcing how intent filters are defined in XML. The manifest for WeatherReporter is shown in listing 4.3.

Listing 4.3 The Android manifest file for the WeatherReporter application, showing intent filter definition and other application details.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.weather">
    Please post comments or corrections to the Author Online forum at
    http://www.manning-sandbox.com/forum.jspa?forumID=411
    Licensed to Thow Way Chiam <ken.ctw@gmail.com>
```

```

<application android:icon="@drawable/weather_sun_clouds_120"
    android:label="@string/app_name" android:theme="@android:style/Theme.Black"
    android:allowClearUserData="true">

    <activity android:name="ReportViewSavedLocations"
        android:label="@string/app_name_view_saved_locations" />    #1

    <activity android:name="ReportSpecifyLocation"
        android:label="@string/app_name_specify_location" />    #1

    <activity android:name="ReportViewDetail"
        android:label="@string/app_name_view_detail">    #1
        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:scheme="weather" android:host="com.msi.manning" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.VIEW" />
            <data android:scheme="weather" android:host="com.msi.manning" />
        </intent-filter>
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

    <receiver android:name=".service.WeatherAlertServiceReceiver">    #2
        <intent-filter>
            <action android:name="android.intent.action.BOOT_COMPLETED" />
        </intent-filter>
    </receiver>

    <service android:name=".service.WeatherAlertService" />    #3

</application>

<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />    #|4
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />    #|4
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />    #|4
<uses-permission android:name="android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />    #|4
<uses-permission android:name="android.permission.INTERNET" />

</manifest>

```

1. Define activities
2. Define a receiver
3. Define a service
4. Include necessary permissions

In the WeatherReporter manifest we see that we have three activities defined **#1**. The most interesting is the ReportViewDetail Activity, which we will see a portion of in listing 4.4. This Activity has multiple intent filters defined that match it, including one denoting it is the MAIN LAUNCHER, and one with the “weather://com.msi.manning” scheme and authority we saw in figures 4.2 and 4.3. This is the custom URI our application supports.

You can use any combination of scheme, authority, and path, as we have here - or you can use an explicit MIME type. We will find out more about MIME types and how they are processed in chapter 5, where we will look specifically at how to work with data sources and use an Android concept known as a ContentProvider.

After our activities we see that we are using the <receiver> element in the manifest file to refer to an BroadcastReceiver class **#2**. We will uncover what an BroadcastReceiver is all about in section 4.2, but the important part for now is that an <intent-filter> is also used here to associate an Intent – in this case for the the BOOT_COMPLETED action. With this association we are telling the platform to invoke the WeatherAlertServiceReceiver class after the bootup sequence is completed.

Then in our manifest we also have a Service definition **#3**. We will see how this Service is built, and how it is used with our WeatherReporter application to poll for severe weather alerts in

the background, in section 4.3 After this the last thing in our manifest is a series of permissions the application requires **#4**.

With the foundation for our sample application in place via the manifest, the next thing we need to look at, with regard to intents and data, before moving on to additional concepts, is the `onStart` method of main Activity WeatherReporter will use – where data from the `Uri` that matches the intent filter is parsed and used to display a weather report. This is seen in listing 4.4.

Listing 4.4 The `onStart` method of the WeatherReporter ReportViewDetail Activity, which parses the Intent data to pull location from the a Uri and display a weather report.

```
@Override
public void onStart() {
    super.onStart();
    this.dbHelper = new DBHelper(this);          #1

    this.deviceZip = WeatherAlertService.deviceLocationZIP;    #2

    if ((this.getIntent().getData() != null)
        && (this.getIntent().getData().getEncodedQuery() != null)
        && (this.getIntent().getData().getEncodedQuery().length() > 8)) {
        String queryString = this.getIntent().getData().getEncodedQuery();    #3
        this.reportZip = queryString.substring(4, 9);
        this.useDeviceLocation = false;
    } else {
        this.reportZip = this.deviceZip;
        this.useDeviceLocation = true;
    }

    this.savedLocation = this.dbHelper.get(this.reportZip);
    this.deviceAlertEnabledLocation = this.dbHelper.get(DBHelper.DEVICE_ALERT_ENABLED_ZIP);

    if (this.useDeviceLocation) {          #4
        this.currentCheck.setText(R.string.view_checkbox_current);
        if (this.deviceAlertEnabledLocation != null) {
            this.currentCheck.setChecked(true);
        } else {
            this.currentCheck.setChecked(false);
        }
    } else {          #4
        this.currentCheck.setText(R.string.view_checkbox_specific);
        if (this.savedLocation != null) {
            if (this.savedLocation.alertenabled == 1) {
                this.currentCheck.setChecked(true);
            } else {
                this.currentCheck.setChecked(false);
            }
        }
    }
    this.loadReport(this.reportZip);    #5
}
```

1. Establish database helper object
2. Get the device location postal code
3. Parse intent data
4. Set the status of the alert enabled or disabled checkbox
5. Load weather report for specified postal code

The complete `ReportViewDetail` Activity can be obtained by grabbing the source code in it's entirety from the Manning web site. In the portion of the class we see in listing 4.4, the `onStart` method, we are focusing on parsing data from the `Uri` passed in as part of the `Intent` that invokes the Activity.

First in this class snippet we see that we are establishing a database helper object **#1**. This will be used to query a local SQLite database that stores user specified location data. We will see more about how data is handled in general, and the details of this helper class, in chapter 5.

In this method we are also obtaining the postal code of the current device location from a `LocationManager` in the `WeatherAlertService` class (currently 94102, San Francisco, CA) **#2**. This is

significant for a few reasons. First it's important to understand that we want our application to be location aware. We want the location of the device (wherever it is) to be the default weather report and alert location. As the user travels with the phone, this location should automatically be updated. We will cover more about location, and `LocationManager`, in chapter 12 - for now, note that the device location is returned to us here as a postal code.

After obtaining the device location we then move on to the key aspect of obtaining `Uri` data from an `Intent`. We are parsing the `Uri` passed in to obtain the `queryString` and embedded postal code to use for the users specified location **#3**. If this location is present, we use it, if not we default to the device location postal code.

Once we have determined the postal code to use, we then move on to set the status of the checkbox that indicates whether or not alerts should be enabled for the location being displayed **#4**. We have two kinds of alerts, one for the device location (wherever that location may be at a given time), and another for the users specified saved locations.

Finally we call the `loadReport` method, which is used to make the call out to the Yahoo! Weather API to obtain data, and then use a `Handler` to send a `Message` to update the needed UI `View` elements **#5**. These details are not shown in this code portion, because we are focusing on `Intent` handling in this section, but the pattern is the same one we have seen in previous listings.

The key with this `Activity` is the way it is registered in the manifest to receive `weather://com.msi.manning` intents, and then parses the path of the URI for data. This allows any application to invoke this `Activity` without knowing any details other than the URI. This is the separation of responsibilities pattern the Android platform design encourages at work (the late binding).

Now that we have we have seen the manifest and pertinent details of the main `Activity` class for the WeatherReporter application we will be building in the next few sections, and we have covered a good bit about how intents and intent filters work together to wire up calls between components in general, we will next take a brief look at some of the built in Android applications that work the same way, and enable you to launch various activities by simply passing in the correct URI.

4.1.4 Using Android Provided Activities

Another way to get a feel for how `Intent` resolution works in Android, and how URIs are used, is to explore the built in `Activity` support. Android ships with a very useful set of core applications that provide access via the formats shown in table 4.2.

Table 4.2 Common Android application Intent action and Uri combinations, and purpose.

Action	Uri	Description
Intent.ACTION_VIEW	geo:latitude,longitude	Open the maps application to the specified latitude and longitude
Intent.ACTION_VIEW	geo:0,0?q=street+address	Open the maps application to the specified address
Intent.ACTION_CALL	tel:phone_number	Open the phone application and call the specified number
Intent.ACTION_DIAL	tel:phone_number	Open the phone application and dial (but not call) the specified number
Intent.ACTION_DIAL	voicemail:	Open the phone application and dial (but not call) the voicemail number
Intent.ACTION_VIEW	http://web_address	Open the browser application to the specified URL
Intent.ACTION_VIEW	https://web_address	Open the browser application to the specified URL
Intent.ACTION_WEB_SEARCH	plain_text	Open the browser application and use Google search

Using the actions and URIs shown in table 4.2 you can hook into the built in map application, phone application, or browser application. These are powerful applications that are very easy to invoke using the correct `Intent`. You may recall that we used several of these in the last chapter with our `RestaurantFinder` application. Android also includes support for another construct, the `ContentProvider`, which also uses a form of a URI to provide access to data. This system, which is what the contacts and media parts of the Android system are exposed via, for instance, is something we will learn more about in chapter 4.

By comparing the actions and URIs for the built in Android applications you can get a feel for the fact that some applications use a `Uri` that is parsed into a type (contacts, media), and others use the scheme, or scheme and authority, or scheme and authority and path – the various ways to match data as we saw in section 4.1.2.

With a handle on the basics of resolution, and a quick look at built in intents out of the way, we now need to get back to our `WeatherReporter` sample application where the next thing we will come to is another usage for the `Intent` concept - namely, using intent receivers.

4.2 Listening in with `BroadcastReceivers`

Another way to use an `Intent` involves sending a broadcast to any interested “receiver.” There are many reasons an application may want to broadcast an event, for example, when an incoming phone call or text message is received. In this section we will take a look at how events are broadcast, and how they are captured using an `BroadcastReceiver`.

Here we will continue working through the `WeatherReporter` sample application we began in the previous section. One of the most important parts of the `WeatherReporter` application will be its ability to display alerts to the user when severe weather is in the forecast for a location that the user has indicated they are interested in. To enable this to happen we will need a background process running that checks the weather and sends any needed alerts. This is where the Android `Service` concept will come into play. We won't be creating the actual `Service` class until section 4.3, but we need a way to get the platform running the `Service` as soon as it boots up, and this is where we will use an `Intent` broadcast.

4.2.1 Overloading the `Intent` Concept

As we have seen `Intent` objects are used to go from `Activity` to `Activity` inside an Android application. While this is the main usage of intents in Android, it is not the only one. Intents are also used to broadcast events to any configured receiver using any one of the methods on the `Context` class that are shown in table 4.3.

Table 4.3 Methods for broadcasting Intents.

Method	Description
<code>sendBroadcast(Intent intent)</code>	Simple form for broadcasting an <code>Intent</code>
<code>sendBroadcast(Intent intent, String receiverPermission)</code>	Broadcast an <code>Intent</code> with a permission <code>String</code> that receivers must declare to receive the broadcast.
<code>sendStickyBroadcast(Intent intent)</code>	Broadcast an <code>Intent</code> that hangs around a short time after it is sent so that receivers can retrieve data. Applications using this must declare the <code>BROADCAST_STICKY</code> permission.
<code>sendOrderedBroadcast(Intent intent, String receiverPermission)</code>	Broadcast an <code>Intent</code> call the receivers one by one serially.
<code>sendOrderedBroadcast(Intent intent, String receiverPermission, BroadcastReceiver resultReceiver, Handler scheduler, int requestCode, String initialData, Bundle initialExtras)</code>	Broadcast an <code>Intent</code> and get a response back by implementing your own <code>BroadcastReceiver</code> for the broadcast (and passing it in). All receivers can append data that will be returned in the <code>BroadcastReceiver</code> . When using this method the

When broadcasting intents you are basically re-using the `Intent` concept to send an event in the background. Though the `Intent` class is used, it is used differently than when invoking foreground `Activity` paths. A broadcast `Intent` does not invoke an `Activity` (though an `BroadcastReceiver` can after the event is received, if necessary).

Another important aspect with broadcast intents is how permissions are handled. When you broadcast an `Intent` you can optionally specify a permission. Permissions are something we will get into in more detail in chapter 9, but basically they are `String` declarations that can be used when making a broadcast that require receivers to declare the same permission.

Broadcasting an `Intent` itself is therefore fairly straightforward, you use the `Context` object to throw it on the wire, and interested receivers catch it. Android provides a set of platform related `Intent` broadcasts that use this approach. When the time zone on the platform changes, when the device completes booting, or when a package is added or removed, for example, the system broadcasts an event using an `Intent`. Some of the specific intent broadcasts the platform provides are shown in table 4.4.

Table 4.4 Provided Android platform broadcast actions.

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

The other half of broadcasting events is the receiving end. To register to receive an `Intent` broadcast, you implement an `BroadcastReceiver`. This is where we are going to implement a receiver that will catch the platform provided `BOOT_COMPLETED` `Intent` in order to then start the weather alert service we will create for the `WeatherReporter` application.

4.2.2 Creating a Receiver

Because the weather alert `Service` we want to create needs to be running in the background whenever the platform itself is running, we need a way to start it when the platform boots. To do this we will create a `BroadcastReceiver` that listens for the `BOOT_COMPLETED` intent broadcast.

The `BroadcastReceiver` base class provides a series of methods that allow for getting and setting a result code, result data (in the form of a `String`), and an extras `Bundle`. In addition there are a series of life-cycle related methods that correspond to the life-cycle events of a receiver – we will learn more about these as we progress through this section.

Associating a `BroadcastReceiver` with an `IntentFilter` can be done in code or in the manifest XML file. Once again the XML usage is often easier, and thus more common. This is the way we did it for `WeatherReporter` in listing 4.3, where we associated the `BOOT_COMPLETED` broadcast with the `WeatherAlertServiceReceiver` class. This class itself is shown in listing 4.5.

Listing 4.5 The `WeatherAlertServiceReceiver` `BroadcastReceiver` class, which starts the weather alert `Service`.

```
public class WeatherAlertServiceReceiver extends BroadcastReceiver {
    Please post comments or corrections to the Author Online forum at #1
    http://www.manning-sandbox.com/forum.jspa?forumID=411
    Licensed to Thow Way Chiam <ken.ctw@gmail.com>
```



```

@Override
public void onReceive(Context context, Intent intent) { #2
    if (intent.getAction().equals(Intent.ACTION_BOOT_COMPLETED)) {
        context.startService(new Intent(context, WeatherAlertService.class)); #3
    }
}
}

```

1. Extend BroadcastReceiver
2. Implement the onReceive method
3. Start the WeatherAlertService

When creating your own intent receiver you extend the `BroadcastReceiver` class Android provides **#1**, and implement the the abstract `onReceive(Context c, Intent i)` method **#2**. Within this method we see that we are starting the `WeatherAlertService`. This `Service` class, which we will create next, is started using the `Context.startService(Intent i, Bundle b)` method **#3**.

An important thing to keep in mind with regard to receiver class instances is that they have a very short, specific, life-cycle. When the `onReceive(Context c, Intent i)` method is complete, the instance and process that invoked the receiver is no longer needed and may be killed by the system. Because of this, you can't perform any asynchronous operations in a `BroadcastReceiver`, such as binding to a `Service`, or showing a dialog. Alternatively you can start a `Service`, as we have done here, and leave it running in the background (binding to a `Service` is different than starting one, this is a distinction we will cover in the next section).

Now that our receiver is starting the `WeatherAlertService`, which will run in the background and warn users of severe weather in the forecast with a `Notification` based alert, we next need to delve into the realm of the Android `Service` concept itself, where we will implement this class.

4.3 Building a Service

In the typical Android application you create `Activity` classes and move from screen to screen using `Intent` calls. This is the approach we introduced in chapter 1, and used in other previous chapters. This works for the canonical Android screen-to-screen foreground application, but is not applicable for a longer running background process – for that you need a `Service`.

The `Service` we will work with here is the `WeatherAlertService` we sent an `Intent` request for in the `WeatherAlertServiceReceiver` in listing 4.4. One of the things this `Service` does is to send an alert to the user when there is severe weather in a location they have indicated they are interested in. This alert will be displayed in any application, in the form of a **Notification**, by the background `Service` if severe weather is detected. The notifications we will send are seen in the screen shot in figure 4.5.



Figure 4.5 Screen shot of the Notification based alert the WeatherAlertService displays to the user when severe weather is detected in the forecast.

One key aspect of Android services that we need to cover prior to jumping in and implementing one, is their dual-purpose nature. Something like the duality of man (you know, the “Jungian Thing”), services lead a double life.

4.3.1 Dual-purpose nature of a Service

In Android a Service is intended to serve two purposes: running a background task, or exposing a remotable object for Inter-Process Communication (IPC). We will explore both of these purposes for a Service in turn. Importantly, though we are going to build separate Service instances for each purpose, you can also build one Service that serves both purposes – if needed.

A background task is typically a long running process that does involve direct user interaction or any type of UI. This of course is a perfect fit for polling for severe weather in the background. As far as exposing a remotable object for IPC, we will see how that works, and why it is necessary, in section 4.4.1. There we will build another Service that walks through creating and exposing a remotable object.

As we have already discussed briefly, and we will learn more about here as we go, a Service can either be started, or bound, or both. Starting a Service relates to the background task aspect. Once started a Service runs until it is explicitly stopped (we will learn more about this in section 4.4, where we discuss the overall life-cycle of a Service). Binding to a Service involves using a ServiceConnection object to connect and get a remotable reference.

Creating the WeatherAlertService itself, which serves the first type of Service purpose and enables our background weather checks, is where we will focus next.

4.3.2 Creating a background task Service

The WeatherAlertService background task focused Service is shown in listing 4.6.

Listing 4.6 The WeatherAlertService class, which is used to register locations for alerts, and send alerts.

```
public class WeatherAlertService extends Service { #1
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

private static final String LOC = "LOC";
private static final String ZIP = "ZIP";
private static final long ALERT_QUIET_PERIOD = 10000;      #|2
private static final long ALERT_POLL_INTERVAL = 15000;     #|2

public static String deviceLocationZIP = "94102";

private Timer timer;
private DBHelper dbHelper;
private NotificationManager nm;

TimerTask task = new TimerTask() {
    public void run() {
        List<Location> locations = dbHelper.getAllAlertEnabled();      #3
        for (Location loc : locations) {
            WeatherRecord record = loadRecord(loc.zip);
            if (record.isSevere()) {                                     #4
                if ((loc.lastalert + ALERT_QUIET_PERIOD) < System.currentTimeMillis()) {
                    loc.lastalert = System.currentTimeMillis();
                    dbHelper.update(loc);
                    sendNotification(loc.zip, record);
                }
            }
        }

        Location deviceAlertEnabledLoc = dbHelper.get(DBHelper.DEVICE_ALERT_ENABLED_ZIP);
        if (deviceAlertEnabledLoc != null) {                             #5
            WeatherRecord record = loadRecord(deviceLocationZIP);
            if (record.isSevere()) {
                if ((deviceAlertEnabledLoc.lastalert + ALERT_QUIET_PERIOD) <
System.currentTimeMillis()) {
                    deviceAlertEnabledLoc.lastalert = System.currentTimeMillis();
                    dbHelper.update(deviceAlertEnabledLoc);
                    sendNotification(deviceLocationZIP, record);
                }
            }
        }
    }
};

private Handler handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        notifyFromHandler((String) msg.getData().get(LOC), (String) msg.getData().get(ZIP)); #6
    }
};

@Override
public void onCreate() {
    dbHelper = new DBHelper(this);      #7
    timer = new Timer();
    timer.schedule(task, 5000, ALERT_POLL_INTERVAL);
    nm = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);
}

@Override
public void onStart(Intent intent, int startId) {
    super.onStart(intent, startId);

    . . . LocationManager code omitted here for brevity (updates deviceLocationZIP)
}

@Override
public void onDestroy() {
    super.onDestroy();
    this.dbHelper.cleanup();            #8
}

@Override
public IBinder onBind(Intent intent) {
    return null;                        #9
}

private WeatherRecord loadRecord(String zip) {
    final YWeatherFetcher ywh = new YWeatherFetcher(zip, true);
    return ywh.getWeather();            #10
}

private void sendNotification(String zip, WeatherRecord record) {      #11
    Message message = Message.obtain();
    Bundle bundle = new Bundle();
    bundle.putString(LOC, zip);
    bundle.putParcelable(WR, record);
    nm.send(message, 0, bundle);
}

```

```

        bundle.putString(ZIP, zip);
        bundle.putString(LOC, record.getCity() + ", " + record.getRegion());
        message.setData(bundle);
        handler.sendMessage(message);
    }

    private void notifyFromHandler(String location, String zip) { #12
        Uri uri = Uri.parse("weather://com.msi.manning/loc?zip=" + zip);
        Intent intent = new Intent(Intent.ACTION_VIEW, uri);
        PendingIntent pendingIntent = PendingIntent
            .getActivity(this, Intent.FLAG_ACTIVITY_NEW_TASK, intent,
                PendingIntent.FLAG_ONE_SHOT);
        final Notification n = new Notification(R.drawable.severe_weather_24,
            "Severe Weather Alert!", System.currentTimeMillis());
        n.setLatestEventInfo(this, "Severe Weather Alert!", location, pendingIntent);
        nm.notify(Integer.parseInt(zip), n);
    }
}

```

1. Extend Service
2. Define constants for polling intervals
3. Use database to get all locations with alerts enabled
4. Load weather for each location and fire alert if severe
5. Also check the if device location alerts should be sent
6. Call notify method from handler, using Message data
7. Inside onCreate method setup database and notification manager
8. Inside onDestroy cleanup database connection
9. Return null from the onBind method
10. Load a weather record using the YWeatherFetcher
11. Include helper method to create and send a message to a handler
12. Include helper method to create notification and fire it

The first thing of note in the `WeatherAlertService` class is the fact that it extends `Service` **#1**. This is the same approach we have seen with activities and receivers; extend the base class, implement the abstract methods, and override the life-cycle methods as needed.

After the initial class declaration we then have a series of member variables that are defined. The first of these are constants that represent intervals for polling for severe weather, and a quiet period **#2**. These are significant because we have set a very low threshold for polling during development – severe weather alerts will spam the emulator often because of this setting. In production this would be throttled back to once every 6 or 12 hours so such.

Next we have a `TimerTask` variable that we will use to do the polling and get all of the user's saved locations that have alerting enabled, through a database call **#3**. We will learn the specifics of using a database in Android in the next chapter, where we will finish out the `WeatherReporter` application and focus on data.

Once we have the saved locations, we then parse each one and load the weather report. If the report shows severe weather in the forecast we update the time of the last alert field and call a helper method to initiate a notification being sent **#4**. After we process the user's saved locations we then get the device alert location from the database using a special postal code designation **#5**. The process of polling and sending an alert is repeated for the device current location – as opposed to saved specific locations – if the user has this feature enabled. The device location itself is obtained via a `LocationManager`. We have omitted this code here so that we can focus on the “service” concept – complete details on location facilities will be covered in chapter 9.

After our `TimerTask` is setup we then have a `Handler` member variable **#6**. This variable will be used later, using the same technique we have seen in previous listings, to receive a `Message` object that is fired from a non UI related thread, and then react. In this case when the message is received we call a helper method that instantiates and displays a `Notification`.

Beyond our member variables we then come to the `Service` life-cycle methods that we have implemented, starting with `onCreate` **#7**. Inside this method we setup our database helper object, and a `NotificationManager`. Again, we will cover data in the next chapter, and alert and notification details we be specifically

addressed in chapter 8. After `onCreate` we also implement `onDestroy`, which is where we cleanup our database connection #8. Service classes have these life-cycle methods so you can control how resources are allocated and de-allocated, similarly to Activity classes, in section 4.4.5 we will address this in more depth.

After the life-cycle related methods we then implement the required `onBind` method #9. This method returns an `IBinder`, which is generally what other components that call into Service methods use for communication. Service classes, as we discussed in section 4.3.1, can serve two purposes: first to run background processes, and second for binding to enable Inter-Process Communication (IPC). Our weather alert service is only performing a background task, not enabling `IBinder/Binder` based IPC. Therefore, this class just returns a `null` for `onBind`. The binding and IPC aspect of a Service is something we will delve into in section 4.4.

Next we see the implementations of our own helper type methods. First we have `loadRecord`, which is where we actually call out to the Yahoo! Weather API via `YWeatherFetcher` #10. Then we have `sendNotification` which sets up a `Message` with location details to pass into our earlier declared `Handler` #11. The way this method uses the handler ensures that processing time to get weather data doesn't hang the main UI thread. Lastly we see the `notifyFromHandler` method that is invoked from the `Handler`, this fires off a `Notification` with `Intent` objects that will call back into `WeatherReporter` if the user clicks on the notification #12.

Now that we have discussed what services are for, have created a Service class, and have previously seen a service “started” via an `BroadcastReceiver`, we next need to cover a bit more detail about the IPC process in Android, and other Service details related to it such as starting versus binding, and life-cycle.

4.4 Performing Inter-Process Communication

Communication between application components in different processes is made possible in Android by a specific IPC approach. This, again, is necessary because each application on the platform runs in its own process, and processes are intentionally separated from one another. In order to pass messages and objects between processes, you have to use the Android IPC path.

To begin exploring this path we are first going to build a small focused sample application to take a look at the means to generate a remote interface using AIDL, and then we will connect to that interface through a proxy that we will expose using a Service (the other Service purpose). Along the way we will expand on the `IBinder` and `Binder` concepts Android uses to pass messages and types during IPC.

4.4.1 Android Interface Definition Language (AIDL)

Android provides its own “interface definition language” that you can use to create IDL files. These files then become the input to the “aidl” tool, which Android also includes. This tool is used to generate a Java interface and inner `Stub` class that you can, in turn, use to create a remotely accessible object.

AIDL files have a specific syntax that allows you to define methods, with return types and parameters (you cannot define static fields, unlike with a typical Java interface). In the basic AIDL syntax you define your package, imports, and interface just like you would in Java, as shown in listing 4.7.

Listing 4.7 The An example .aidl remote interface definition language file showing interface and method definitions.

```
package com.msi.manning.binder;           #1

interface ISimpleMathService {             #2
    int add(int a, int b);                 #|3
    int subtract(int a, int b);             #|3
    String echo(in String input);           #|3
}
```

1. Define the package
2. Declare the interface name

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
Licensed to Thow Way Chiam <ken.ctw@gmail.com>

3. Describe a method

The package **#1**, import statements (of which we have none here), and interface **#2** constructs in AIDL are straightforward – they are analogous to regular Java. When you define methods though, notice that you must specify a directional tag for all non primitive types with each parameter (**in**, **out**, or **inout**). Primitives are only allowed as **in**, and are therefore treated as **in** by default (and thus don't need the tag). This directional tag is used by the platform to generate the necessary code for marshalling and unmarshalling instances of your interface across IPC boundaries. It's better to only go one direction where you can, for performance reasons, so try to use only what you really need.

In this case we have declared an interface named `ISimpleMathService` that includes methods which perform addition, subtraction, and echoing a `String`. This is an oversimplified example of course, but it does demonstrate the approach.

When using AIDL you also have to be aware that only certain types are allowed, these types are shown in table 4.5.

Table 4.5 Android IDL allowed types.

Type	Description	Import Required
Java primitives	boolean, byte, short, int, float, double, etc.	No
String	<code>java.lang.String</code>	No
CharSequence	<code>java.lang.CharSequence</code>	No
List	Can be generic, all types used in collection must be one of IDL allowed. Ultimately implemented as an <code>ArrayList</code> .	No
Map	Can be generic, all types used in collection must be one of IDL allowed. Ultimately implemented as a <code>HashMap</code> .	No
Other AIDL interfaces	Any other AIDL generated interface type	Yes
Parcelable objects	Objects that implement the Android Parcelable interface (more about this in section 4.4.3)	Yes

Once you have defined your interface methods, with return types, and parameters with directional tags, in the AIDL format, you then invoke the “aidl” tool to generate a Java interface that represents your AIDL specification. From the command line you can invoke `[ANDROID_HOME]/tools/aidl` to see the options and syntax for the aidl tool. Generally you just need to point it at your AIDL file, and it will emit a Java interface of the same name. If you use the Eclipse plugin it will automatically invoke the aidl tool for you (it recognizes .aidl files, and invokes the tool).

The interface that gets generated through AIDL includes an inner static abstract class named `Stub` that extends `Binder`, and implements the outer class interface. This `Stub` class represents the **local** side of your remotable interface. `Stub` also includes an `asInterface(IBinder binder)` method that returns a **remote** version of your interface type. Callers can use this method to get a handle on the remote object, and from there invoke remote methods. The AIDL process generates a `Proxy` class (another inner class, this time inside of `Stub`), that is used to wire up the plumbing and return to callers from the `asInterface` method. The diagram in figure 4.6 depicts this IPC local/remote relationship.

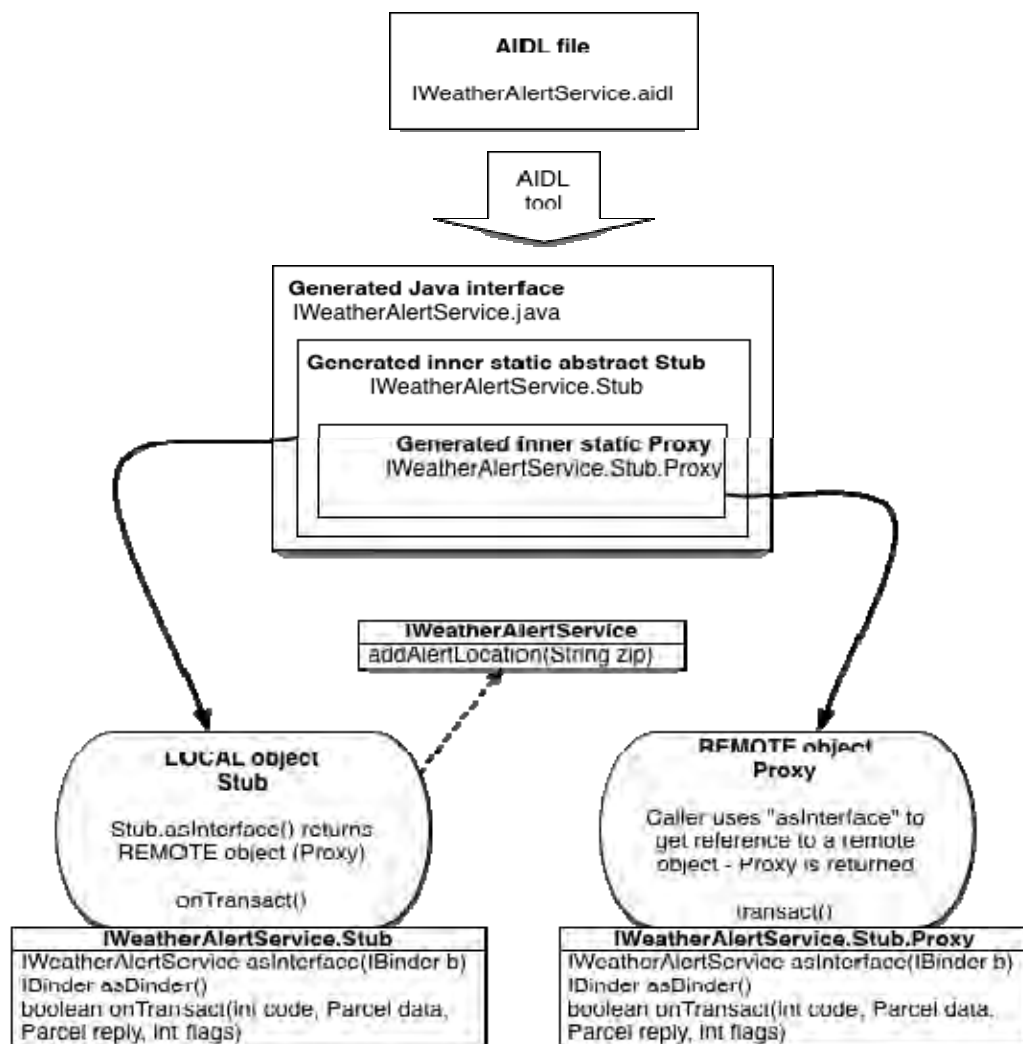


Figure 4.6 Overview diagram of the Android AIDL process.

Once you have all of the generated parts involved, you then create a concrete class that extends from `Stub` and implements your interface. You then expose this interface to callers through a `Service`.

4.4.2 Exposing a remote interface

The glue in all of the moving parts of AIDL that we have discussed up to now is the point where a remote interface is exposed – via a `Service`. In Android parlance exposing a remote interface through a `Service` is known as “publishing.”

To publish a remote interface you create a class that extends `Service`, and returns an `IBinder` through the `onBind(Intent intent)` method within. The `IBinder` that you return here is what clients will use to access a particular remote object. As we discussed in the previous section, the AIDL generated `Stub` class (which itself extends `Binder`) is usually used, to extend from, and return an implementation of a remotable interface. This is usually what is returned from a `Service` class's `onBind` method – and hence this is how a remote interface is exposed to any other process that can bind to a `Service`. All of this is seen in listing 4.8, where we implement and publish the `ISimpleMathService` we created in the previous section.

Listing 4.8 A Service Implementation that exposes an IBinder remotable object.

```
public class SimpleMathService extends Service {

    private final ISimpleMathService.Stub binder = new ISimpleMathService.Stub() {    #1
        public int add(int a, int b) {
            return a + b;
        }
        public int subtract(int a, int b) {
            return a - b;
        }
        public String echo (String input) {
            return "echo " + input;
        }
    };

    @Override
    public IBinder onBind(Intent intent) {    #2
        return this.binder;
    }
}
```

1. Implement the remote interface
2. Return an IBinder representing the remotable object

A concrete instance of the generated AIDL Java interface is required to return an `IBinder` to any caller than binds to a `Service`. The way to create an implementation is to implement the `Stub` class that the `aidl` tool generates #1. This class, again, implements the AIDL interface, and extends `Binder`. Once the `IBinder` is established, it is then simply returned from the `onBind` method.

Now that we have seen where a caller can hook into a `Service` and get a reference to a remotable object, we next need to walk through finishing that connection by binding to a `Service` from an `Activity`.

4.4.3 Binding to a Service

When an `Activity` class binds to a `Service`, which is done using `Context.bindService(Intent i, ServiceConnection connection, int flags)` method. The `ServiceConnection` object that is passed in is used to send several callbacks, from the `Service` back to the `Activity`. One significant callback happens when the binding process completes. This callback comes in the form of the `onServiceConnected(ComponentName className, IBinder binder)` method. The platform automatically injects the `IBinder` `onBind` result (from the `Service` being bound to) into this method, and hence makes this object available to the caller. We see how this works in code in listing 4.9.

Listing 4.9 An example Activity that binds to a Service to get a reference to, and utilize, a remotable object.

```
public class ActivityExample extends Activity {

    private ISimpleMathService service;    #1
    private boolean bound;    #2

    private ServiceConnection connection = new ServiceConnection() {    #3
        public void onServiceConnected(ComponentName className, IBinder iservice) {    #4
            service = ISimpleMathService.Stub.asInterface(iservice);    #5
            Toast.makeText(ActivityExample.this,
                "connected to Service", Toast.LENGTH_SHORT).show();
            bound = true;
        }
        public void onServiceDisconnected(ComponentName className) {    #6
            service = null;
            Toast.makeText(ActivityExample.this,
                "disconnected from Service", Toast.LENGTH_SHORT).show();
            bound = false;
        }
    };

    private EditText inputa;
    private EditText inputb;
    private TextView output;
    private Button addButton;
    private Button subtractButton;
```

```

private Button echoButton;

@Override
public void onCreate(Bundle savedInstanceState) {
    . . . View element inflation and Button click listeners omitted for brevity
}

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

    if (!bound) {
        this.bindService(
            new Intent(ActivityExample.this, SimpleMathService.class),
            connection,
            Context.BIND_AUTO_CREATE);
    }
}

@Override
public void onPause() {
    if (bound) {
        bound = false;
        this.unbindService(connection);
    }
}
}

```

1. Define a variable referring to the remote interface type
2. Define a boolean to keep track of bound state
3. Include a ServiceConnection implementation
4. React to the onServiceConnected callback
5. Establish the remote interface type using asInterface
6. React to the onServiceDisconnected callback
7. Use the remote object to perform operations
8. Perform binding in onStart
9. Perform unbinding in onPause

In order to use the remotable `ISimpleMathService` we defined in AIDL we declare a variable of the generated Java interface type **#1**. Along with this service variable, we also include a boolean to keep track of the current state of the binding **#2**.

Essential to the binding process, we next see the `ServiceConnection` object **#3**. This object is used with `Context` methods to bind and unbind. When a `Service` is bound the `onServiceConnected` callback is fired **#4**. Within this callback the remote `IBinder` reference is returned and can be assigned to the remotable type **#5**. After the connection related callback there is a similar `onServiceDisconnected` callback that is fired when a `Service` is unbound **#6**.

Once the connection is established and the remote `IBinder` is in place it then can be used to perform the operations it defines **#7**. Here we are using the `add`, `subtract`, and `echo` methods we created in AIDL in listing 4.7.

Lastly with this class we see the `Activity` life-cycle methods that are now familiar. In `onStart` we establish the binding using `bindService` **#8**, and in `onPause` we use `unbindService` **#9**. A `Service` that is bound, but not started, can itself be cleaned up by the system to free up resources. If we don't unbind these resources might unnecessarily hang around.

A `Service`, as we have seen, and we will learn more about next, is invoked using an `Intent`. Here again explicit or implicit intent invocation can be used. Significantly, any application (with the correct permissions) can all into a `Service` and bind to it, returning the `IBinder` to perform operations – it need not be an `Activity` in the same application as the `Service` (this is how applications in different processes communicate).

That brings us to the difference between starting a `Service`, and binding to one, and what the implications are for each usage.

4.4.4 Starting versus Binding

Again, services serve two purposes in Android, and you can use them as we have now seen in two corresponding ways:

- Starting – `Context.startService(Intent service, Bundle b)`
- Binding – `Context.bindService(Intent service, ServiceConnection c, int flag)`

Starting a `Service` tells the platform to launch it in the background and keep it running, without any particular connection to any other `Activity` or application. We used the `WeatherReportService` in this manner to run in the background and issue severe weather alerts.

Binding to a `Service`, as we did with our sample `SimpleMathService`, is how you get a handle to a remote object and call methods defined there from an `Activity`. As we have discussed, because every Android application is running in its own process, using a bound `Service` (which returns an `IBinder` through `ServiceConnection`) is how you pass data between processes.

Marshalling and unmarshalling remotable objects across process boundaries is fairly complicated. This is the reason the Android IDL process has so many moving parts. Fortunately you don't generally have to deal with all of the internals, you can instead stick to a simple recipe that will enable you to create and use remotable objects.

1. Define your interface using AIDL, in the form of an `[INTERFACE_NAME].aidl` file – listing 4.7
2. Generate a Java interface for your `.aidl` file (automatic in Eclipse)
3. Extend from the generated `[INTERFACE_NAME].Stub` class and implement your interface methods – listing 4.8
4. Expose your interface to clients through a `Service`, and the `Service onBind(Intent i)` method – listing 4.8
5. Bind to your `Service` with a `ServiceConnection` to get a handle to the remotable object, and use it – listing 4.9

Another important aspect of the `Service` concept to be aware of, and one that is affected by whether or not a `Service` is bound or started or both, is the life-cycle.

4.4.5 Service Life-Cycle

Along with overall application life-cycle, that we were introduced to in chapter 2, and `Activity` life-cycle that we saw in detail in chapter 3, services also have their own well defined process phases. Which parts of the `Service` life-cycle that are invoked is affected by how the `Service` is being used: started, bound, or both.

SERVICE STARTED LIFE-CYCLE

If a `Service` is started by `Context.startService(Intent service, Bundle b)`, as saw in listing 4.5, then it runs in the background whether or not anything is bound to it. In this case if it is needed the `Service onCreate()` method will be called, and then the `onStart(int id, Bundle args)` method will be called. If a `Service` is started more than once, the `onStart(int id, Bundle args)` method will be called multiple times, but additional instances of the `Service` will not be created (still needs only one stop call).

The `Service` will continue to run in the background until it is explicitly stopped by the `Context.stopService()` method, or its own `stopSelf()` method (unless the platform runs out of memory and needs to start paging, then it may be stopped by the platform in extreme cases).

SERVICE BOUND LIFE-CYCLE

If a `Service` is bound by an `Activity` calling `Context.bindService(Intent service, ServiceConnection connection, int flags)`, as we saw in listing 4.9, then it will run as long as the connection is established. An `Activity` establishes the connection using the `Context`, and is responsible for closing it as well.

When a `Service` is only bound in this manner, and not also started, its `onCreate()` method is invoked, but `onStart(int id, Bundle args)` is **not** used. In these cases the `Service` is eligible to be stopped and cleaned up by the platform when no longer bound.

SERVICE STARTED AND BOUND LIFE-CYCLE

If a `Service` is bound started and bound, which is allowable, then it will basically keep running in the background, similarly to the started life-cycle. The only real difference is the life-cycle itself. Because of the starting and binding both `onStart(int id, Bundle args)` and `onCreate()` will be called.

CLEANING UP WHEN A SERVICE STOPS

When a `Service` is stopped, either explicitly after having been started, or implicitly when there are no more bound connections (and it was not started), then the `onDestroy()` method is invoked. Inside of `onDestroy()` every `Service` should perform final clean up, stopping any spawned threads and the like.

Now that we have seen how a `Service` is implemented, how one can be used both in terms of starting and binding, and what the life-cycle looks like, we next need to take a closer look at some of the details of remotable data types when using Android IPC and IDL.

4.4.6 Binder and Parcelable

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

The key to the `IBinder` and `Binder` enabling IPC, once the interfaces are defined and implemented, is the `IBinder.transact()` method, and corresponding `Binder.onTransact()` method. Though you don't typically work with these internal methods directly, they are the backbone of the remoting process. Each method you define using AIDL, is then handled synchronously through the transaction process (enabling the same semantics as if the method were local).

All of the objects you pass in and out, through the interface methods you define using AIDL, use the `transact` process. These objects must be `Parcelable`, in order to be able to be placed inside of a `Parcel` and moved across the local/remote process barrier in the `Binder` transaction methods.

The only time you need to worry about something being `Parcelable` is when you want to send a custom object through Android IPC. If you use the default allowable types in your interface definition files: `primitives`, `String`, `CharSequence`, `List`, and `Map` – then everything is automatically handled. If you need to use something beyond those, only then do you need to implement `Parcelable`.

The Android documentation describes what methods you need to implement to create a `Parcelable` class. The only tricky part of doing this is remembering to create an `aidl` file for each `Parcelable` interface. These `aidl` files are different from those you use to define `Binder` classes themselves though, for these you need to remember **not** to generate from using the `aidl` tool. Trying to do so won't work, and isn't intended to. The documentation states these files are used "like a header in C," and so they are not intended to be processed by the `aidl` tool. The big caveat here is that the current Eclipse plugin, which comes in so handy 90% of the time, doesn't understand one `.aidl` file from another and therefore won't work if you need to create your own `Parcelable` types.

Also, when considering creation of your own `Parcelable` types, make sure you really need them. Passing complex objects across the IPC boundary in an embedded environment is an expensive operation, and should be avoided if possible (not to mention that manually creating these types is fairly tedious).

Rounding out our IPC discussion with a quick overview of `Parcelable` completes our tour of Android `Intent` and `Service` usage.

4.5 Summary

In this chapter we covered a broad swath of Android territory. We first focused on the `Intent` abstraction, defining what intents are, how they are resolved using `IntentFilter` objects, and what some built in platform provided intent handlers are. In that discussion we completed the `RestaurantFinder` sample application.

After we covered the basics of intents, we moved on to a new sample application, `WeatherReporter`, where we explored the concept of an `BroadcastReceiver` and an Android `Service`. Along with `Service` implementation details we also delved into the difference between starting and binding services, and the moving parts behind the Android IPC system, which uses the Android IDL process.

Through looking at all these components in several complete examples you should now have a good idea of the basic foundation of these concepts. In the next chapter we will build on this foundation a bit further by looking at the various means Android provides to retrieve and store data, including creating a `ContentProvider`.

5

Storing and Retrieving Data

Anytime you are developing software one of the most common, and most basic, constructs that you have to deal with is the means to store and retrieve data. It's all about the data after all. Though there are many ways to pipe data into and out of various languages and technologies, there are typically only a few ways to persist it: in memory structures, the file system, databases, and network services.

Like other technologies, Android has its own concepts for getting and sharing data in applications, yet these concepts are ultimately implemented using familiar approaches (for the most part). Android provides access to the file system, has support for a local relational database through SQLite, and includes a `SharedPreferences` object and preferences system that allows you to store simple key value pairs within applications.

In this chapter we are going to take a tour of each of the local data related mechanisms (we will examine the network possibilities in chapter 6). We will start with preferences, and create a small sample application to exercise those concepts. From there we will also create another sample application to examine using the file system to store data, both internal to our application, and external using the platform's SD card support. Then, we will look at creating and accessing a database. To do this we will take a closer look at some of the code and concepts from the `WeatherReporter` application we created in chapter 4, which uses SQLite.

Beyond the basics, Android also includes its own construct that allows applications to share data through a clever URI based approach called a `ContentProvider`. This technique combines several other Android concepts, such as the URI based style of intents, and the `Cursor` result set seen in SQLite, to make data accessible across different applications. To see how this works we will create another small sample application that uses some built-in providers, and then we will walk through the steps required to create a `ContentProvider` on our own.

We begin with the easiest form of data storage and retrieval Android provides, preferences.

5.1 Using preferences

When moving from `Activity` to `Activity` in Android it is very handy to be able to save some global application state in a `SharedPreferences` object. Here we will discuss how you can set data into a preferences object, and how you can later retrieve it. Also, we will discuss how to make preferences private to your application, or accessible to other applications on the same device.

5.1.1 Working with `SharedPreferences`

You access a `SharedPreferences` object through the `Context` you are working in. Many Android classes have a reference to, or themselves extend from, and `Context`. For example, `Activity` and `Service` extend `Context`.

`Context` includes a `getSharedPreferences(String name, int accessMode)` method that allows you to get a preferences handle. The name you specify indicates the file that backs the preferences you are interested in. If no such file exists when you try to “get” preferences one is automatically created using the passed in name. The access mode refers to what permissions you want to allow.

Listing 5.1 is an example `Activity` that demonstrates allowing the user to enter input, and then storing that data through `SharedPreferences` objects with different access modes.

Listing 5.1 Storing `SharedPreferences` using different modes.

```
package com.msi.manning.chapter5.prefs;

// imports omitted for brevity

public class SharedPrefTestInput extends Activity {

    public static final String PREFS_PRIVATE = "PREFS_PRIVATE";
    public static final String PREFS_WORLD_READ = "PREFS_WORLD_READABLE";
    public static final String PREFS_WORLD_WRITE = "PREFS_WORLD_WRITABLE";
    public static final String PREFS_WORLD_READ_WRITE = "PREFS_WORLD_READABLE_WRITABLE";

    public static final String KEY_PRIVATE = "KEY_PRIVATE";
    public static final String KEY_WORLD_READ = "KEY_WORLD_READ";
    public static final String KEY_WORLD_WRITE = "KEY_WORLD_WRITE";
    public static final String KEY_WORLD_READ_WRITE = "KEY_WORLD_READ_WRITE";

    private EditText inputPrivate;
    private EditText inputWorldRead;
    private EditText inputWorldWrite;
    private EditText inputWorldReadWrite;
    private Button button;

    private SharedPreferences prefsPrivate;           #|1
    private SharedPreferences prefsWorldRead;        #|1
    private SharedPreferences prefsWorldWrite;       #|1
    private SharedPreferences prefsWorldReadWrite;   #|1

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.shared_pref_test_input);

        this.inputPrivate = (EditText) this.findViewById(R.id.input_private);
        this.inputWorldRead = (EditText) this.findViewById(R.id.input_worldread);
        this.inputWorldWrite = (EditText) this.findViewById(R.id.input_worldwrite);
        this.inputWorldReadWrite = (EditText) this.findViewById(R.id.input_worldreadwrite);
        this.button = (Button) this.findViewById(R.id.prefs_test_button);
        this.button.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {

                boolean valid = SharedPrefTestInput.this.validate();
                if (valid) {
                    SharedPrefTestInput.this.prefsPrivate =
                        SharedPrefTestInput.this.getSharedPreferences(SharedPrefTestInput.PREFS_PRIVATE,
                            Context.MODE_PRIVATE); #2
                    SharedPrefTestInput.this.prefsWorldRead =
                        SharedPrefTestInput.this.getSharedPreferences(
                            SharedPrefTestInput.PREFS_WORLD_READ,
                            Context.MODE_WORLD_READABLE);
                    SharedPrefTestInput.this.prefsWorldWrite =
                        SharedPrefTestInput.this.getSharedPreferences(
                            SharedPrefTestInput.PREFS_WORLD_WRITE,
                            Context.MODE_WORLD_WRITEABLE);
                    SharedPrefTestInput.this.prefsWorldReadWrite =
                        SharedPrefTestInput.this.getSharedPreferences(
                            SharedPrefTestInput.PREFS_WORLD_READ_WRITE, Context.MODE_WORLD_READABLE
                            + Context.MODE_WORLD_WRITEABLE); #3

                    Editor prefsPrivateEditor = SharedPrefTestInput.this.prefsPrivate.edit(); #4
                    Editor prefsWorldReadEditor = SharedPrefTestInput.this.prefsWorldRead.edit();
                    Editor prefsWorldWriteEditor = SharedPrefTestInput.this.prefsWorldWrite.edit();
                }
            }
        });
    }
}
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
Licensed to Thow Way Chiam <ken.ctw@gmail.com>


```

        Editor prefsWorldReadWriteEditor =
SharedPrefTestInput.this.prefsWorldReadWrite.edit();

        prefsPrivateEditor.putString(SharedPrefTestInput.KEY_PRIVATE,
SharedPrefTestInput.this.inputPrivate.getText().toString());      #5
        prefsWorldReadEditor.putString(SharedPrefTestInput.KEY_WORLD_READ,
SharedPrefTestInput.this.inputWorldRead.getText().toString());
        prefsWorldWriteEditor.putString(SharedPrefTestInput.KEY_WORLD_WRITE,
SharedPrefTestInput.this.inputWorldWrite.getText().toString());
        prefsWorldReadWriteEditor.putString(SharedPrefTestInput.KEY_WORLD_READ_WRITE,
SharedPrefTestInput.this.inputWorldReadWrite.getText().toString());

        prefsPrivateEditor.commit();      #6
        prefsWorldReadEditor.commit();
        prefsWorldWriteEditor.commit();
        prefsWorldReadWriteEditor.commit();

        Intent intent = new Intent(SharedPrefTestInput.this, SharedPrefTestOutput.class);
        SharedPrefTestInput.this.startActivity(intent);
    }
}

... validate omitted for brevity
}

```

1. Declare variables for each type of `SharedPreferences` object
2. Use `Context.getSharedPreferences` to assign a reference to each variable
3. Use different modes, including adding modes to combine them
4. Get an `Editor` for each `SharedPreferences` object
5. Use the `Editor` to put a `String` value into the preference
6. Commit the change via the `Editor`

Once you have a `SharedPreferences` variable **#1**, you may then assign a reference through the `Context` **#2**. Note that for each `SharedPreferences` object we are getting, we are using a different constant value for the access mode, and in some cases we are even adding modes (modes are of `int` type) **#3**. Modes specify whether or not the preferences should be private, world readable, world writable, or a combination.

After you have preferences, you can then get an `Editor` handle in order to start manipulating values **#4**. With the `Editor` you can set `String`, `boolean`, `float`, `int`, and `long` types as key-value pairs **#5**. This limited set of types can be restrictive, and it is why we extended the `Context` in chapter 3 to store some application state in the form of a complex object, rather than using preferences. Even with this restriction though, often preferences are adequate, and as you can see they are nice and simple to use.

After you have stored some data with an `Editor`, which creates an in memory `Map`, you have to remember to call `commit()` to persist it to the preferences backing file **#6**. After data is committed, you can get it from a `SharedPreferences` object even easier than storing it. Listing 5.2 is an example `Activity`, from the same application (same package), that gets, and displays, the data that was stored in listing 5.1.

Listing 5.2 Getting `SharedPreferences` data from within the same application after it has been stored.

```

package com.msi.manning.chapter5.prefs;

// imports omitted for brevity

public class SharedPrefTestOutput extends Activity {

    private TextView outputPrivate;
    private TextView outputWorldRead;
    private TextView outputWorldWrite;
    private TextView outputWorldReadWrite;

    private SharedPreferences prefsPrivate;      #1
    private SharedPreferences prefsWorldRead;
    private SharedPreferences prefsWorldWrite;

```

```

private SharedPreferences prefsWorldReadWrite;

@Override
public void onCreate(final Bundle icle) {
    super.onCreate(icle);
    this.setContentView(R.layout.shared_pref_test_output);

    this.outputPrivate = (TextView) this.findViewById(R.id.output_private);
    this.outputWorldRead = (TextView) this.findViewById(R.id.output_worldread);
    this.outputWorldWrite = (TextView) this.findViewById(R.id.output_worldwrite);
    this.outputWorldReadWrite = (TextView) this.findViewById(R.id.output_worldreadwrite);
}

@Override
public void onStart() {
    super.onStart();
    this.prefsPrivate = this.getSharedPreferences(SharedPrefTestInput.PREFS_PRIVATE,
        Context.MODE_PRIVATE);           #2
    this.prefsWorldRead = this.getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_READ,
        Context.MODE_WORLD_READABLE);
    this.prefsWorldWrite = this.getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_WRITE,
        Context.MODE_WORLD_WRITEABLE);
    this.prefsWorldReadWrite = this.getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_READ_WRITE,
        Context.MODE_WORLD_READABLE + Context.MODE_WORLD_WRITEABLE);

    this.outputPrivate.setText(
        this.prefsPrivate.getString(SharedPrefTestInput.KEY_PRIVATE, "NA"));   #3
    this.outputWorldRead.setText(
        this.prefsWorldRead.getString(SharedPrefTestInput.KEY_WORLD_READ, "NA"));
    this.outputWorldWrite.setText(
        this.prefsWorldWrite.getString(SharedPrefTestInput.KEY_WORLD_WRITE, "NA"));
    this.outputWorldReadWrite.setText(
        this.prefsWorldReadWrite.getString(SharedPrefTestInput.KEY_WORLD_READ_WRITE, "NA"));
}
}

```

1. Declare a `SharedPreferences` variable
2. Assign `SharedPreferences` variable using `getSharedPreferences()`
3. Get a value from a `SharedPreferences` reference

To get `SharedPreferences` values that we have previously stored, we again declare variables **#1**, and assign references **#2**. Once these are in place we can simply get values using methods such as `getString(String key, String default)` **#3**.

So, as you can see, setting and getting preferences is very straightforward. The only potential flies in the ointment are the access modes, which we will focus on next.

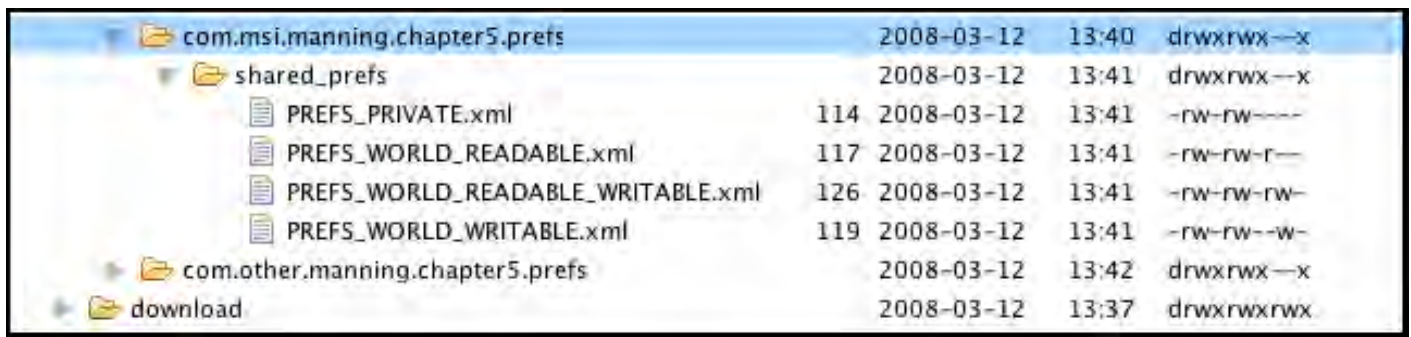
5.1.2 Preference access permissions

`SharedPreferences` can be opened or created with any combination of several `Context` mode constants. Because these values are `int` types, they can be added together, as we did in listings 5.1 and 5.2, to combine permissions. The supported mode constants are as follows:

- `Context.MODE_PRIVATE` (value 0)
- `Context.MODE_WORLD_READABLE` (value 1)
- `Context.MODE_WORLD_WRITEABLE` (value 2)

These modes are handy, allowing you to finely tune who has access to what preference. If we take a look at the file system on the emulator, after having creating `SharedPreferences` objects (which themselves create XML files to persist the data), we can see how this works using the Linux file system.

Figure 5.1 is a screen shot of the Android Eclipse plug-in File Explorer view, it shows the Linux level permissions for the `SharedPreferences` XML files that were created in listing 5.1 (these were automatically created for us when we used `SharedPreferences`).



File/Directory	Size	Date	Time	Permissions
com.msi.manning.chapter5.prefs		2008-03-12	13:40	drwxrwx--x
shared_prefs		2008-03-12	13:41	drwxrwx--x
PREFS_PRIVATE.xml	114	2008-03-12	13:41	-rw-rw----
PREFS_WORLD_READABLE.xml	117	2008-03-12	13:41	-rw-rw-r--
PREFS_WORLD_READABLE_WRITABLE.xml	126	2008-03-12	13:41	-rw-rw-rw-
PREFS_WORLD_WRITABLE.xml	119	2008-03-12	13:41	-rw-rw--w-
com.other.manning.chapter5.prefs		2008-03-12	13:42	drwxrwx--x
download		2008-03-12	13:37	drwxrwxrwx

Figure 5.1 Screen shot of the Android File Explorer view showing preferences file permissions.

The quick and dirty version of how Linux file permissions work are that each file (or directory) has a type, and three sets of permissions represented by a "drwxrwxrwx" notation. The first character indicates the type ("d" means directory, "-" means regular file, symbolic links and other things can be represented using the type as well). After the type, the three sets of "rwx" represent read, write, and or execute permissions for "user," "group," and "other" - in that order. So looking at this notation we can tell which files are accessible by the "user" they are owned by, or by the "group" they belong to, or by "other."

Directories with the other x permission

Directory permissions can be confusing. The important thing to remember with regard to Android though, is that each package directory is created with the other "x" permission. This means anyone can search and list the files in the directory. This, in turn, means that Android packages have directory level access to each others files – from there the file level access determines file permissions.

SharedPreferences XML files are placed in the `"/data/data/PACKAGE_NAME/shared_prefs"` path on the file system. Every application or package (each .apk file) has it's own user ID (unless you use `sharedUserId` in the manifest, which allows you to share the UID, but that's a special exception). When an application creates files (including `SharedPreferences`) they are owned by that application's UID. To allow "other" applications to access these files the other permissions have to be set (as we see within figure 5.2, where one of our preferences files has no outside permissions, one of our files is world readable, one is world read and writeable, and one is world writable).

The tricky part with getting access to the files of one application from another application, even when they have accessible permissions, is the starting path. The path is built from the `Context`. So, to get files from another application you have to know, and use, that application's `Context`. An example of this is seen in listing 5.3, where we get the `SharedPreferences` we set in listing 5.1 again, this time from a different application (different .apk, and different package).

Listing 5.3 Getting SharedPreferences data from a different application after it has been stored.

```
package com.other.manning.chapter5.prefs;           #1

. . . imports omitted for brevity

public class SharedPrefTestOutput extends Activity {

. . . constants and variable declarations omitted for brevity

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.shared_pref_test_output);

        this.outputPrivate = (TextView) this.findViewById(R.id.output_private);
    }
}
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        this.outputWorldRead = (TextView) this.findViewById(R.id.output_worldread);
        this.outputWorldWrite = (TextView) this.findViewById(R.id.output_worldwrite);
        this.outputWorldReadWrite = (TextView) this.findViewById(R.id.output_worldreadwrite);
    }

    @Override
    public void onStart() {
        super.onStart();
        Context otherAppsContext = null;
        try {
            otherAppsContext = this.createPackageContext("com.msi.manning.chapter5.prefs",
                Context.MODE_WORLD_WRITEABLE);    #2
        } catch (NameNotFoundException e) {
            Log.e("SharedPrefTestOutput", e.getLocalizedMessage());
        }

        this.prefsPrivate = otherAppsContext.getSharedPreferences(
            SharedPrefTestOutput.PREFS_PRIVATE, 0);    #3
        this.prefsWorldRead = otherAppsContext.getSharedPreferences(
            SharedPrefTestOutput.PREFS_WORLD_READ, 0);
        this.prefsWorldWrite = otherAppsContext.getSharedPreferences(
            SharedPrefTestOutput.PREFS_WORLD_WRITE, 0);
        this.prefsWorldReadWrite = otherAppsContext.getSharedPreferences(
            SharedPrefTestOutput.PREFS_WORLD_READ_WRITE, 0);

        this.outputPrivate.setText(this.prefsPrivate.getString(
            SharedPrefTestOutput.KEY_PRIVATE, "NA"));
        this.outputWorldRead.setText(this.prefsWorldRead.getString(
            SharedPrefTestOutput.KEY_WORLD_READ, "NA"));
        this.outputWorldWrite.setText(this.prefsWorldWrite.getString(
            SharedPrefTestOutput.KEY_WORLD_WRITE, "NA"));
        this.outputWorldReadWrite.setText(this.prefsWorldReadWrite.getString(
            SharedPrefTestOutput.KEY_WORLD_READ_WRITE, "NA"));
    }
}

```

1. Using a different package, for a different application
2. Get another application's context using `createPackageContext()` with a String
3. Use `otherAppsContext` to get `SharedPreferences`

To get to the `SharedPreferences` one application has defined from another application in a different package **#1**, we have to use the `createPackageContext(String contextName, int mode)` method **#2**. Once we have a reference to the other applications `Context`, we can then use the same names for the `SharedPreferences` objects the other application created (we do have to know the names), to access those preferences **#3**.

With these examples we now have one application that sets and gets `SharedPreferences`, and a second application (in a different package, with a different .apk file) that gets the preferences set by the first. The composite screen shot shown in figure 5.2 demonstrates what this looks like (where "NA" are the preferences we could not access from the second application, due to permissions).



Figure 5.2 Screen shots of two separate applications getting and setting `SharedPreferences`.

The way `SharedPreferences` are backed by XML files on the Android file system, and use permission modes, leads us to the next method of storing and retrieving data, the file system itself.

5.2 Using the file system

As we have seen, Android has a file system that is based on Linux and supports mode based permissions. There are several ways you can access this file system. You can create and read files from within applications, you can access raw files that are included as resources, and you can work with specially compiled custom XML files. In this section we will take a quick tour of each approach.

5.2.1 Creating files

You can easily create files in Android, and have them stored in the file system under the data path for the application you are working in. Listing 5.4 demonstrates how you get a `FileOutputStream` handle, and how you write to it to create a file.

Listing 5.4 Creating a file in Android from an Activity

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

public class CreateFile extends Activity {

    private EditText createInput;
    private Button createButton;

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.create_file);

        this.createInput = (EditText) this.findViewById(R.id.create_input);
        this.createButton = (Button) this.findViewById(R.id.create_button);

        this.createButton.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                FileOutputStream fos = null;
                try {
                    fos = CreateFile.this.openFileOutput("filename.txt", Context.MODE_PRIVATE); #1
                    fos.write(CreateFile.this.createInput.getText().toString().getBytes()); #2
                } catch (FileNotFoundException e) {
                    Log.e("CreateFile", e.getLocalizedMessage());
                } catch (IOException e) {
                    Log.e("CreateFile", e.getLocalizedMessage());
                } finally {

                    if (fos != null) {
                        try {
                            fos.flush(); #3
                            fos.close(); #3
                        } catch (IOException e) {
                            // swallow
                        }
                    }
                }
                CreateFile.this.startActivity(new Intent(CreateFile.this, ReadFile.class));
            }
        });
    }
}

```

1. Use `openFileOutput` to get a `FileOutputStream`
2. Write byte data to the stream
3. Remember to flush and close the stream

Android provides a convenience method on `Context` to get a `FileOutputStream` reference, `openFileOutput(String name, int mode)` #1. Using this method you can create a stream to a file. That file will ultimately be stored at the `data/data/[PACKAGE_NAME]/files/file.name` path on the platform. Once you have the stream you can write to it as you would with typical Java #2. After you are done with a stream you have to remember to flush it and close it to cleanup #3.

Reading from a file within an application context (that is, within the package path of the application) is also very simple, in the next section we will see how this can be done.

5.2.2 Accessing files

Similarly to `openFileOutput`, the `Context` also has a convenience `openFileInput` method. This method can be used to access a file on the file system and read it in, as seen in listing 5.5.

Listing 5.5 Accessing an existing file in Android from an Activity.

```

public class ReadFile extends Activity {

    private TextView readOutput;
    private Button gotoReadResource;

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.read_file);

        this.readOutput = (TextView) this.findViewById(R.id.read_output);

        FileInputStream fis = null;
        try {
            Please post comments or corrections to the Author Online forum at
            http://www.manning-sandbox.com/forum.jspa?forumID=411
            Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

```

        fis = this.openFileInput("filename.txt");           #1
        byte[] reader = new byte[fis.available()];
        while (fis.read(reader) != -1) {                   #2
        }
        this.readOutput.setText(new String(reader));
    } catch (IOException e) {
        Log.e("ReadFile", e.getMessage(), e);
    } finally {
        if (fis != null) {
            try {
                fis.close();                               #3
            } catch (IOException e) {
                // swallow
            }
        }
    }

    this.gotoReadResource = (Button) this.findViewById(R.id.read_button);
    this.gotoReadResource.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            ReadFile.this.startActivity(new Intent(ReadFile.this, ReadRawResourceFile.class));
        }
    });
}

```

1. Use `openFileInput` to get a `FileInputStream`
2. Use the stream to read data from a file
3. Clean up when done

Getting a `FileInputStream`, to read in a file from the file system, is the mirror opposite of getting a `FileOutputStream`. For input you use `openFileInput(String name, int mode)` to get the stream, and then you read in the file as with standard Java. Again, once you are done, you need to close the stream properly to avoid hanging on to resources.

With `openFileOutput` and `openFileInput` you can write to, and read from, any file within the files directory of the application package you are working within. Also, much like the access modes and permissions we saw in the previous sections you can access files across different applications if the permissions allow it, and if you know the full path to the file (you know the package to establish the path from the other application's context).

Running a bundle of apps with the same UID

Though it is the exception, rather than rule, there are times when setting the UID (user ID) your application runs as can be extremely useful (most of the time the platform just selects a unique UID for you). For instance, if you have multiple applications that need to store data among each other, but you also want that data to not be accessible outside that group of applications, you may want to set the permissions to private, and share the UID to allow access. You can allow a shared UID by using the "sharedUserId" attribute in your manifest: `android:sharedUserId="YourFancyID"`.

Along with creating files from within your application, you can also push and pull files to the platform, using the Android Debug Bridge tool (adb, which we met in chapters 1 and 2). You can optionally put such files in the directory for your application – once there you can read these files just like you would any other file. Keep in mind though, outside of development related use, you won't usually be pushing and pulling files. Rather you will be creating and reading files from within the application, or working with files that are included with an application as a raw resource, as we will see next.

5.2.3 Files as raw resources

If you want to include raw files with your application, of any form, you can do so using the `res/raw` resources location. We discussed resources in general in chapter 3, but we did not drill down into files there, as we will do here. When you place a file in the `res/raw` location, it is not compiled by the platform, but is available through the means shown in listing 5.6.

Listing 5.6 Accessing a non compiled raw file from `res/raw`.

```

public class ReadRawResourceFile extends Activity {

    private TextView readOutput;

    Please post comments or corrections to the Author Online forum at
    http://www.manning-sandbox.com/forum.jspa?forumID=411
    Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```



```

private Button gotoReadXMLResource;

@Override
public void onCreate(final Bundle icle) {
    super.onCreate(icle);
    this setContentView(R.layout.read_rawresource_file);

    this.readOutput = (TextView) this.findViewById(R.id.readrawres_output);

    Resources resources = this.getResources();
    InputStream is = null;    #1
    try {
        is = resources.openRawResource(R.raw.people);    #2
        byte[] reader = new byte[is.available()];
        while (is.read(reader) != -1) {
        }
        this.readOutput.setText(new String(reader));
    } catch (IOException e) {
        Log.e("ReadRawResourceFile", e.getMessage(), e);
    } finally {    #3
        if (is != null) {
            try {
                is.close();
            } catch (IOException e) {
                // swallow
            }
        }
    }

    this.gotoReadXMLResource = (Button) this.findViewById(R.id.readrawres_button);
    this.gotoReadXMLResource.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            ReadRawResourceFile.this.startActivity(new Intent(ReadRawResourceFile.this,
ReadXMLResourceFile.class));
        }
    });
}
}

```

1. Define an InputStream to hold the raw resource
2. Use getResources() and openRawResource() to get the resource
3. Again, clean up

Getting raw resources is very similar to getting files. You get a handle to an InputStream, and then you can use that stream to assign to a raw reference later **#1**. You call Context.getResources() to get the Resources reference for your current application's context, and then you call openRawResource(int id) to link to the particular item you want **#2**. The id will automatically be available from the R class if you place your asset in the res/raw directory. Note that raw resources don't have to be text files, even though that is what we are using here. They can also be images, documents, you name it.

The significance with raw resources is that they are not pre-compiled by the platform, and they can refer to any type of raw file. The last type of file resource we need to discuss is the res/xml type – which is compiled by the platform into an efficient binary type that you need to access in a special manner.

5.2.4 XML file resources

The terms can get confusing when talking about “XML resources” in Android circles. This is because XML resources can mean resources in general that are defined in XML, such as layout files, styles, arrays, and the like, or it can specifically mean res/xml XML files.

In this section we will be dealing with res/xml XML files. These files are treated a bit differently than other Android resources. They are different than raw files in that you don't use a stream to access them because they are compiled into an efficient binary form when deployed, and they are different than other resources in that they can be of any custom XML structure that you desire.

To demonstrate this concept we are going to use an XML file that defines multiple <person> elements, and uses attributes for firstname and lastname – people.xml. We will then grab this resource and display the elements within it on screen in last name, first name order, as seen in figure 5.3.

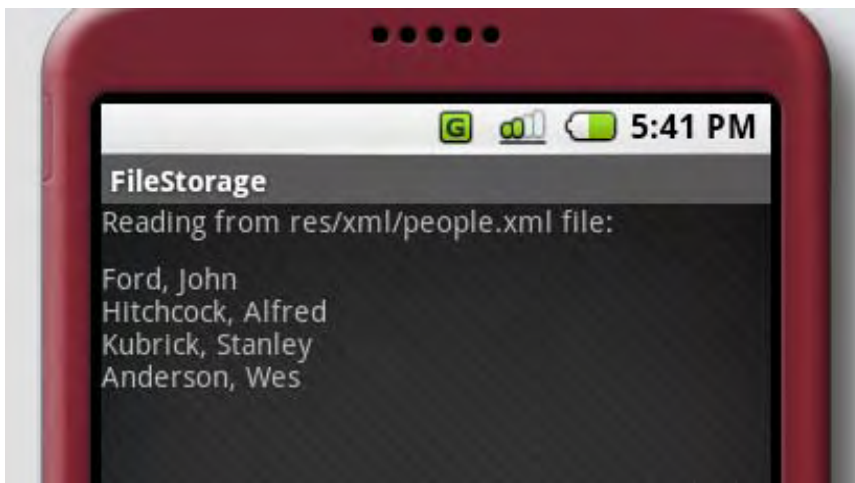


Figure 5.3 A screen shot of the example `ReadXMLResourceFile` Activity created in listing 5.8, which reads a `res/xml` resource file.

Our data file for this process, which we will place in `res/xml` in source, is shown in listing 5.7.

Listing 5.7 A custom XML file included in `res/xml`.

```
<people>
  <person firstname="John" lastname="Ford" />
  <person firstname="Alfred" lastname="Hitchcock" />
  <person firstname="Stanley" lastname="Kubrick" />
  <person firstname="Wes" lastname="Anderson" />
</people>
```

Once a file is in the `res/xml` path, it will be automatically picked up by the platform (if you are using Eclipse) and compiled into a resource asset. This asset can then be accessed in code by parsing the binary XML format Android supports, as shown in listing 5.8.

Listing 5.8 Accessing a compiled XML resource from `res/xml`.

```
public class ReadXMLResourceFile extends Activity {

    private TextView readOutput;

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.read_xmlresource_file);

        this.readOutput = (TextView) this.findViewById(R.id.readxmlres_output);

        XmlPullParser parser = this.getResources().getXml(R.xml.people); #1
        StringBuffer sb = new StringBuffer();

        try {
            while (parser.next() != XmlPullParser.END_DOCUMENT) { #2
                String name = parser.getName();
                String first = null;
                String last = null;
                if ((name != null) && name.equals("person")) {
                    int size = parser.getAttributeCount(); #3
                    for (int i = 0; i < size; i++) {
                        String attrName = parser.getAttributeName(i); #4
                        String attrValue = parser.getAttributeValue(i); #4
                        if ((attrName != null) && attrName.equals("firstname")) {
                            first = attrValue;
                        } else if ((attrName != null) && attrName.equals("lastname")) {
                            last = attrValue;
                        }
                    }
                }
            }
        }
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        if ((first != null) && (last != null)) {
            sb.append(last + ", " + first + "\n");
        }
    }
} catch (Exception e) {
    Log.e("ReadXMLResourceFile", e.getMessage(), e);
}
this.readOutput.setText(sb.toString());
}
}

```

1. Using XmlPullParser to parse a custom XML file
2. Walking the XML tree with next()
3. Getting the attributeCount for the current element
4. Getting the attribute name and value

To process a binary XML resource you use an `XmlPullParser` **#1**. This class can walk through the XML tree SAX style. The parser provides an event type represented by an int for each element it encounters such as: `DOCDECL`, `COMMENT`, `START_DOCUMENT`, `START_TAG`, `END_TAG`, `END_DOCUMENT` – and so on. Using the `next()` method you can retrieve the current event type value and compare it to event constants in the class **#2**. Each element encountered has a name, text value, and an optional set of attributes. You can walk the document as we are here, by getting the `attributeCount` **#3** for each item, and then grabbing the name and value **#4**.

Apart from local file storage on the device file system, you also have another option that is more appropriate for certain types of content, writing to an external SD card file system.

5.2.5 External Storage via an SD Card

One of the advantages the Android platform provides over some other similar device competitors is that it offers access to an available Secure Digital (SD) flash memory card. Ultimately, it is possible that not every Android device will have an SD card, but, the good news is that if the device does have it, the platform supports it and provides an easy way for you to use it.

SD cards and the emulator

In order to work with an SD card image in the Android emulator, you will first need to use the “mkshcard” tool provided to setup your SD image file (you will find this executable in the “tools” directory of the SDK). After you have created the file, you then will need to start the emulator with the `-sdcard <path_to_file>` option in order to have the SD image mounted.

Using the SD card makes a lot of sense if you are dealing with large files, or when you don't necessarily need to have permanent secure access to certain files. Obviously if you are working with image data, or audio files, or the like, you will want to store these on the SD card. The built in internal file system is stored on the system memory, which is limited on a small mobile device – you don't typically want to throw snapshots of grandma on the device itself, if you have other options (like an SD card). On the other hand, for application specialized data that you do need to be permanent, and that you are concerned about secure access for, you should use the internal file system (or a database, or such).

This is because the SD card is impermanent (the user can remove it), and SD card support on most devices, including Android powered devices, supports the FAT (File Allocation Table) file system. That's important to remember because it will help you keep in mind that the SD card doesn't have the access modes and permissions that come from the Linux file system.

Using the SD card, when you need it, is fortunately pretty basic. The standard `java.io.File` and related objects can be used to create and read (and remove) files on the `/sdcard` path (assuming that path is available, which you do need to check, also using the standard `File` methods). Listing 5.9 is an example of checking that the `/sdcard` path is present, then creating another subdirectory therein, and then writing, and subsequently reading, file data at that location.

Listing 5.9 Using standard java.io.File techniques with and SD card at the /sdcard path.

```
public class ReadWriteSDCardFile extends Activity {

    private static final String LOGTAG = "FileStorage";

    private TextView readOutput;

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.read_write_sdcard_file);

        this.readOutput = (TextView) this.findViewById(R.id.readwritesd_output);

        String fileName = "testfile-" + System.currentTimeMillis() + ".txt"; #1

        File sdDir = new File("/sdcard/"); #2
        if (sdDir.exists() && sdDir.canWrite()) {
            File uadDir = new File(sdDir.getAbsolutePath() + "/unlocking_android"); #3
            uadDir.mkdir(); #4
            if (uadDir.exists() && uadDir.canWrite()) {
                File file = new File(uadDir.getAbsolutePath() + "/" + fileName); #5
                try {
                    file.createNewFile(); #6
                } catch (IOException e) {
                    Log.e(ReadWriteSDCardFile.LOGTAG, "error creating file", e);
                }

                if (file.exists() && file.canWrite()) {
                    FileOutputStream fos = null;
                    try {
                        fos = new FileOutputStream(file);
                        fos.write("I fear you speak upon the rack,
                                where men enforced do speak anything.".getBytes()); #7
                    } catch (FileNotFoundException e) {
                        Log.e(ReadWriteSDCardFile.LOGTAG, "ERROR", e);
                    } catch (IOException e) {
                        Log.e(ReadWriteSDCardFile.LOGTAG, "ERROR", e);
                    } finally {
                        if (fos != null) {
                            try {
                                fos.flush();
                                fos.close();
                            } catch (IOException e) {
                                // swallow
                            }
                        }
                    }
                } else {
                    Log.e(ReadWriteSDCardFile.LOGTAG, "error writing to file");
                }
            } else {
                Log.e(ReadWriteSDCardFile.LOGTAG, "ERROR, unable to write to /sdcard/unlocking_android");
            }
        } else {
            Log.e(ReadWriteSDCardFile.LOGTAG, "ERROR, /sdcard path not available (did you create
            an SD image with the mkcard tool, and start emulator with -sdcard
            <path_to_file> option?");
        }

        File rFile = new File("/sdcard/unlocking_android/" + fileName); #8
        if (rFile.exists() && rFile.canRead()) {
            FileInputStream fis = null;
            try {
                fis = new FileInputStream(rFile);
                byte[] reader = new byte[fis.available()]; #9
                while (fis.read(reader) != -1) {
                }
                this.readOutput.setText(new String(reader));
            } catch (IOException e) {
                Log.e(ReadWriteSDCardFile.LOGTAG, e.getMessage(), e);
            } finally {
                if (fis != null) {
                    try {
                        fis.close();
                    } catch (IOException e) {
                        // swallow
                    }
                }
            }
        }
    }
}
```

```

    }
    } else {
        this.readOutput.setText("Unable to read/write sdcard file, see logcat output");
    }
}
}

```

1. Establish filename
2. Get a reference to the /sdcard directory
3. Instantiate the /sdcard/unlocking_android directory File object
4. Call mkdir() to create the directory if it does not already exist
5. Get a reference to the actual File we want to write to (not just the directory)
6. Create the file
7. Using a FileInputStream write some data to the file
8. Create a new File object for reading the file back
9. Read the file with a FileOutputStream

The first thing we need have done in the `ReadWriteSDCardFile` class is to establish a filename for the file we want to create **#1**. We have done this appending a timestamp so as to create a new unique file each time this example application is run. After we have the filename we then create a `File` object reference to the /sdcard directory **#2**. From there we create a `File` reference to a new subdirectory /sdcard/unlocking_android **#3** (in Java both files and directories can be represented by the `File` object). After we have the subdirectory reference we call `mkdir()` to ensure it is created if it does not already exist **#4**.

With the structure we need then in place, we next follow a similar pattern for the actual file. We instantiate a reference `File` object **#5**, and then we call `createFile()` to actually create a file on the filesystem **#6**. Once we have the `File`, and we know it exists and we are allowed to write to it (recall files on the sdcard will be world writeable by default, it's using a FAT filesystem), we then use a `FileInputStream` to write some data into the file.

After we create the file and have some data in it, we then create another `File` object with the full path to read the data back **#8**. Yes, we could use the same `File` object handle that we had when creating the file, but for the purposes of the example we wanted to explicitly demonstrate starting with a fresh `File`. With the `File` reference we then create a `FileOutputStream` and read back the data that was earlier stored in the file **#9**.

As you can see working with files on the SD card is pretty much standard `java.io.File` fare. This does entail a good bit of "boilerplate" Java code to make a robust solution, with permissions and error checking every step of the way, and logging about what is happening, but it is still simple and powerful. If you need to do a lot of `File` handling you will probably want to create some simple local utilities for wrapping the mundane tasks so you don't have to repeat them over and over again (opening files, writing to them, creating them, and so on). (Also you may want to look at using or porting something like the Apache commons.io package – which includes a `FileUtils` class that handles these types of tasks and more.)

The SD card example completes our exploration in this section, where we have seen that there are various ways to store different types of file data on the Android platform. If you have static elements that are pre-defined you can use `res/raw`, if you have XML files you can use `res/xml`, and you can also work directly with the file system by creating, modifying, and retrieving data in files (either in the local internal filesystem, or on the SD card if available).

Another way to deal with data, one which may be more appropriate for many situations (such as when you need to share relational data across applications), is through the use of a database.

5.3 Persisting data to a database

One nice convenience that the Android platform provides is the fact that a relational database is built in. SQLite doesn't have all of the features of larger client/server database products, but it

does cover just about anything you might need for local data storage – while being easy to deal with, and quick.

In this section we are going to cover working with the built in SQLite database system, from creating and querying a database, to upgrading, and working with the sqlite3 tool that is available in the Android Debug Bridge (ADB) shell. Once again we will do this in the context of the WeatherReporter application we began in chapter 4. This application uses a database to store the user's "saved" locations, and also persists user preferences for each location. The screen shot shown in figure 5.4 displays this saved data for the user to select from – when the user selects a location, data is retrieved from the database and a location weather report is shown.

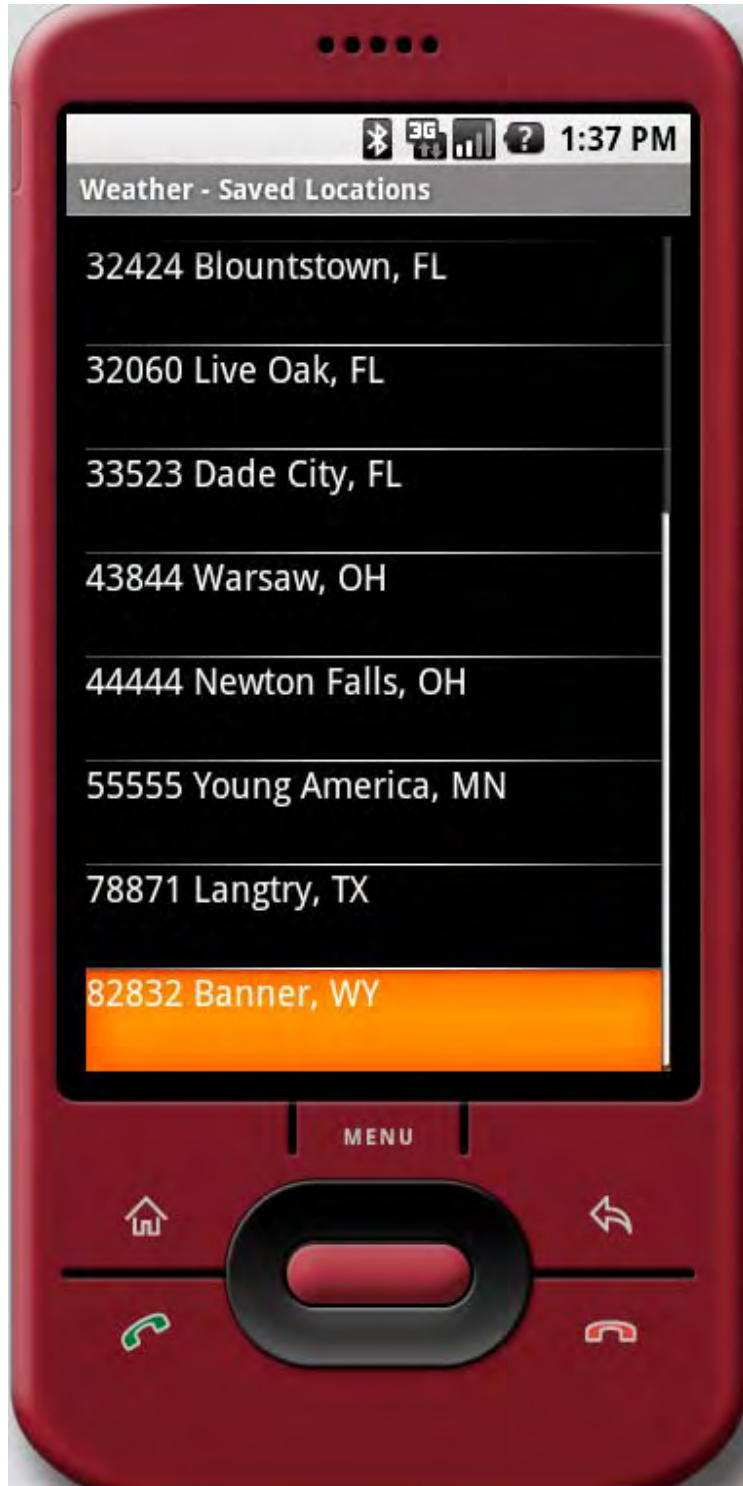


Figure 5.4 The WeatherReporter saved locations screen, which pulls data from a SQLite database.

To see how this comes together we will begin with what it takes to create the database WeatherReporter uses.

5.3.1 Building and accessing a database

To use SQLite you have to know a bit about Structured Query Language (SQL) usage in general. If you need to brush up on the background of the basic commands: CREATE, INSERT, UPDATE, DELETE, and SELECT; then you may want to take a quick look at the SQLite documentation (<http://www.sqlite.org/lang.html>).

For our purposes we are going to jump right in and build a database helper class that our application will use. We are creating a helper class so that the details concerning creating and upgrading our database, opening and closing connections, and running through specific queries, are all encapsulated in one place and not otherwise exposed or repeated in our application code. This is so that our Activity and Service classes can later just use simple “get” and “insert” methods, with specific bean objects representing our model, or Collections, rather than database specific abstractions (such as the Android Cursor object that represents a query result set). You can think of this class as a miniature Data Access Layer (DAL) if you will.

The first part of our DBHelper class, which itself includes a few inner classes we will learn about, is shown in listing 5.10.

Listing 5.10 The first portion of the WeatherReporter DBHelper class showing the DBOpenHelper inner class.

```
public class DBHelper {

    public static final String DEVICE_ALERT_ENABLED_ZIP = "DAEZ99";           #|1
    public static final String DB_NAME = "w_alert";                         #|1
    public static final String DB_TABLE = "w_alert_loc";                     #|1
    public static final int DB_VERSION = 3;                                  #|1

    private static final String CLASSNAME = DBHelper.class.getSimpleName();
    private static final String[] COLS = new String[]
    { "_id", "zip", "city", "region", "lastalert", "alertenabled" };

    private SQLiteDatabase db;
    private final DBOpenHelper dbOpenHelper;

    public static class Location {      #2
        public long id;
        public long lastalert;
        public int alertenabled;
        public String zip;
        public String city;
        public String region;

        public Location() {
        }

        public Location(final long id, final long lastalert, final int alertenabled, final String zip,
            final String city, final String region) {
            this.id = id;
            this.lastalert = lastalert;
            this.alertenabled = alertenabled;
            this.zip = zip;
            this.city = city;
            this.region = region;
        }

        @Override
        public String toString() {
            return this.zip + " " + this.city + ", " + this.region;
        }
    }

    private static class DBOpenHelper extends SQLiteOpenHelper {           #3

        private static final String DB_CREATE = "CREATE TABLE "
        Please post comments or corrections to the Author Online forum at
        http://www.manning-sandbox.com/forum.jspa?forumID=411
        Licensed to Thow Way Chiam <ken.ctw@gmail.com>
```



```

        + DBHelper.DB_TABLE
        + " (_id INTEGER PRIMARY KEY, zip TEXT UNIQUE NOT NULL,
        city TEXT, region TEXT, lastalert INTEGER, alertenabled INTEGER);";      #4

    public DBOpenHelper(final Context context, final String dbName, final int version) {
        super(context, DBHelper.DB_NAME, null, DBHelper.DB_VERSION);
    }

    @Override
    public void onCreate(final SQLiteDatabase db) {      #5
        try {
            db.execSQL(DBOpenHelper.DB_CREATE);
        } catch (SQLException e) {
            Log.e(Constants.LOGTAG, DBHelper.CLASSNAME, e);
        }
    }

    @Override
    public void onOpen(final SQLiteDatabase db) {      #5
        super.onOpen(db);
    }

    @Override
    public void onUpgrade(final SQLiteDatabase db, final int oldVersion, final int newVersion) {      #5
        db.execSQL("DROP TABLE IF EXISTS " + DBHelper.DB_TABLE);
        this.onCreate(db);
    }
}

```

1. Using constants for database name, database version, and table name
2. Defining an inner Location bean class
3. Defining an inner DBOpenHelper class to manage creation, upgrade, and connection
4. Defining the SQL query string used to create database
5. Overriding callback for onCreate, onOpen, and onUpgrade

Within our `DBHelper` class we first have a series of constants that define important static values relating to the database we want to work with, such as: database name, database version, and table name **#1**. Then, several of the most important parts of the database helper class that we have created for the WeatherReporter application are the inner classes.

The first inner class is a simple `Location` bean that is used to represent a user's selected location to save **#2**. This class intentionally does not have accessors and mutators, because these add overhead and we don't really need them when we will only use this bean within our application (we won't expose it). The second inner class is a `SQLiteOpenHelper` implementation **#3**.

Our `DBOpenHelper` inner class extends `SQLiteOpenHelper`, which is a class that Android provides to help with creating, upgrading, and opening databases. Within this class we are including a `String` that represents the CREATE query we will use to build our database table – this shows the exact columns and types our table will have **#4**. The data types we are using are fairly self explanatory, most of the time you will use `INTEGER` and `TEXT` types, as we have (if you need more information about the other types `SQLite` supports please see the documentation: <http://www.sqlite.org/datatype3.html>). Also within `DBOpenHelper` we are implementing several key `SQLiteOpenHelper` callback methods, notably `onCreate` and `onUpgrade` (`onOpen` is also supported, but we aren't using it). We will see how these callbacks come into play, and why this class is so helpful, in the second part of our `DBHelper` (the outer class), which is shown in listing 5.11.

Listing 5.11 The second portion of the WeatherReporter DBHelper class showing convenience methods to insert, get, delete, and update data.

```

    public DBHelper(final Context context) {      #1
        this.dbOpenHelper = new DBOpenHelper(context, "WR_DATA", 1);
        this.establishDb();
    }

    private void establishDb() {      #2
        if (this.db == null) {
            this.db = this.dbOpenHelper.getWritableDatabase();
        }
    }

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

    }

    public void cleanup() {                                #3
        if (this.db != null) {
            this.db.close();
            this.db = null;
        }
    }

    public void insert(final Location location) {           #4
        ContentValues values = new ContentValues();
        values.put("zip", location.zip);
        values.put("city", location.city);
        values.put("region", location.region);
        values.put("lastalert", location.lastalert);
        values.put("alertenabled", location.alertenabled);
        this.db.insert(DBHelper.DB_TABLE, null, values);
    }

    public void update(final Location location) {           #4
        ContentValues values = new ContentValues();
        values.put("zip", location.zip);
        values.put("city", location.city);
        values.put("region", location.region);
        values.put("lastalert", location.lastalert);
        values.put("alertenabled", location.alertenabled);
        this.db.update(DBHelper.DB_TABLE, values, "_id=" + location.id, null);
    }

    public void delete(final long id) {                     #4
        this.db.delete(DBHelper.DB_TABLE, "_id=" + id, null);
    }

    public void delete(final String zip) {                 #4
        this.db.delete(DBHelper.DB_TABLE, "zip='" + zip + "'", null);
    }

    public Location get(final String zip) {                 #5
        Cursor c = null;
        Location location = null;
        try {
            c = this.db.query(true, DBHelper.DB_TABLE, DBHelper.COLS,
                "zip = '" + zip + "'", null, null, null, null,
                null);
            if (c.getCount() > 0) {
                c.moveToFirst();
                location = new Location();
                location.id = c.getLong(0);
                location.zip = c.getString(1);
                location.city = c.getString(2);
                location.region = c.getString(3);
                location.lastalert = c.getLong(4);
                location.alertenabled = c.getInt(5);
            }
        } catch (SQLException e) {
            Log.v(Constants.LOGTAG, DBHelper.CLASSNAME, e);
        } finally {
            if (c != null && !c.isClosed()) {
                c.close();
            }
        }
        return location;
    }

    public List<Location> getAll() {                         #5
        ArrayList<Location> ret = new ArrayList<Location>();
        Cursor c = null;
        try {
            c = this.db.query(DBHelper.DB_TABLE, DBHelper.COLS, null, null, null, null, null);
            int numRows = c.getCount();
            c.moveToFirst();
            for (int i = 0; i < numRows; ++i) {
                Location location = new Location();
                location.id = c.getLong(0);
                location.zip = c.getString(1);
                location.city = c.getString(2);
                location.region = c.getString(3);
                location.lastalert = c.getLong(4);
                location.alertenabled = c.getInt(5);
                if (!location.zip.equals(DBHelper.DEVICE_ALERT_ENABLED_ZIP)) {
                    ret.add(location);
                }
            }
        }
    }

```

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        }
        c.moveToNext();
    }
} catch (SQLException e) {
    Log.v(Constants.LOGTAG, DBHelper.CLASSNAME, e);
} finally {
    if (c != null && !c.isClosed()) {
        c.close();
    }
}
return ret;
}

public List<Location> getAllAlertEnabled() {           #5
    Cursor c = null;
    ArrayList<Location> ret = new ArrayList<Location>();
    try {
        c = this.db.query(DBHelper.DB_TABLE, DBHelper.COLS,
            "alertenabled = 1", null, null, null, null);
        int numRows = c.getCount();
        c.moveToFirst();
        for (int i = 0; i < numRows; ++i) {
            Location location = new Location();
            location.id = c.getLong(0);
            location.zip = c.getString(1);
            location.city = c.getString(2);
            location.region = c.getString(3);
            location.lastalert = c.getLong(4);
            location.alertenabled = c.getInt(5);
            if (!location.zip.equals(DBHelper.DEVICE_ALERT_ENABLED_ZIP)) {
                ret.add(location);
            }
            c.moveToNext();
        }
    } catch (SQLException e) {
        Log.v(Constants.LOGTAG, DBHelper.CLASSNAME, e);
    } finally {
        if (c != null && !c.isClosed()) {
            c.close();
        }
    }
    return ret;
}
}

```

1. Create an instance of DBOpenHelper in the constructor of DBHelper
2. Provide an establishDb method that creates database if needed
3. Provide a cleanup method for callers to use
4. Provide a series of convenience methods that insert, update, and delete data
5. Provide several methods for getting data

Our DBHelper class contains a member level variable reference to a SQLiteDatabase object, as we saw in listing 5.10 (the first half of this class). This object is the Android database workhorse. This is what is used to open database connections, to execute SQL statements, and more.

Then the DBOpenHelper inner class we also saw in the first part of the DBHelper class listing is instantiated inside the constructor **#1**. From there the dbOpenHelper is used, inside the establishDb method if the db reference is null, to call openDatabase with the current Context, database name, and database version **#2**. This establishes db as an instance of SQLiteDatabase through DBOpenHelper.

Though you can also just open a database connection directly on your own, using the “open” helper in this way invokes the provided callbacks and makes the process easier. With this technique, when we try to open our database connection, it is automatically created or upgraded (or just returned), if necessary, through our DBOpenHelper. Though using a DBOpenHelper entails a few extra steps up front, once you have it in place it is extremely handy when you need to modify your table structure (you can simply increment your version, and do what you need in the onUpgrade callback – without this you have to manually alter, and or remove and recreate, your existing structure).

Another important thing to provide in a helper class like this is a cleanup method **#3**. This method is used by callers who can invoke it when they pause, in order to close connections and free up resources.

After the cleanup method we then see the raw SQL convenience methods that encapsulate the operations our helper provides. In this class we have methods to insert, update, and delete data **#4**. And, we also have a few specialized get, and get all methods **#5**. Within these methods you get a feel for how the `db` object is used to run queries. The `SQLiteDatabase` class itself has many convenience methods, such as `insert`, `update`, and `delete` – which we are wrapping – and it provides direct query access that returns a `Cursor` over a result set.

Databases are package private

Unlike the `SharedPreferences` we saw earlier, you can't make a database `WORLD_READABLE`. Each database is only accessible by the package in which it was created – this means only accessible to the process that created it. If you need to pass data across processes, you can use `AIDL/Binder` (as we saw in chapter 4), or create a `ContentProvider` (as we will see next), but you can't use a database directly.

Typically you can get a lot of mileage and utility from basic steps relating to the `SQLiteDatabase` class, as we have here, and by using it you can create a very useful, and fast, data storage mechanism for your Android applications. The last thing we need to discuss with regard to databases is the `sqlite3` tool, which you can use to manipulate data outside of your application.

5.3.2 Using the `sqlite3` tool

When you create a database for an application in Android the files for that database are created on the device in the `/data/data/[PACKAGE_NAME]/database/db.name` location. These files are SQLite proprietary, but there is a way to manipulate, dump, restore, and otherwise work with your databases through these files in the ADB shell – the `sqlite3` tool.

This tool is accessible through the shell, you can get to it by issuing the following commands on the command line (remember to use your own package name, here we are using the package name for the `WeatherReporter` sample application):

```
cd [ANDROID_HOME]/tools
adb shell
sqlite3 /data/data/com.msi.manning.chapter4/databases/w_alert.db
```

Once you are in the shell (you have the `#`) prompt, you can then issue `sqlite3` commands, `.help` should get you started (if you need more see the tool's documentation: <http://www.sqlite.org/sqlite.html>). From the tool you can issue basic commands, such a `SELECT` or `INSERT`, or you can go farther and `CREATE` or `ALTER` tables. This tool comes in handy for basic poking around and troubleshooting, and for using to `.dump` and `.load` data. As with many command line SQL tools it takes some time to get used to the format, but there is no better way to backup or load your data (if you need that facility – in most cases with mobile development you really shouldn't have a huge database, and keep in mind, this tool is only available through the development shell, it's not something you will be able to use to load a real application with data).

Now that we have seen how to use the SQLite support provided in Android, from creating and accessing tables to store data, to investigating databases with the provided tools in the shell, the next thing we need to cover is the last aspect of handling data on the platform, and that is building and using a `ContentProvider`.

5.4 Working with `ContentProviders`

A `ContentProvider` is used in Android to share data between different applications. We have already discussed the fact that each application runs in its own process (normally), and data and files stored there are not accessible by other applications by default. We have learned that you can make preferences and files available across application boundaries with the correct permissions, and if each application knows the

Please post comments or corrections to the Author Online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

context/path. Nevertheless, that is a limited solution for related applications that already “know” details about one another. In contrast to that, with a `ContentProvider` you can publish and expose a particular data type for other applications to use to query, add, update, and delete – and those applications don't need to have any prior knowledge of paths or resources, or even know who or what is providing the content.

The canonical `ContentProvider` example in Android is the contacts list – the list of name, address, and phone information for people stored in the phone. You can access this data from any application using a specific URI, `content://contacts/people/`, and a series of methods provided by the `Activity` and `ContentResolver` classes to retrieve and store data. We will learn more about `ContentResolver` as we explore provider details. One other data related concept that a `ContentProvider` brings along with it is the `Cursor`, the same object we used previously when dealing with SQLite database result sets. `Cursor` is also returned by the provider query methods we will learn about coming up.

ContentProvider leaks a Cursor

Returning a `Cursor` is one of the quirks of a `ContentProvider`. Exposing a `Cursor` from a `ContentProvider` is a fairly “leaky” abstraction, and it makes for an inconsistent API, as we shall learn. `Cursor` is part of the `android.database` package, which implies you are working with database records, and binds you to certain database concepts when you get results. Yet, the entire idea behind a `ContentProvider` is supposed to be that it is backend agnostic. That is to say you should be able to implement a `ContentProvider` and not use a database to get and store data within it, if the situation warrants (the current Android documentation contradicts itself on this point, in one place it says not using a database is possible, in another it says it is not). Currently, regardless of the merits or demerits, you will need to learn to deal with `Cursor` based results and SQL constructs when working with `ContentProvider` calls.

In this section we are going to build several small sample applications to help us look at all of the `ContentProvider` angles. First we will build a single `Activity` based application, which we are calling `ProviderExplorer`, that will work with the built-in contacts database to query, add, update, and delete data. Then, we will create another application that actually implements its own `ContentProvider`, and includes a similar explorer type `Activity` to manipulate that data as well. Along with covering these fundamentals, we will lastly discuss other built-in providers on the platform beyond contacts.

The `ProviderExplorer` application we are going to build here will ultimately have one large scrollable screen, which is depicted in figure 5.5. Keep in mind that we are focusing on covering all the bases in one `Activity` – exposing all of the `ContentProvider` operations in a single place – rather than on aesthetics or usability (this application is downright ugly, but that's intentional – at least this time).



Figure 5.5 ProviderExplorer sample application that uses the contacts `ContentProvider`.

To begin here we will explore the syntax of URIs, and the combinations and paths used to perform different types of operations with the `ContentProvider` and `ContentResolver` classes.

5.4.1 Understanding URI representations and manipulating records

Each `ContentProvider` is required to expose a unique `CONTENT_URI` that is used to identify the content type it will handle. This URI is used in one of two forms, singular or plural, as shown in table 5.1, to query data.

Table 5.1 `ContentProvider` URI variations for different purposes.

URI	Purpose
<code>content://contacts/people/</code>	Return List of all “people” from provider registered to handle <code>content://contacts</code>

The URI concept comes into play whether or not you are querying data, or adding or deleting it, as we shall see. To get familiar with this process we will take a look at the basic CRUD data manipulation methods and see how they are carried out with the contacts database and respective URIs.

We will step through each individual task to highlight the details: create, read, update, and delete. To do this concisely we will build one `Activity` in the `ProviderExplorer` example application that performs all of these actions. In the next few sections we will take a look at different parts of this `Activity` to focus on each task.

The first thing we need to do is setup a bit of scaffolding for the contacts provider we will be using, this is done in the first portion of listing 5.12, the start of the `ProviderExplorer` class.

Listing 5.12 The start of the `ProviderExplorer` Activity that sets up needed inner classes and handles `onCreate`.

```
public class ProviderExplorer extends Activity {

    private EditText addName;
    private EditText addPhoneNumber;
    private EditText editName;
    private EditText editPhoneNumber;
    private Button addContact;
    private Button editContact;

    private long contactId;

    private class Contact {          #1
        public long id;
        public String name;
        public String phoneNumber;

        public Contact(final long id, final String name, final String phoneNumber) {
            this.id = id;
            this.name = name;
            this.phoneNumber = phoneNumber;
        }

        @Override
        public String toString() {
            return this.name + "\n" + this.phoneNumber;
        }
    }

    private class ContactButton extends Button {    #2
        public Contact contact;

        public ContactButton(final Context ctx, final Contact contact) {
            super(ctx);
            this.contact = contact;
        }
    }

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.provider_explorer);

        this.addName = (EditText) this.findViewById(R.id.add_name);
        this.addPhoneNumber = (EditText) this.findViewById(R.id.add_phone_number);
        this.editName = (EditText) this.findViewById(R.id.edit_name);
        this.editPhoneNumber = (EditText) this.findViewById(R.id.edit_phone_number);

        this.addContact = (Button) this.findViewById(R.id.add_contact_button);
        this.addContact.setOnClickListener(new OnClickListener() {    #3
            public void onClick(final View v) {
                ProviderExplorer.this.addContact();    #4
            }
        });
        this.editContact = (Button) this.findViewById(R.id.edit_contact_button);
        this.editContact.setOnClickListener(new OnClickListener() {    #5
            public void onClick(final View v) {
                ProviderExplorer.this.editContact();    #6
            }
        });
    }
}
```



```

        public void onClick(final View v) {
            ProviderExplorer.this.editContact();    #4
        }
    });
}

```

1. Include an inner Contact bean to represent a contact
2. Include a special ContactButton that extends Button
3. Create anonymous click listeners for add and edit
4. Call the addContact and editContact methods

To start out the ProviderExplorer Activity we are creating a simple inner bean class to represent a Contact record (this is not a comprehensive representation, but does capture the fields we are interested in here) **#1**. Then we also include another inner class to represent a ContactButton **#2**. This is a class that extends Button and includes a reference to a particular Contact.

After we have the add and edit buttons established, we then also create anonymous `OnClickListener` implementations **#3** that call the respective add and edit methods when a button is clicked **#4**.

That rounds out the setup related tasks for `ProviderExplorer`, so the next thing we need to implement is the `onStart` method, which adds more buttons dynamically for populating edit data, and deleting data. This is shown in listing 5.13.

Listing 5.13 The `onStart` portion of the `ProviderExplorer` Activity, which sets up dynamic edit and delete buttons.

```

@Override
public void onStart() {
    super.onStart();
    List<Contact> contacts = this.getContacts();    #1
    LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(200,
        android.view.ViewGroup.LayoutParams.WRAP_CONTENT);
    if (contacts != null) {
        LinearLayout editLayout = (LinearLayout) this.findViewById(R.id.edit_buttons_layout);    #2
        LinearLayout deleteLayout = (LinearLayout) this.findViewById(R.id.delete_buttons_layout);    #2
        params.setMargins(10, 0, 0, 0);
        for (Contact c : contacts) {
            ContactButton contactEditButton = new ContactButton(this, c);    #3
            contactEditButton.setText(c.toString());
            editLayout.addView(contactEditButton, params);
            contactEditButton.setOnClickListener(new OnClickListener() {
                public void onClick(final View v) {
                    ContactButton view = (ContactButton) v;
                    ProviderExplorer.this.editName.setText(view.contact.name);
                    ProviderExplorer.this.editPhoneNumber.setText(view.contact.phoneNumber);
                    ProviderExplorer.this.contactId = view.contact.id;
                }
            });

            ContactButton contactDeleteButton = new ContactButton(this, c);    #3
            contactDeleteButton.setText("Delete " + c.name);
            deleteLayout.addView(contactDeleteButton, params);
            contactDeleteButton.setOnClickListener(new OnClickListener() {
                public void onClick(final View v) {
                    ContactButton view = (ContactButton) v;
                    ProviderExplorer.this.contactId = view.contact.id;
                    ProviderExplorer.this.deleteContact();
                }
            });
        }
    } else {
        LinearLayout layout = (LinearLayout) this.findViewById(R.id.edit_buttons_layout);
        TextView empty = new TextView(this);
        empty.setText("No current contacts");
        layout.addView(empty, params);
    }
}

```

1. Get a list of contacts
2. Create dynamic edit and delete layouts for every contact

3. Create dynamic edit and delete buttons for every contact

The `onStart` method makes a call to the `getContacts` method **#1**. This method, which we will see in listing 5.14 next, returns a `List` of current `Contact` objects from the Android contacts database. Once we have the current contacts, we then loop through them and dynamically create a layout in code for edit and delete respectively **#2**. After we have the layout withing it we create a few view objects, including a `ContactButton` to populate an edit form, and a button to delete a record **#3**. Each button is then manually added to a respective `LinearLayout` we have referenced through `R.java` **#4**.

Once our `onStart` method is in place, we then have a `View` to display all the current contacts, and all of the buttons, static and dynamic, that we need in order to be able to add, edit, and delete contact data. From there we need to implement the methods to perform these actions – this is where we will use a `ContentResolver` and other related classes.

Initially we need to populate our display of current contacts, and to do that we need to query for (read) data.

QUERYING DATA

The `Activity` class has a `managedQuery` method on it that is used to make calls into registered `ContentProvider` classes. When we create our own `ContentProvider` in section 5.5.3 we will see how a provider is registered with the platform, for now we are going to focus on calling existing providers. Each provider is required to advertise (or publish in Android terms) the `CONTENT_URI` it supports. To query the contacts provider, as we are in listing 5.14, we have to know this URI and then get a `Cursor` by calling `managedQuery`.

Listing 5.14 The query (read) details for a `ContentProvider` within the `ProviderExplorer` Activity.

```
private List<Contact> getContacts() {
    List<Contact> results = null;
    long id = 0L;
    String name = null;
    String phoneNumber = null;
    String[] projection = new String[] {
        { Contacts.People._ID, Contacts.People.NAME, Contacts.People.NUMBER };    #1
    };
    ContentResolver resolver = this.getContentResolver();                        #2
    Cursor cur = resolver.query(Contacts.People.CONTENT_URI, projection, null, null,
        Contacts.People.DEFAULT_SORT_ORDER);    #3
    while (cur.moveToNext()) {    #4
        if (results == null) {
            results = new ArrayList<Contact>();
        }
        id = cur.getLong(cur.getColumnIndex(BaseColumns._ID));
        name = cur.getString(cur.getColumnIndex(Contacts.PeopleColumns.NAME));
        phoneNumber = cur.getString(cur.getColumnIndex(Contacts.PhonesColumns.NUMBER));
        results.add(new Contact(id, name, phoneNumber));
    }
    return results;
}
```

1. Make a “projection” to represent the columns to return
2. Get a `ContentResolver` reference
3. Get a `Cursor` from the resolver
4. Walk the results and populate data

The Android contacts database is really a composite of several types of data. A “contact” includes details of a person (name, company, photo, and the like), one or more phone numbers (each of which has a number, type, label, and such), and other information. A `ContentProvider` typically supplies all the details of URI and types it supports as constants in a class. In the `android.provider` package, there is `Contacts` class that corresponds to the contacts provider. This class has nested inner classes that represent `People` and `Phones`. Then, in additional inner classes in those, there are constants that represent fields or columns of data for each type. This structure with all the inner classes can be mind bending at times, but the bottom line is that `Contacts` data ends up in multiple tables, and the values you need to query and manipulate this data comes from the inner classes for each type.

The columns we will be using to get and set data are defined in these classes. Here we are going to work with only the people and phones parts of contacts. We start by creating a projection of the columns we want to return as a `String` array **#1**. Then we get a reference to the `ContentResolver` we will use **#2**. Using the resolver, we obtain a `Cursor` object **#3**. Once we have the `Cursor`, which represents the rows in the data we have returned, we then iterate over it to create our contact objects **#4**.

MANAGED CURSOR

In order to obtain a `Cursor` reference you can also use the `managedQuery` method of the `Activity` class. A “managed” `Cursor` is automatically cleaned up when your `Activity` pauses, and is also restarted when it starts. It is a `Cursor` instance that has its state maintained by the platform in conjunction with `Activity` life-cycle. This is very helpful, in most cases, if you just need to retrieve data within an `Activity` you will want to use a managed `Cursor` – as opposed to a `ContentResolver`. (We are not using one in the last example though, because there we need to do more than just retrieve data, and we want to focus on the provider/resolver components.)

The `query` method on the `ContentResolver` class also lets you pass in additional arguments to narrow the results. Specifically, where we passed `null`, `null` in listing 5.14, you can alternatively pass in a “filter” to narrow the rows you want to return in the form of an SQL `WHERE` clause, and optional replacement tokens for that where clause (injected at each `?`). This is somewhat typical SQL usage, so it's easy to work with. The downside though comes when you aren't using a database to back your `ContentProvider`. This is where the abstraction leaks like a sieve – though it might be possible to not use a database for a data source, you still have to handle SQL statements in your provider implementation, and you must require that anyone that uses your provider also has to deal with SQL constructs.

Now that we have covered how to query for data to return results, we will next take a look at inserting new data – adding a row.

INSERTING DATA

In listing 5.15 we see the next part of the `ProviderExplorer` class, the `addContact` method. This is used with the add form elements in our `Activity` to insert a new row of data into the contacts related tables.

Listing 5.15 The insert (create) details for a `ContentProvider` within the `ProviderExplorer` Activity.

```
private void addContact() {
    ContentResolver resolver = this.getContentResolver();    #1
    ContentValues values = new ContentValues();              #2

    values.put(Contacts.People.NAME, this.addName.getText().toString());
    Uri personUri = Contacts.People.createPersonInMyContactsGroup(resolver, values);    #3
    Log.v("ProviderExplorer", "ADD personUri - " + personUri.toString());

    values.clear();
    Uri phoneUri = Uri.withAppendedPath(personUri, Contacts.People.Phones.CONTENT_DIRECTORY);    #4
    Log.v("ProviderExplorer", "ADD phoneUri - " + phoneUri.toString());
    values.put(Contacts.Phones.TYPE, Phones.TYPE_MOBILE);
    values.put(Contacts.Phones.NUMBER, this.addPhoneNumber.getText().toString());

    resolver.insert(phoneUri, values);    #5

    this.startActivity(new Intent(this, ProviderExplorer.class));
}
```

1. Get a handle on a `ContentResolver`
2. Use a `ContentValues` object for query values
3. Use the `Contacts` helper method to create a person in the “my contacts” group
4. Append to an existing person `Uri` to create a phone `Uri`
5. Insert data using a resolver

The first thing we see in the `addContact` method is that we are getting a `ContentResolver` reference **#1**, and then using a `ContentValues` object to map column names with values **#2**. This is an Android specific type of map object. After we have our variables in place, we then use the special `createPersonInMyContactsGroup` helper method on the `Contacts.People` class to both insert a record and return the `Uri` **#3**. This method uses the resolver for us, under the covers, and performs an insert. The `Contacts` class structure has a few helper methods sprinkled throughout (see the `JavaDocs`). These are used to cut down on the amount of code you have to know and write to

perform common tasks – such as adding a contact to the “My Contacts” group (the built-in group that the phone displays by default in the contacts app).

After we have created a new contact `People` record, we then append some new data to that existing `Uri` in order to next create a phone record associated with the same person **#4**. This is a nice touch that the API provides. You can often append and or “build” onto an existing `Uri` in order to access different aspects of the data structure. After we have the `Uri`, and have reset and updated the values object, we then directly insert a phone record this time, using the `ContentResolver insert` method (no helper for this one) **#5**.

After adding data, we next need to take a look at how to update or edit existing data.

UPDATING DATA

To update a row of data you first obtain a `Cursor` row reference to it, and then use the update related `Cursor` methods. This is shown in listing 5.16.

Listing 5.16 The update (edit) details for a `ContentProvider` within the `ProviderExplorer` Activity.

```
private void editContact() {
    ContentResolver resolver = this.getContentResolver();
    ContentValues values = new ContentValues();

    Uri personUri = Contacts.People.CONTENT_URI.buildUpon()
        .appendPath(Long.toString(this.contactId)).build();    #1
    Log.v("ProviderExplorer", "EDIT personUri - " + personUri.toString());

    values.put(Contacts.People.NAME, this.editName.getText().toString());    #2
    resolver.update(personUri, values, null, null);    #3

    values.clear();
    Uri phoneUri = Uri.withAppendedPath(personUri,
        Contacts.Phones.CONTENT_DIRECTORY + "/" + "1");    #4
    values.put(Contacts.Phones.NUMBER, this.editPhoneNumber.getText().toString());
    resolver.update(phoneUri, values, null, null);

    this.startActivity(new Intent(this, ProviderExplorer.class));
}
```

1. Another way to append to an existing `Uri`
2. Update the values to change data
3. Call `resolver.update` to update a record
4. After the person is updated, get the phone `Uri`

In order to update data, we start with the standard `People.CONTENT_URI`, and then append a specific ID path to it using `UriBuilder` **#1**. `UriBuilder` is a very helpful class that uses the builder pattern to allow you to construct and access the various components of a `Uri` object. After we have the `Uri` ready we then update the values data **#2**, and call `resolver.update` to make the update happen **#3**. As you can see, the update process, when using a `ContentResolver`, is pretty much the same as the create process – with the noted exception that the update method allows you to again pass in a where clause, and replacement tokens (SQL style).

For this example, after we have updated the person's name, we then also need to once again obtain the correct `Uri` to also update the associated phone record. We do this by again appending additional `Uri` path data to an object we already have, and we also slap on the specific ID we want **#4**. Outside of example purposes there would be more work to do here in order to determine which phone record for the person needs to be updated (here we are just using the ID 1 as a shortcut).

Though we are only updating single records based on a specific `Uri`, keep in mind here that you can update a set of records using the non specific form of the `Uri` and the `WHERE` clause.

Lastly, in our look at manipulating data through a `ContentProvider`, we need to implement our delete method.

DELETING DATA

In order to delete data we will return to the `ContentResolver` object we used to insert data. This time we will call the delete method though, as seen in listing 5.17.

Listing 5.17 The delete details for a `ContentProvider` within the `ProviderExplorer` Activity.

```
private void deleteContact() {
    Uri personUri = Contacts.People.CONTENT_URI;
    personUri = personUri.buildUpon().
        appendPath(Long.toString(contactId)).build();           #1
    getContentResolver().delete(personUri, null, null);         #2

    startActivity(new Intent(this, ProviderExplorer.class));
}
}
```

1. Use `UriBuilder` to append to the URI path
2. Call `getContentResolver.delete` to delete data

The delete concept is pretty simple, once you have the rest of the process in hand. Again we use the `UriBuilder` approach to setup a `Uri` for a specific record **#1**, and then we obtain a `ContentResolver` reference, this time inline with our delete method call **#2**.

What if the content changes after the fact

When you use a `ContentProvider`, which by definition is accessible by any application on the system, and you make a query you are only getting the current state of the data back. The data could change after your call, so how do you stay up to date? To be notified when a `Cursor` changes you can use the `ContentObserver` API. `ContentObserver` supports a set of callbacks that are invoked when data changes. `Cursor` has register and unregister methods for `ContentObserver` objects.

After having seen how the built-in contacts provider works, you may also want to check out the `android.provider` package in the JavaDocs, as it lists more built-in providers. Now that we have covered a bit about using a built-in provider, and have the CRUD fundamentals under our belt, we will next take a look at the other side of the coin – creating a `ContentProvider`.

5.4.3 Creating a `ContentProvider`

In this section we are going to build a provider that will handle data responsibilities for a generic `Widget` object we will define. This object is simple, with a name, type, category, and other members, and intentionally generic, so we can focus on the how here and not the why (the reasons why you might implement a provider in real life are many, to supply access to your data type from any application, for the purposes of this example though our type will be the mythical `Widget`).

To create a `ContentProvider` you extend that class and implement the required abstract methods. We will see how this is done specifically in a moment. Before getting to that though, it is also first a good idea to define a provider constants class that defines the `CONTENT_URI` and `MIME_TYPE` your provider will support. Additionally, the column names your provider will handle can also be placed here in one class (or you can use multiple nested inner classes, as the built-in contacts system does – we find a flatter approach to be easier to understand).

DEFINING A `CONTENT_URI` AND `MIME_TYPE`

In listing 5.18, as a prerequisite to extending the `ContentProvider` class for a custom provider we have defined needed constants for our `Widget` type.

Listing 5.18 `Widget` provider constants, including columns and URI.

```
public final class Widget implements BaseColumns {                                     #1

    public static final String MIME_DIR_PREFIX = "vnd.android.cursor.dir";           #2
    public static final String MIME_ITEM_PREFIX = "vnd.android.cursor.item";         #3
    public static final String MIME_ITEM = "vnd.msi.widget";                        #4
    public static final String MIME_TYPE_SINGLE = MIME_ITEM_PREFIX + "/" + MIME_ITEM; #4

    Please post comments or corrections to the Author Online forum at
    http://www.manning-sandbox.com/forum.jspa?forumID=411
    Licensed to Thow Way Chiam <ken.ctw@gmail.com>
```

```

public static final String MIME_TYPE_MULTIPLE = MIME_DIR_PREFIX + "/" + MIME_ITEM;           #|4

public static final String AUTHORITY = "com.msi.manning.chapter5.Widget";                 #5
public static final String PATH_SINGLE = "widgets/#";                                  #6
public static final String PATH_MULTIPLE = "widgets";                                   #7
public static final Uri CONTENT_URI =
    Uri.parse("content://" + AUTHORITY + "/" + PATH_MULTIPLE);                           #8

public static final String DEFAULT_SORT_ORDER = "updated DESC";

public static final String NAME = "name";                                               #|9
public static final String TYPE = "type";                                               #|9
public static final String CATEGORY = "category";                                       #|9
public static final String CREATED = "created";                                         #|9
public static final String UPDATED = "updated";                                         #|9
}

```

1. Extend BaseColumns
2. Define MIME prefix for multiple items
3. Define MIME prefix for single item
4. Define MIME type
5. Define authority
6. Define path for a single item
7. Define path for multiple items
8. Define the ultimate CONTENT_URI
9. Define columns

In our `Widget` related provider constants class we first see that we are extending the `BaseColumns` class from Android **#1**. This gives our class a few base constants such as `_ID`. Next we see that we are defining the `MIME_TYPE` prefix for a set of multiple items **#2**, and a single item **#3**. This is outlined in the Android documentation, the convention is that `vnd.android.cursor.dir` represents multiple items, and `vnd.android.cursor.item` represents a single item. Thereafter we define a specific MIME item, and combine it with the single and multiple paths to create two respective `MIME_TYPE` representations **#4**.

Once we have the MIME details out of the way we also define the authority **#5**, and path for both single **#6**, and multiple **#7** items that will be used in the `CONTENT_URI` callers will pass in to use our provider. The multiple item URI is ultimately the one that callers will start from, and the one we publish (they can append specific items from there) **#8**.

After all of the other details we then define column names that represent the variable types in our `Widget` object, which are also going to fields in the database table we will use **#9**. Callers will use these constants to get and set specific fields. That leads us to the next part of the process, extending `ContentProvider`.

EXTENDING CONTENTPROVIDER

In listing 5.19 we see the beginning of our `ContentProvider` implementation class, `WidgetProvider`. In this part of the class we do some housekeeping relating to the database we will use, and the URI we are supporting.

Listing 5.19 The first portion of the `WidgetProvider` `ContentProvider`, which shows the URI matching map and `getType` method.

```

public class WidgetProvider extends ContentProvider {    #1

    private static final String CLASSNAME = WidgetProvider.class.getSimpleName();    #|2
    private static final int WIDGETS = 1;        #|2
    private static final int WIDGET = 2;        #|2
    public static final String DB_NAME = "widgets_db";    #|2
    public static final String DB_TABLE = "widget";    #|2
    public static final int DB_VERSION = 1;        #|2

    private static UriMatcher URI_MATCHER = null;    #3
    private static HashMap<String, String> PROJECTION_MAP;    #4

    private SQLiteDatabase db;    #5

    static {
        WidgetProvider.URI_MATCHER = new UriMatcher(UriMatcher.NO_MATCH);
        WidgetProvider.URI_MATCHER.addURI(Widget.AUTHORITY, Widget.PATH_MULTIPLE, WidgetProvider.WIDGETS);
        WidgetProvider.URI_MATCHER.addURI(Widget.AUTHORITY, Widget.PATH_SINGLE, WidgetProvider.WIDGET);
    }
}

```

```

WidgetProvider.PROJECTION_MAP = new HashMap<String, String>();
WidgetProvider.PROJECTION_MAP.put(BaseColumns._ID, "_id");
WidgetProvider.PROJECTION_MAP.put(Widget.NAME, "name");
WidgetProvider.PROJECTION_MAP.put(Widget.TYPE, "type");
WidgetProvider.PROJECTION_MAP.put(Widget.CATEGORY, "category");
WidgetProvider.PROJECTION_MAP.put(Widget.CREATED, "created");
WidgetProvider.PROJECTION_MAP.put(Widget.UPDATED, "updated");
}

private static class DBOpenHelper extends SQLiteOpenHelper {    #6
    private static final String DB_CREATE = "CREATE TABLE "
        + WidgetProvider.DB_TABLE
        + " (_id INTEGER PRIMARY KEY, name TEXT UNIQUE NOT NULL,
        type TEXT, category TEXT, updated INTEGER, created INTEGER);";

    public DBOpenHelper(final Context context) {
        super(context, WidgetProvider.DB_NAME, null, WidgetProvider.DB_VERSION);
    }

    @Override
    public void onCreate(final SQLiteDatabase db) {
        try {
            db.execSQL(DBOpenHelper.DB_CREATE);
        } catch (SQLException e) {
            Log.e(Constants.LOGTAG, WidgetProvider.CLASSNAME, e);
        }
    }

    @Override
    public void onOpen(final SQLiteDatabase db) {
    }

    @Override
    public void onUpgrade(final SQLiteDatabase db, final int oldVersion, final int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + WidgetProvider.DB_TABLE);
        this.onCreate(db);
    }
}

@Override
public boolean onCreate() {    #7
    DBOpenHelper dbHelper = new DBOpenHelper(this.getContext());
    this.db = dbHelper.getWritableDatabase();

    if (this.db == null) {
        return false;
    } else {
        return true;
    }
}

@Override
public String getType(final Uri uri) {    #8
    switch (WidgetProvider.URL_MATCHER.match(uri)) {
    case WIDGETS:
        return Widget.MIME_TYPE_MULTIPLE;
    case WIDGET:
        return Widget.MIME_TYPE_SINGLE;
    default:
        throw new IllegalArgumentException("Unknown URI " + uri);
    }
}

```

1. Extend ContentProvider
2. Define database constants
3. Use a UriMatcher
4. Include a projection Map
5. Use a SQLiteDatabase reference
6. Create and open database through SQLiteOpenHelper
7. Override onCreate
8. Implement the getType method

Our provider extends `ContentProvider`, which defines the methods we will need to implement **#1**. Then, we use several database related constants to define the database name and table we will

use **#2**. After that we include a `UriMatcher` **#3**, which we will use when matching types in a moment, and a projection `Map` for field names **#4**.

Also we include a reference to a `SQLiteDatabase` object, this is what we will use to store and retrieve the data that our provider handles **#5**. This database is created, opened, and upgraded, using a `SQLiteOpenHelper` in an inner class **#6**. We have seen this helper pattern before, when we worked directly with the database in listing 5.14. The `onCreate` method of our provider is where the open helper is used to setup the database reference.

After our setup related steps we then come to the first method a `ContentProvider` requires us to implement, `getType` **#8**. This method is used by the provider to resolve each passed in `Uri`, to determine if it is supported, and if so which type of data the current call is requesting (a single item, or the entire set). The `MIME_TYPE` String we return here is based on the constants we defined in our `Widget` class.

The next steps we need to cover are the remaining required methods to implement to satisfy the `ContentProvider` contract. These methods, which are shown in listing 5.20, correspond to the CRUD related activities used with the contacts provider in the previous section: `query`, `insert`, `update`, and `delete`.

Listing 5.20 The second portion of the `WidgetProvider` `ContentProvider`, which shows `query`, `insert`, `update`, and `delete` implementations.

```
@Override
public Cursor query(final Uri uri, final String[] projection,
    final String selection, final String[] selectionArgs,
    final String sortOrder) {
    SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();    #1
    String orderBy = null;

    switch (WidgetProvider.URI_MATCHER.match(uri)) {    #2
    case WIDGETS:
        queryBuilder.setTables(WidgetProvider.DB_TABLE);
        queryBuilder.setProjectionMap(WidgetProvider.PROJECTION_MAP);
        break;
    case WIDGET:
        queryBuilder.setTables(WidgetProvider.DB_TABLE);
        queryBuilder.appendWhere("_id=" + uri.getPathSegments().get(1));
        break;
    default:
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    if (TextUtils.isEmpty(sortOrder)) {
        orderBy = Widget.DEFAULT_SORT_ORDER;
    } else {
        orderBy = sortOrder;
    }

    Cursor c = queryBuilder.query(this.db, projection, selection, selectionArgs,
        null, null, orderBy);    #3
    c.setNotificationUri(this.getContext().getContentResolver(), uri);    #4
    return c;
}

@Override
public Uri insert(final Uri uri, final ContentValues initialValues) {
    long rowId = 0L;
    ContentValues values = null;    #5

    if (initialValues != null) {
        values = new ContentValues(initialValues);
    } else {
        values = new ContentValues();
    }

    if (WidgetProvider.URI_MATCHER.match(uri) != WidgetProvider.WIDGETS) {
        throw new IllegalArgumentException("Unknown URI " + uri);
    }

    Long now = System.currentTimeMillis();

    . . . omit defaulting of values for brevity
```

```

        rowId = this.db.insert(WidgetProvider.DB_TABLE, "widget_hack", values);    #6

        if (rowId > 0) {
            Uri result = ContentUris.withAppendedId(Widget.CONTENT_URI, rowId);    #7
            this.getContext().getContentResolver().notifyChange(result, null);    #8
            return result;
        }
        throw new SQLException("Failed to insert row into " + uri);
    }

    @Override
    public int update(final Uri uri, final ContentValues values, final String selection,
        final String[] selectionArgs) {    #9
        int count = 0;
        switch (WidgetProvider.URI_MATCHER.match(uri)) {
            case WIDGETS:
                count = this.db.update(WidgetProvider.DB_TABLE, values, selection, selectionArgs);
                break;
            case WIDGET:
                String segment = uri.getPathSegments().get(1);
                String where = "";
                if (!TextUtils.isEmpty(selection)) {
                    where = " AND (" + selection + ")";
                }
                count = this.db.update(WidgetProvider.DB_TABLE, values, "_id="
                    + segment + where, selectionArgs);
                break;
            default:
                throw new IllegalArgumentException("Unknown URI " + uri);
        }
        this.getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }

    @Override
    public int delete(final Uri uri, final String selection, final String[] selectionArgs) {    #10
        int count;

        switch (WidgetProvider.URI_MATCHER.match(uri)) {
            case WIDGETS:
                count = this.db.delete(WidgetProvider.DB_TABLE, selection, selectionArgs);
                break;
            case WIDGET:
                String segment = uri.getPathSegments().get(1);
                String where = "";
                if (!TextUtils.isEmpty(selection)) {
                    where = " AND (" + selection + ")";
                }
                count = this.db.delete(WidgetProvider.DB_TABLE, "_id="
                    + segment + where, selectionArgs);
                break;
            default:
                throw new IllegalArgumentException("Unknown URI " + uri);
        }
        this.getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }
}

```

1. Use a query builder in the query method
2. Setup the query based on the URI passed in
3. Perform the query to get a Cursor
4. Set the notification Uri on the Cursor
5. Use ContentValues in the insert method, with default values
6. Call the database insert method
7. Get the Uri to return
8. Notify listeners that data was inserted
9. Provide an update method
10. Provide a delete method

In the last part of our `WidgetProvider` class we see how the `ContentProvider` methods are implemented. These are the same methods – different provider - that we called earlier in our `ProviderExplorer` example.

First we use a `SQLQueryBuilder` inside the query method to append the projection mapped passed in **#1**, and any SQL clauses, along with the correct URI based on our matcher **#2**, before we make the actual query and get a handle on a `Cursor` to return **#3**.

At the end of the query method we then use the `setNotificationUri` method to set the returned `Uri` to be watched for changes **#4**. This is an event based mechanism that can be used to keep track of when `Cursor` data items are changed, regardless of how changes are made.

Next we see the insert method, where the passed in `ContentValues` object is validated and populated with default values if not present **#5**. After the values are ready, we then call the database insert method **#6**, and get the resulting `Uri` to return with the appended ID of the new record **#7**. After the insert is complete we see another notification system in use, this time for `ContentResolver`. Here, since we have made a data change, we are informing the `ContentResolver` what happened, so that any registered listeners can be updated **#8**.

After the insert method is complete we then come to the update **#9** and delete methods **#10**. These repeat many of the concepts we have already seen. First the match the `Uri` passed in to a single element or the set, and then they call the respective update and delete methods on the database object. Again at the end of these methods we notify listeners that the data has changed.

Implementing the needed provider methods completes our class. This provider, which now serves the `Widget` data type, will be able to be used from any application to query, insert, update, or delete data, once we have registered it as a provider with the platform. This is done using the application manifest, which we will look at next.

PROVIDER MANIFESTS

In order for the platform to be aware of the content providers that are available, and what data types they represent, they must be defined in an application manifest file – and then installed on the platform. The manifest for our provider is shown in listing 5.21.

Listing 5.21 The `AndroidManifest.xml` file for the `WidgetProvider` provider application.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.chapter5.widget">
    <application android:icon="@drawable/icon" android:label="@string/app_short_name">
        <activity android:name=".WidgetExplorer" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <provider android:name="WidgetProvider"
            android:authorities="com.msi.manning.chapter5.Widget" /> #1
    </application>
</manifest>
```

1. Use the provider element to define the class and authority

The significant part of the manifest concerning content provider support is the `<provider>` element **#1**. This is used to define the class that implements the provider, and associate a particular authority with that class. This element does support additional attributes we are not using here. Most of the attributes we aren't using are very straightforward, such as name and label, but there are two that relate to synchronization that deserve a bit more mention.

`ContentProvider` instances in Android can be synchronized using the `android:syncable` attribute. This can be set to true, and provider access will be synchronized. Also, the `android:multiprocess` attribute allows a `ContentProvider` to be created once for each process that invokes it, effectively creating multiple instances of itself (as opposed to the default behavior of one separate process). You can use this in special cases when multiple instances of a provider are acceptable, and when you want to avoid IPC overhead within.

That completes our exploration of using and building `ContentProvider` classes. And, with that we have taken a look at all of the ways to store and retrieve data on the Android platform.

5.5 Summary

From a simple `SharedPreferences` mechanism that saves data backed by files, to file storage itself, databases, and finally the concept of a `ContentProvider` - Android provides myriad ways for applications to retrieve and store data.

As we discussed in this chapter, several of these means are intended to be used across application and process boundaries, and several aren't. Here we saw that `SharedPreferences` can be created with a permissions mode, allowing the flexibility to keep things private, or can be shared globally with read only or read-write permissions. Preferences are stored as simple XML files in a specific path on the device per application, as are other file resources you can create and read yourself. However, files cannot be shared across applications, as there is no way to pass in the starting path based on the `Context` (the application package-path). This makes the file system, which we also looked at in this chapter, good for handling some levels of application local state and data persistence, but not appropriate for more broad reaching goals.

After filesystem access the next level of storage Android provides is a relational database system based on SQLite. This system is lightweight, speedy, and very capable – but again, as we saw here, it is intended only for local data persistence within a single application. Beyond storing and retrieving data locally you can still use a database, but you need to expose an interface through a `Service` (as we saw in chapter 4), or a `ContentProvider`. Providers, which we covered in this chapter, expose data types and operations through a URI based approach.

All in all in this chapter we examined each of the data paths available to an Android application. We did this here by using several small, focused sample applications to utilize preferences and the file system, and we looked at more of the `WeatherReporter` sample application that we began in the last chapter. This Android application uses a SQLite database to access and persist data.

Expanding our Android horizons beyond data, and beyond foundational concepts we have already seen in earlier chapters such as views, intents, and services, we will move on general networking in the next chapter. There we will cover networking basics, the networking APIs Android provides, and we will expand on the data concepts we have covered here to include the network itself as a data source.

6

Networking

Every mobile provider supports both voice and data networks, of one or more types. The interesting part with an Android enabled device is really the data network, and the power to link the data available on the network to interesting applications. Those applications can then be built with the open intent and service based approach we have learned about in previous chapters. That approach combines built-in (or custom) intents such as fully capable web browsing, with access to hardware components such as a 3D capable graphics subsystem, a GPS receiver, a camera, removable storage, and more. It is this combination of platform, hardware capability, software architecture, and access to network data that makes Android so compelling.

This is not to say that the voice network is not also important (and we will cover telephony explicitly in Chapter 7), but rather it is simply an admittance that voice is almost a commodity, and data is where we will focus when talk about the “network”.

In terms of the data network, Android provides access in several ways: mobile IP networks, WiFi, and Bluetooth. Here we are going to concentrate on getting our Android applications to communicate using IP network data, using several different approaches. We will cover a bit of networking background, and then we will deal with some Android specifics as we explore sockets, the `java.net` package, and the `org.apache.httpclient` package. We will briefly peek at the `android.net` package too, but is really more about internal networking and details (such as the `Uri.Builder` class we have already used with intents), and general connectivity properties, than it is about talking to a data network.

In terms of connectivity properties, in this chapter we will look at using the `ConnectivityManager` class to determine when the network connection is active, and what type of connection it is (mobile or WiFi), but we specifically won't get into browsing wireless hotspots, switching networks, or any other details specific to one network type or another. This is because many of the details surrounding these topics are not yet completely public, even in the 1.0 SDK. Additionally, if when things do settle down relating to these topics we still hope many of the specifics will be handled by the platform so that our applications won't need to get deeply involved.

As for Bluetooth, we unfortunately won't be dealing with it either. Yes, this is a downer. Bluetooth is an important way to connect to many local devices and transfer data (a different kind of “network”). Bluetooth also opens up many application possibilities – but again, the APIs just aren't there yet. Apparently there are many difficulties in terms of emulating Bluetooth (and WiFi for that matter) on top of the native hardware stacks of the various platforms Android supports.

So with our focus here set to the data, and specifically to using the IP network with our applications, we are going to build another sample application to explore the possibilities. This aptly named application, `NetworkExplorer`, will look at the various ways to communicate with the

network in Android, and include some handy HTTP utilities. NetworkExplorer will ultimately have multiple screens that exercise different networking techniques, as seen in figure 6.1.



Figure 6.1 The NetworkExplorer application we will build to cover networking topics.

After we cover general IP networking with regard to Android, we will then also discuss turning the server side into a more robust API itself by using “web services.” On this topic we will work with Plain Old XML over HTTP (POX), and Representational State Transfer (REST). And, we will also discuss the Simple Object Access Protocol (SOAP). We will address the pros and cons of the various approaches and why you might want to choose one method over another for an Android client.

Before we dive headlong into the details of networked Android applications, we will begin with a brief overview of networking basics. If you are already a networking expert you can certainly skip ahead to section 6.2, but it is important to have this foundation if you think you need it, and we promise to keep it short either way.

6.1 An overview of networking

A group of interconnected computers is a network. Over time networking has grown from something that was once only available to governments and large organizations, to the almost ubiquitous, and truly amazing, Internet. Though the concepts are simple – allow computers to communicate - underneath networking does involve some advanced technology. We won't get into great detail here, though we will cover the core tenets as a background to the general networking we will do in the remainder of this chapter.

6.1.1 Networking basics

A large percentage of the time the APIs you will use to program Android applications will abstract the underlying network details. This is good. This is how the APIs and the network protocols themselves are designed so that you can focus on your application and not worry about routing and reliable packet delivery and so on.

Nevertheless, it helps to have some understanding of the way a network works so that you can better design and troubleshoot your applications. To that end, here we are going to blaze through some general networking concepts, with a Transmission Control Protocol/Internet Protocol (TCP/IP) bent. We will begin with nodes, layers, and protocols.

NODES

The basic idea behind a network is that data is sent between connected devices with particular addresses. Connections can be made over wire, over radio waves, and so on. Each addressed device is known as a node. A node can be a mainframe, a PC, a fancy toaster, or any other device with a network stack and connectivity, such as an Android enabled handheld.

Nodes in a TCP/IP network are identified by their IP addresses. We will learn a bit more about IP addressing in a moment, for now we need to briefly cover protocols and the network layer abstraction they represent. The protocol family that is used to power the Internet (and many other private networks), as we learned in our tour of history, is the TCP/IP stack.

Each protocol in this family has a specific role. These roles are often described using the metaphor of network “layers.”

LAYERS AND PROTOCOLS

Protocols are often layered on top of one another as they handle different levels of responsibility, and TCP/IP is no exception. To send email for example, the Simple Message Transport Protocol (SMTP) is used. This protocol defines the format and sequence for data that is required to connect to a mail server and send a message. SMTP itself though knows nothing about the delivery of data. For that SMTP runs on top of TCP or the Uniform Datagram Protocol (UDP). These in turn don't deal with the details of addressing and routing, which are left to the underlying IP layer.

Layers are described a bit differently depending on the source (with some sources using more fine grained divisions than others), but the general idea is that there are four main layers: Link, Internet, Transport, and Application. The Link layer deals with physical media and the lowest level of data transmission, the Internet layer handles addressing and routing, the Transport layer copes with higher level data transmission details, and the Application layer provides support for specific application details. The diagram in figure 6.2 demonstrates these layers by depicting a subset of the protocols that fall within each.

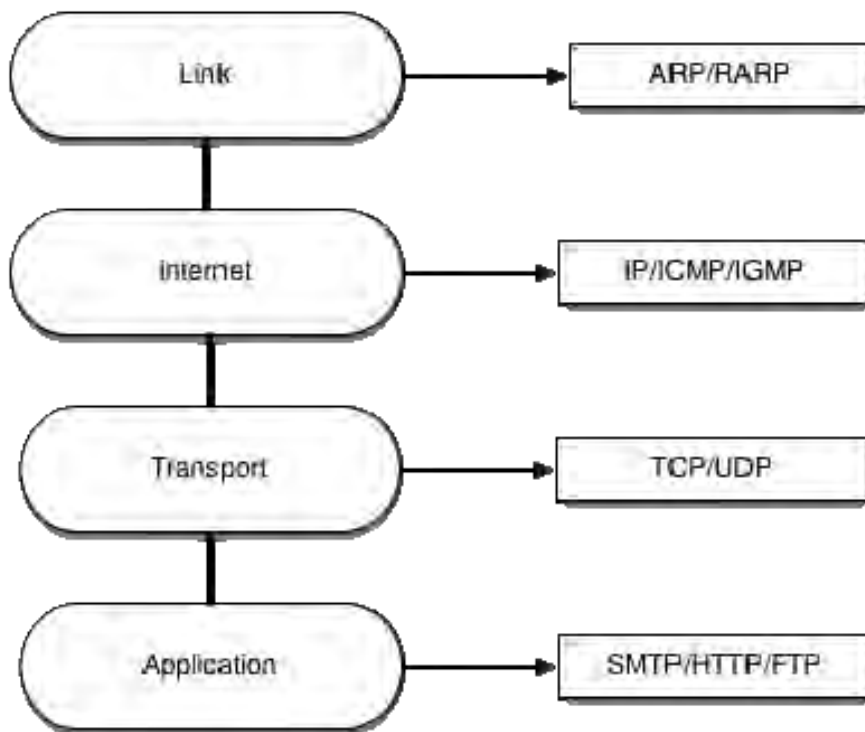


Figure 6.2 Different networking protocols are applied at different logical layers

Layers therefore are an abstraction of the different levels of a network protocol stack. Some of the important specific protocols in the TCP/IP stack are also worth covering in a bit more detail, beginning with IP.

IP

The Internet Protocol itself (IP) controls the addressing system, and is in charge of delivering data packets. IP defines packets as “datagrams,” and dictates that they have a header between 20 and 60 bytes, and a payload (data section) of up to 65,515 bytes. IP addresses, a ubiquitous term with which almost everyone is no doubt familiar, are assigned and divided up into classes which are identified using 4 byte numbers.

IP addresses are typically written using the decimal notation for each byte, separated into 4 sections by dots (representing each octet, or 8 bit section). For example “127.0.0.1”. Certain address classes have special roles and meaning. For example, 127. always identifies a “loopback” or local address on every machine, this class does not communicate with any other devices (it can be used internally, on a single machine, only). Addresses that begin with 10. or 192. are “not routable,” meaning they can communicate with other devices on the same local network segment, but cannot connect to other segments.

The routing of packets on an IP network, how packets traverse the network and go from one segment to another based on the source and destination address, is handled by a device known as a router. Many people are familiar with the term router, if not the underlying details. From small offices, and personal homes, to large corporations and even inter-ISP Internet exchange points, cross network traffic is handled by routing it based on the address information.

IP address numbers are turned into human readable names through the Domain Name System (DNS). DNS is basically just a map of numbers to names. For example, the IP address 74.125.45.147 is assigned to Google, and mapped to the host and domain name “www.google.com.”

Any time a name is used on an IP network (actually in the stack before it hits the network), it is translated into it's number, and vice versa, as needed. Because IP controls the addressing and delivery of packets, and not much else, more is needed to turn the swarm of datagrams on a given network segment into a controlled and usable stream of information.

TCP

The Transmission Control Protocol (TCP), part of the Transport layer, provides needed “control” over the wild-west show that any IP network segment really is underneath it all. Packets on any given wire may be coming and going in no particular order and with no particular priority. TCP defines the scheme that allows packets to be re-ordered on the receiving end, and allows for the acknowledgment of each packet as having been received (certified mail is the common analogy, the sender gets a confirmation that the receiver was home and signed for the package).

To accomplish this sense of order and reliability on top of IP, TCP adds further overhead to the datagram (the order, and more data sent back to the sender to acknowledge receipt, etc). Because of this, and because sometimes this overhead is unnecessary, there is another transmission related protocol available in the TCP/IP stack as well.

UDP

The Uniform Datagram Protocol, another member of the Transport layer, can be used on TCP/IP networks when you don't want to guarantee the delivery of packets, or the order they are read in. In situations where speed is more important than reliability, UDP can be used in favor of TCP.

When working with Android, and Java related APIs in general, again, you won't typically need to delve into the details of any of the lower level protocols, but you do need to know the major differences we have outlined here, and you need to be well versed in addressing. We will learn more about exactly how addressing is applied later, for now we want to wrap up our general networking overview with a bit about clients and servers.

6.1.2 Clients and Servers

Anyone who has ever user a web browser is familiar with the client server computing model. Data, in one format or another, is stored on a centralized, powerful server. Clients then connect to that server using a designated protocol, such as the Hypertext Trasnport Protocol (HTTP) for the web (which we will address briefly in the next section), to retrieve the data and work with it.

This pattern is of course much older than the web, and has been applied for everything from completely “dumb” terminals connecting to mainframes, to modern desktop applications that connect to a server for only a portion of their purpose (such as with iTunes, which is primarily a media organizer and player, but also has a “store” where customers can connect to the server to get new content). In any case, the concept is the same – the client makes a type of request to the server, and the server responds.

In order to facilitate many client requests, often for different purposes, coming in nearly simultaneously to a single IP address, modern server operating systems use the concept of ports. Ports are not physical, they are simply a representation of a particular area of the computer's memory. A server can “listen” on multiple designated ports at a single address. For example, one port for sending email, one port for web traffic, two ports for file transfer, and so on. Every computer with an IP address also supports a range of thousands of ports to enable multiple “conversations” to happen at the same time.

Ports are divided into three ranges:

- Well Known Ports – 0 thru 1023
- Registered Ports – 1024 thru 49151
- Dynamic and or Private Ports – 49152 thru 65535

The “Well Known” ports are all published (controlled by ICANN/IANA) and are just that, well known. HTTP is port 80 (and HTTP Secure, or HTTPS, is port 443), File Transport Protocol (FTP) is ports 20 (control) and 21 (data), Secure Shell (SSH) is port 22, SMTP is port 25, Post Office Protocol (POP) is port 110, and Internet Message Access Protocol (IMAP) is port 143 – and so on. All of these are further protocols, at the Application layer, in the TCP/IP family.

Beyond the Well Known ports, the Registered ports are still controlled and published, but for more specific purposes. Often these ports are used for a particular application or company - for

example, MySQL is port 3306. For a complete list of Well Known and Registered ports see the ICANN port-numbers document: <http://www.iana.org/assignments/port-numbers>.

Lastly the dynamic or private ports are intentionally unregistered as they are used by the TCP/IP stack to facilitate communication. These ports are dynamically registered on each computer and used in the conversation. Dynamic port 49500 for example might be used to handle sending a request to a web server and dealing with the response. Once the conversation is over, the port is reclaimed and can be reused, locally, for any other data transfer.

Clients and servers therefore communicate as nodes with addresses, using ports, on a network that supports various protocols. With that super-short tour of networking in general behind us, we next turn to our first Android networking task, determining the state of the connection.

6.2 Checking the network status

Android provides a host of utilities to determine the device configuration, and the status of various services – including the network. The `ConnectivityManager` class is what you will typically use in order to determine whether or not there is network connectivity, and get notifications of network changes. Listing 6.1, a portion of the main `Activity` in the `NetworkExplorer` application, demonstrates basic usage of the `ConnectivityManager`.

Listing 6.1 The `onStart` method of the `NetworkExplorer` Main Activity, demonstrating usage of the `ConnectivityManager`.

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

        ConnectivityManager cMgr = (ConnectivityManager)
this.getSystemService(Context.CONNECTIVITY_SERVICE);           #1
        NetworkInfo netInfo = cMgr.getActiveNetworkInfo();          #2
        this.status.setText(netInfo.toString());
    }
```

1. Obtain the manager from the Context
2. Get the `NetworkInfo`

This short and sweet example shows that you can get a handle to the `ConnectivityManager` through the context's `getSystemService` method, using the `CONNECTIVITY_SERVICE` constant. Once you have the manager you can then obtain network information via the `NetworkInfo` object. The `NetworkInfo` object includes both detailed and basic network information. The Android documentation recommends that most applications just use the basic information (you can see the detailed information, yes, but there is not a lot you can do with it or about it at present). The various states include values such as `CONNECTING`, `CONNECTED`, `DISCONNECTED`, `UNAVAILABLE`, and so on.

Once you know that you are connected, either via mobile or Wi-Fi, you can then go to town and use the IP network. For the purposes of our `NetworkExplorer` application, we are going to start with the most rudimentary connection, a raw socket, and work our way up to HTTP and web services.

6.3 Communicating with a server socket

A server socket is a stream that you can read or write raw bytes to, at a specified IP address and port. This lets the you deal with data, and not worry about media types, packet sizes, and so on. This is yet another network abstraction intended to make the job of the programmer a bit easier. This philosophy, everything should look like file I/O to the developer, comes from the early UNIX days, but has been adopted by all major operating systems in use today (Windows, Mac, and Linux, have all adopted the socket concept).

We will move on to higher levels of network communication in a bit, but first we will start with a raw socket. For that we need a “server” listening on a particular port. The `EchoServer` code shown in listing 6.2 fits the bill.

Listing 6.2 A simple echo server for demonstrating socket usage.

```
public final class EchoServer extends Thread {

    private static final int PORT = 8889;

    private EchoServer() {
    }

    public static void main(final String args[]) {
        EchoServer echoServer = new EchoServer();
        if (echoServer != null) {
            echoServer.start();
        }
    }

    public void run() {                                     #1
        try {
            ServerSocket server = new ServerSocket(PORT, 1); #2

            while (true) {
                Socket client = server.accept();           #3
                System.out.println("Client connected");

                while (true) {

                    BufferedReader reader = new BufferedReader(
                        new InputStreamReader(client.getInputStream())); #4
                    System.out.println("Read from client");
                    String textLine = reader.readLine() + "\n";

                    if (textLine.equalsIgnoreCase("EXIT\n")) { #5
                        System.out.println("EXIT invoked, closing client");
                        break;
                    }

                    BufferedWriter writer = new BufferedWriter(
                        new OutputStreamWriter(client.getOutputStream())); #6
                    System.out.println("Echo input to client");
                    writer.write("ECHO from server: " + textLine, 0, textLine
                        .length() + 18);
                    writer.flush();
                }
                client.close();
            }
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

1. The Thread run method, which invoked when start is called.
2. The server, a java.net.ServerSocket
3. Each client that connects is handled with a java.net.Socket
4. The client input is read using a BufferedReader
5. If the input is EXIT, break the loop
6. The server echo is sent back to the client with a BufferedWriter

The EchoServer class we are using is fairly basic Java I/O. It extends Thread, and implements run #1, so that each client that connects can be handled in its own context. Then we use a ServerSocket #2 to “listen” on a defined port. Each client is then an implementation of a Socket #3. The client input is fed into a BufferedReader that each line is read from #4. The only special consideration this simple server has is that if the input is “EXIT,” it breaks the loops and exits #5. If the input does not prompt an exit, then the server echoes the input back to the client’s OutputStream with a BufferedWriter #6.

This is a good, albeit intentionally very basic, representation of a what a server does,. It handles input, usually in a separate thread, and then responds to the client based on the input. To try this server out, before using Android, you can telnet to the specified port (after the server is running, of course) and type some input – if all is well it will echo the output.

To run the server you just need to invoke it locally with Java. It has a main method, so it will run on its own, just start it from the command line or from your IDE. Importantly, when you connect to servers from Android, this or any other, you need to connect to the IP address of the host you run the server process on – not the loopback (not 127.0.0.1). The emulator thinks of itself as 127.0.0.1, so use the non-loopback address of the server host when you attempt to connect from Android.

The client portion of this example is where NetworkExplorer itself begins, with the `callSocket` method of the SimpleSocket Activity seen in listing 6.3.

Listing 6.3 An Android client invoking a raw socket server resource, the echo server.

```
public class SimpleSocket extends Activity {  
    . . .  
    private void callSocket(final String ip, final String port, final String socketData) {  
        Socket socket = null;  
        BufferedWriter writer = null;  
        BufferedReader reader = null;  
        try {  
            socket = new Socket(ip, Integer.parseInt(port));           #1  
            writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())); #2  
            reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));      #3  
  
            String input = socketData;  
            writer.write(input + "\n", 0, input.length() + 1);         #4  
            writer.flush();  
  
            String output = reader.readLine();                           #5  
            this.socketOutput.setText(output);  
  
            // send EXIT and close  
            writer.write("EXIT\n", 0, 5);  
            writer.flush();  
        }  
        // catches and reader, writer, and socket closes omitted for brevity  
    }  
    1. Create a new client Socket  
    2. Establish a BufferedWriter for socket input  
    3. Establish a BufferedReader for socket output  
    4. Write to the socket  
    5. Get the socket output
```

Here we create a `Socket` to represent the client side of our connection #1, and we establish a writer for the input #2, and a reader for the output #3. With the housekeeping taken care of, we then write to the socket #4, which communicates with the server, and get the output and set its value into our output `EditText` view component #5.

A socket is probably the lowest level networking usage in Android you will encounter. Using a raw socket, while abstracted a great deal, still leaves many of the details up to you (especially server side details, threading, and queuing, etc.). Though you may run up against situations in which you either have to use a raw socket (the server side is already built), or you elect to use them for a quick and dirty approach, higher level solutions such as leveraging HTTP have decided advantages.

6.4 Working with HTTP

As we saw in the previous section, you can use a raw socket to transfer IP data to and from a server with Android. This is an important approach to be aware of so that you know you have that option, and so that you understand a bit about the low level details. Nevertheless, you may want to avoid this technique where possible, and instead take advantage of existing server products to send your data. The most common way to do this is to use a web server and leverage HTTP.

Here we are going to take a look at making HTTP requests from an Android client, to an HTTP server. We will let the HTTP server handle all the socket details, and we will focus on our application.

The HTTP protocol itself is fairly involved. If you are unfamiliar with it, and want the complete details, they are readily available via RFCs (such as for version 1.1: <http://www.w3.org/Protocols/rfc2616/rfc2616.html>). The short story though is that the protocol is stateless and involves several different methods that allow users to make “requests” to servers, and those servers return “responses.” The entire web is of course based on HTTP. Beyond the most basic concepts, there are various ways to pass data into and out of requests and responses, and authenticate with servers, and so on. Here we are going to use some of the most common methods and concepts to talk to network resources from Android applications.

To begin we will retrieve data using HTTP GET requests to a simple HTML page using the standard `java.net` API. From there we will take a look at using the Android included Apache `HttpClient` API. After we use `HttpClient` directly to get a feel for it, we will also make a helper class, `HttpRequestHelper`, that we can use to simplify the process and encapsulate the details. This class, and the Apache networking APIs in general, have a few advantages over rolling your own networking with `java.net`, as we shall see. Once the helper class is in place, we will then use it to make additional HTTP and HTTPS requests, both GET and POST, and we will look at basic authentication.

Our first HTTP request will be a basic HTTP GET call using an `URLConnection`.

6.4.1 Simple HTTP and `java.net`

The most basic HTTP request method is a GET. In this type of request any data that is sent is embedded in the URL using the query string. The next class in our `NetworkExplorer` application has an Activity that demonstrates this, `SimpleGet`, which is shown in listing 6.4.

Listing 6.4 The `SimpleGet` Activity showing use of `java.net.URLConnection`.

```
public class SimpleGet extends Activity {

    private EditText getInput;
    private TextView getOutput;
    private Button getButton;

    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.simple_get);

        getInput = (EditText) this.findViewById(R.id.get_input);
        getOutput = (TextView) this.findViewById(R.id.get_output);
        getButton = (Button) this.findViewById(R.id.get_button);

        getButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                getOutput.setText("");
                String output = SimpleGet.this.
                    getHttpResponse(getInput.getText().toString());    #1
                if (output != null) getOutput.setText(output);
            }
        });

        private String getHttpResponse(String location) {
            String result = null;
            URL url = null;

            try {
                url = new URL(location);    #2
            } catch (MalformedURLException e) {
                Log.e(Constants.LOGTAG, "ERROR", e);
            }

            if (url != null) {
                try {
                    HttpURLConnection urlConn = (HttpURLConnection) url
                        .openConnection();    #3
                    BufferedReader in = new BufferedReader(new InputStreamReader(
                        urlConn.getInputStream()));    #4
                    String inputLine;

                    int lineCount = 0; // limit the lines for the example
                    while ((lineCount < 10) && ((inputLine = in.readLine()) != null)) {    #5
                        lineCount++;
                        result += "\n" + inputLine;    #6
                    }

                    in.close();    #7
                    urlConn.disconnect();    #7

                } catch (IOException e) {
                    Log.e(Constants.LOGTAG, "ERROR", e);
                }
            } else {
                Log.e(Constants.LOGTAG, " url NULL");
            }
        }
    }
}
```

```

        return result;
    }
}

```

1. Invoke the `getHttpResponse` method
2. Construct a `URL` object
3. Open a connection using `HttpURLConnection`
4. Create a `BufferedReader` for output
5. Read data
6. Append to the result
7. Close the reader and connection

In order to get an HTTP response and show the first few lines of it in our `SimpleGet` class we are calling a `getHttpResponse` method that we have built **#1**. Within this method we construct a `java.net.URL` object **#2**, which takes care of many of the details for us, and then we open a connection to a server using an `HttpURLConnection` **#3**.

We then use a `BufferedReader` **#4** to read data from the connection one line at a time **#5**. Keep in mind that as we are doing this, we are using the same thread as the UI and therefore blocking the UI. This isn't a good idea. We are only doing this here to demonstrate the network operation, we will learn more about how to use a separate thread for this coming up. Once we have the data we append it to the result `String` that our method returns **#6**, and then lastly we close the reader and connection **#7**. Using the plain and simple `java.net` support that has been ported to Android this way provides quick and dirty access to HTTP network resources.

Communicating with HTTP this way is fairly easy, but it can quickly get cumbersome when you need to do more than just retrieve simple data, and as noted the blocking nature of the call is bad form. We could get around some of the problems with this approach on our own, by spawning separate threads ourselves and keeping track of them, and writing our own small framework/API structure around that concept for each HTTP request, but we don't really have to. Fortunately, Android provides another set of APIs in the form of the Apache `HttpClient` library that abstract the `java.net` classes further that and designed to offer some more robust HTTP support, and help handle the separate thread issue.

6.4.2 Robust HTTP with `HttpClient`

To get started with `HttpClient` we are going to look at using some of the core classes to perform HTTP GET and POST method requests. Here we will concentrate on making network requests in a `Thread` separate from the UI, using a combination of the Apache `ResponseHandler` and Android `Handler` (for different, but related, purposes, as we shall see). Listing 6.5 shows our first example of using the `HttpClient` API.

Listing 6.5 Using the Apache `HttpClient` library with an Android `Handler`, and Apache `ResponseHandler`.

```

. . . . .

private final Handler handler = new Handler() {      #1
    @Override
    public void handleMessage(final Message msg) {
        progressDialog.dismiss();
        String bundleResult = msg.getData().getString("RESPONSE"); #2
        ApacheHTTPS.Simple.this.output.setText(bundleResult);      #2
    }
};

. . . onCreate omitted for brevity

private void performRequest() {

    final ResponseHandler<String> responseHandler = new ResponseHandler<String>() { #3
        public String handleResponse(HttpResponse response) {      #4
            StatusLine status = response.getStatusLine();
            HttpEntity entity = response.getEntity();
            String result = null;
            try {
                result = StringUtils.inputStreamToString(entity.getContent()); #5
                Message message = new Message();
                Bundle bundle = new Bundle();
                bundle.putString("RESPONSE", result);
                message.setData(bundle);
                handler.sendMessage(message); #6
            } catch (IOException e) {
                Log.e("Please post comments or corrections to the Author's Online Forum at", e);
            }
        }
    };

    . . .
}

```

Please post comments or corrections to the Author's Online Forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>


```

        }
        return result;
    }
};

this.progressDialog = ProgressDialog.show(this, "working . . .", "performing HTTP request");

new Thread() {
    #7
    @Override
    public void run() {
        try {
            DefaultHttpClient client = new DefaultHttpClient();
            HttpGet httpMethod = new HttpGet(ApacheHTTPSimple.
                this.urlChooser.getSelectedItem().toString()); #8
            client.execute(httpMethod, responseHandler); #9
        } catch (ClientProtocolException e) {
            Log.e(Constants.LOGTAG, " " + ApacheHTTPSimple.CLASSTAG, e);
        } catch (IOException e) {
            Log.e(Constants.LOGTAG, " " + ApacheHTTPSimple.CLASSTAG, e);
        }
    }
}.start();
}

```

1. Create an Android Handler
2. Use the Handler to update the UI with Message data
3. Create a ResponseHandler to make an asynchronous HTTP call
4. Implement the onResponse callback method
5. Get the HTTP response payload
6. Send a message to the UI handler with the needed response data
7. Use a separate Thread to make the HTTP call
8. Create an HttpGet method object with a String URL
9. Execute the HTTP request using HttpClient

The first thing we do in our initial `HttpClient` example is create a `Handler` that we can send messages to from other threads **#1**. This is the same technique we have seen in previous examples, and is used allow background tasks to send `Message` objects to hook back into the main UI thread **#2**. After we create an Android `Handler`, we then also create an Apache `ResponseHandler` **#3**. This class can be used with `HttpClient` HTTP requests to pass in as a callback point. When an HTTP request that is fired by `HttpClient` completes, it will call the `onResponse` method (if a `ResponseHandler` is used) **#4**. When the response does come in, we then get the payload using the `HttpEntity` the API returns **#5**. This in effect allows the HTTP call to be made in an asynchronous manner – we don't have to block and wait the entire time between when the request is fired, and when it completes. The relationship of the request, response, `Handler`, `ResponseHandler`, and separate threads is diagrammed in figure 6.3.

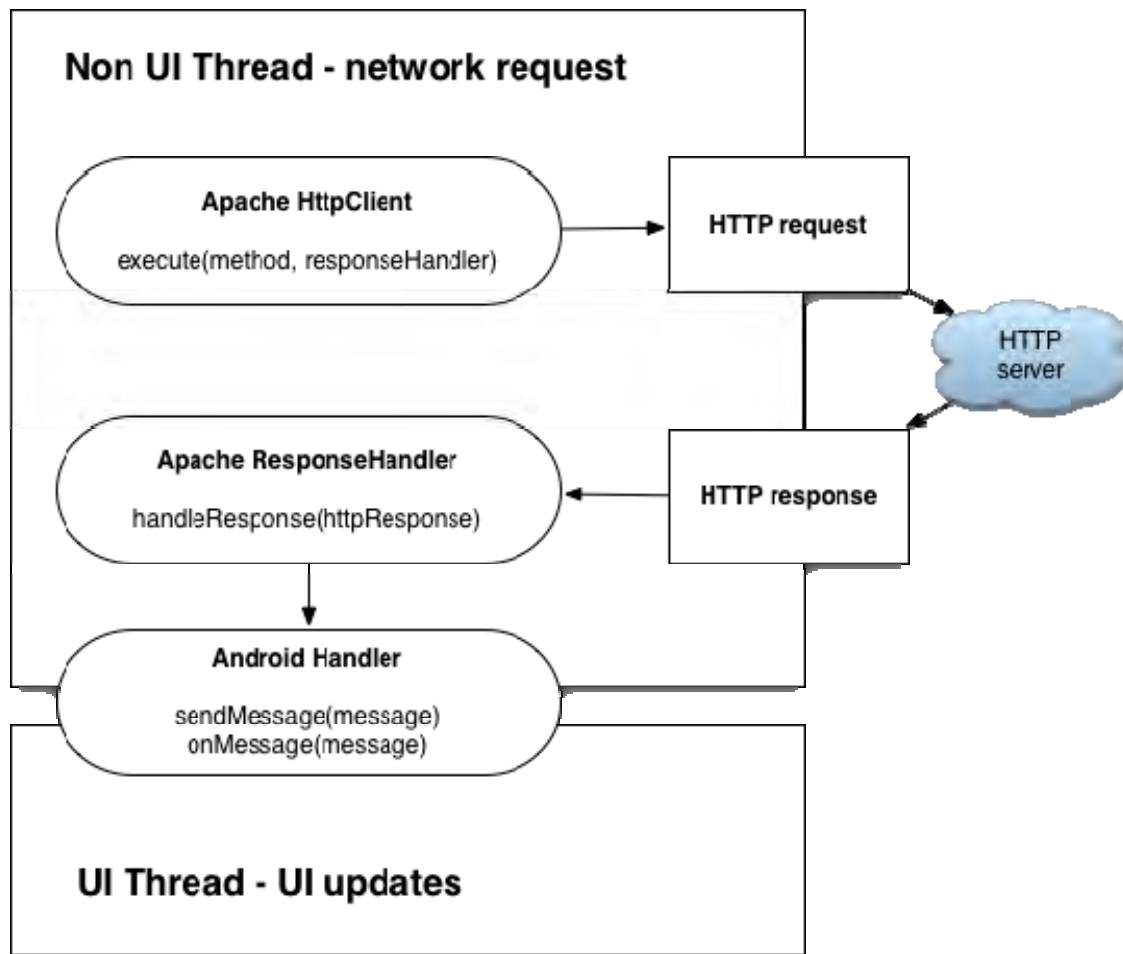


Figure 6.3 HttpClient ResponseHandler and Android Handler relationship diagram.

Now that we have seen `HttpClient` at work, and understand the basic approach, the next thing we will do is encapsulate a few of the details into a convenient helper class so that we can call it over and over without having to repeat a lot of the setup.

6.4.3 Creating an HTTP and HTTPS helper

The next Activity in our `NetworkExplorer` application, which is shown in listing 6.6, is a lot more straightforward and pure Android focused than our other HTTP related classes up to this point. This is made possible by the helper class we have mentioned previously, which hides some of the complexity (we will examine the helper class itself after we look at this first class that uses it).

Listing 6.6 An Activity using Apache HttpClient via a custom `HttpRequestHelper`.

```

public class ApacheHTTPViaHelper extends Activity {

    private static final String CLASSTAG = ApacheHTTPViaHelper.class.getSimpleName();
    private static final String URL1 = "http://www.comedycentral.com/rss/jokes/index.jhtml";
    private static final String URL2 = "http://feeds.theonion.com/theonion/daily";

    private Spinner urlChooser;
    private Button button;
    private TextView output;
    private ProgressDialog progressDialog;

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

private final Handler handler = new Handler() {      #1
    @Override
    public void handleMessage(final Message msg) {
        progressDialog.dismiss();
        String bundleResult = msg.getData().getString("RESPONSE");
        ApacheHTTPViaHelper.this.output.setText(bundleResult);    #2
    }
};

@Override
public void onCreate(final Bundle icle) {
    super.onCreate(icle);
    this setContentView(R.layout.apache_http_simple);

    this.urlChooser = (Spinner) this.findViewById(R.id.apache_url);
    ArrayAdapter<String> urls = new ArrayAdapter<String>(this,
        android.R.layout.simple_spinner_item, new String[] {URL1, URL2 });
    urls.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    this.urlChooser.setAdapter(urls);

    this.button = (Button) this.findViewById(R.id.apachego_button);
    this.output = (TextView) this.findViewById(R.id.apache_output);

    this.button.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            ApacheHTTPViaHelper.this.output.setText("");
            ApacheHTTPViaHelper.this.performRequest(
                ApacheHTTPViaHelper.this.urlChooser.getSelectedItem().toString());    #3
        }
    });
};

. . . onPause omitted for brevity

private void performRequest(final String url) {

    final ResponseHandler<String> responseHandler =
        HTTPRequestHelper.getResponseHandlerInstance(this.handler);    #4
    this.progressDialog = ProgressDialog.show(this, "working . . .",
        "performing HTTP request");

    new Thread() {
        @Override
        public void run() {
            HTTPRequestHelper helper = new HTTPRequestHelper(responseHandler);    #5
            helper.performGet(url, null, null, null);    #6
        }
    }.start();
}
}

```

1. Create a Handler
2. Update the UI from the Handler
3. Call the local performRequest method with specified URL
4. Get a ResponseHandler from a static RequestHelper method
5. Instantiate RequestHelper with a ResponseHandler
6. Perform an HTTP get via the helper

Up first in this class we create another `Handler` **#1**, and from within it we simply update a `TextView` based on data in the `Message` **#2**. Further in the code, in the `onCreate` method, we call a local `performRequest` method when the “go” button is clicked, and we pass a selected `String` representing a URL **#3**.

Inside the `performRequest` method we use a static convenience method to return an `HttpClient` `ResponseHandler`, passing in our `Android Handler`, which it will use **#4**. We will see the helper class next, to get a look at exactly how this works, but the important part for now is that the `ResponseHandler` is created for us by the static method. With the `ResponseHandler` instance taken care of, we then instantiate an `HttpRequestHelper` instance **#5**, and use it to make a simple HTTP GET call (passing in only the `String` URL). Similarly to our previous example, when the request completes, the `ResponseHandler` will fire the `onResponse` method, and therein our `Handler` will be sent a `Message` completing the process.

The example Activity in listing 6.6 is therefore fairly clean and simple, and it's asynchronous and doesn't block the UI thread. The heavy lifting is really taken care of by HttpClient itself, and by the setup our custom HttpRequestHelper makes possible. The first part of the all important HttpRequestHandler, which we will explore in three sections, is seen in listing 6.7.

Listing 6.7 The first part of the HttpRequestHelper class showing convenience methods and setup of data and parameters in preparation for an HTTP request.

```
public class HTTPRequestHelper {

    private static final String CLASSTAG = HTTPRequestHelper.class.getSimpleName();

    private static final int POST_TYPE = 1;
    private static final int GET_TYPE = 2;
    private static final String CONTENT_TYPE = "Content-Type";
    public static final String MIME_FORM_ENCODED = "application/x-www-form-urlencoded";
    public static final String MIME_TEXT_PLAIN = "text/plain";

    private final ResponseHandler<String> responseHandler;

    public HTTPRequestHelper(final ResponseHandler<String> responseHandler) {    #1
        this.responseHandler = responseHandler;
    }

    public void performGet(final String url, final String user, final String pass,
        final Map<String, String> additionalHeaders) {    #2
        this.performRequest(null, url, user, pass,
            additionalHeaders, null, HTTPRequestHelper.GET_TYPE);
    }

    public void performPost(final String contentType, final String url, final String user, final String
pass,
        final Map<String, String> additionalHeaders, final Map<String, String> params) {    #3
        this.performRequest(contentType, url, user, pass,
            additionalHeaders, params, HTTPRequestHelper.POST_TYPE);
    }

    public void performPost(final String url, final String user, final String pass,
        final Map<String, String> additionalHeaders, final Map<String, String> params) {    #3
        this.performRequest(HTTPRequestHelper.MIME_FORM_ENCODED, url, user, pass,
            additionalHeaders, params, HTTPRequestHelper.POST_TYPE);
    }

    private void performRequest(final String contentType, final String url, final String user,
        final String pass, final Map<String, String> headers, final Map<String, String> params,
        final int requestType) {    #4

        DefaultHttpClient client = new DefaultHttpClient();    #5

        BasicHttpResponse errorResponse = new BasicHttpResponse(
            new ProtocolVersion("HTTP_ERROR", 1, 1), 500, "ERROR");    #6

        if ((user != null) && (pass != null)) {
            client.getCredentialsProvider().setCredentials(AuthScope.ANY,
                new UsernamePasswordCredentials(user, pass));    #7
        }

        final Map<String, String> sendHeaders = new HashMap<String, String>();
        if ((headers != null) && (headers.size() > 0)) {
            sendHeaders.putAll(headers);
        }
        if (requestType == HTTPRequestHelper.POST_TYPE) {
            sendHeaders.put(HTTPRequestHelper.CONTENT_TYPE, contentType);
        }
        if (sendHeaders.size() > 0) {
            client.addRequestInterceptor(new HttpRequestInterceptor() {    #8
                public void process(final HttpRequest request,
                    final HttpContext context) throws HttpException,
                    IOException {
                    for (String key : sendHeaders.keySet()) {
                        if (!request.containsHeader(key)) {
                            request.addHeader(key, sendHeaders.get(key));    #9
                        }
                    }
                }
            });
        }
    }
}
```

1. Constructing a helper requires a `ResponseHandler`
2. Providing a simple GET convenience method
3. Providing a few types of simple POST convenience methods
4. Private method to handle all the combinations internally
5. Instantiate a `DefaultHttpClient`
6. Create an error response, for possible future use
7. Add credentials if user and pass are present
8. Use an `Interceptor` to add request headers

The first thing of note in the `HttpRequestHelper` class is that a `ResponseHandler` is required to be passed in as part of the constructor **#1**. This `ResponseHandler` will be used when the `HttpClient` request is ultimately invoked. After the constructor, we see a public HTTP GET related method **#2**, and several different public HTTP POST related methods **#3**. Each of these methods is a wrapper around the private `performRequest` method that can handle all the various HTTP options **#4**. The `performRequest` method supports a content-type header value, URL, username, password, a `Map` of additional headers, a similar `Map` of request parameters, and request method type.

Inside the `performRequest` method a `DefaultHttpClient` is instantiated **#5**. Then, before further details are handled a special `BasicHttpResponse`, that we may use to return an error response, is also included. Remember, callers use this class asynchronously and expect a response, good or bad, so we need to ensure that we always have a response of some kind, and that the `onResponse` method of the passed in `ResponseHandler` is always invoked **#6**.

Next, we check if the `user` and `pass` method parameters are present, and if so we set the request `Credentials` with a `UsernamePasswordCredentials` type (`HttpClient` supports several types of credentials, see the JavaDocs for details). At the same time we set the credentials we also set an `AuthScope`. The scope represents what server, port, authentication realm, and authentication scheme the credentials supplied are applicable for. You can set these as fine or coarse grained as you want, we are using the default `ANY` scope that matches anything. What we notably have **not** set in all of this is the specific authentication scheme to use. `HttpClient` supports various schemes, including basic authentication, digest authentication, and a Windows specific NTLM scheme. Basic authentication, meaning simple username/password challenge from the server, is the default. (Also, if you need to you can use a “preemptive” form login for form based authentication – just submit the form you need and get the token or session ID, etc.)

After the security is out of the way we use an `HttpRequestInterceptor` to add HTTP headers **#8**. Headers are name/value pairs, so this is pretty easy. Once we have all of these properties that apply regardless of our request method type out of the way, we then add further settings that are specific to the method. Listing 6.8, the second part of our helper class, shows the POST and GET specific settings.

Listing 6.8 The second part of the `HttpRequestHelper` class, showing the separate POST and GET method paths.

```

if (requestType == HttpRequestHelper.POST_TYPE) {           #1
    HttpPost method = new HttpPost(url);                    #2

    List<NameValuePair> nvps = null;
    if ((params != null) && (params.size() > 0)) {
        nvps = new ArrayList<NameValuePair>();
        for (String key : params.keySet()) {
            nvps.add(new BasicNameValuePair(key, params.get(key))); #3
        }
    }
    if (nvps != null) {
        try {
            method.setEntity(new UrlEncodedFormEntity(nvps, HTTP.UTF_8));
        } catch (UnsupportedEncodingException e) {
            Log.e(Constants.LOGTAG, " " + HttpRequestHelper.CLASSTAG, e);
        }
    }
}

try {
    client.execute(method, this.responseHandler);           #4
}

```

```

    } catch (Exception e) {
        errorResponse.setReasonPhrase(e.getMessage());    #5
        try {
            this.responseHandler.handleResponse(errorResponse);
        } catch (Exception ex) {
            Log.e(Constants.LOGTAG, " " + HTTPRequestHelper.CLASSTAG, ex);
        }
    }
} else if (requestType == HTTPRequestHelper.GET_TYPE) {    #6
    HttpGet method = new HttpGet(url);

    try {
        client.execute(method, this.responseHandler);
    } catch (Exception e) {
        Log.e(Constants.LOGTAG, " " + HTTPRequestHelper.CLASSTAG, e);
        errorResponse.setReasonPhrase(e.getMessage());
        try {
            this.responseHandler.handleResponse(errorResponse);
        } catch (Exception ex) {
            Log.e(Constants.LOGTAG, " " + HTTPRequestHelper.CLASSTAG, ex);
        }
    }
}
}
}
}

```

1. Handle POST method requests
2. Create an `HttpPost` method object
3. Add name/value pair request parameters
4. Call the client execute method with the `ResponseHandler`
5. Perform some special error handling
6. Handle GET method requests

When the specified request is a POST type **#1**, we create an `HttpPost` object to deal with it **#2**. Then we add POST request parameters, which are another set of name/value pairs and are built with the `BasicNameValuePair` object **#3**. After the parameters we are then ready to actually perform the request, which we do with the `execute` method using the method object, and the `ResponseHandler` from the earlier constructor **#4**. Again, as we have seen, this `ResponseHandler` is what the Apache `HttpClient` framework will use to make the callback when the request completes. In some circumstances, such as the error handling setup we have **#5**, we may call the `handleResponse` method (which internally fires the `onResponse` method to listeners) directly. Here we do that to set error message data into a fake response, should an error occur.

When the POST method operations are out of the way we then next see the GET method details **#6**. We handle the GET method very similarly, but we don't set parameters (with GET requests we expect parameters encoded in the URL itself). Right now our class only supports POST and GET (which cover 98% of the requests we generally need), but it certainly could be expanded upon to support other HTTP method types.

The final part of the request helper class, shown in listing 6.9, takes us back to the first example we saw that used the helper, as it outlines exactly what the convenience `getResponseHandlerInstance` method returns.

Listing 6.9 The final part of the `HttpRequestHelper` class, showing the static `getResponseHandlerInstance` method.

```

public static ResponseHandler<String> getResponseHandlerInstance(final Handler handler) {    #1
    final ResponseHandler<String> responseHandler = new ResponseHandler<String>() {
        public String handleResponse(final HttpResponse response) {
            Message message = new Message();
            Bundle bundle = new Bundle();
            StatusLine status = response.getStatusLine();
            HttpEntity entity = response.getEntity();
            String result = null;
            if (entity != null) {
                try {
                    result = StringUtils.inputStreamToString(entity.getContent());    #2
                    bundle.putString("RESPONSE", result);    #3
                    message.setData(bundle);    #3
                    handler.sendMessage(message);    #3
                } catch (Exception e) {
                    Log.e(Constants.LOGTAG, " " + HTTPRequestHelper.CLASSTAG, e);
                }
            }
            return result;
        }
    };
    return responseHandler;
}

```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        } catch (IOException e) {
            Log.e(Constants.LOGTAG, " " + HTTPRequestHelper.CLASSTAG, e);
            bundle.putString("RESPONSE", "Error - " + e.getMessage());
            message.setData(bundle);
            handler.sendMessage(message);
        }
    } else {
        bundle.putString("RESPONSE", "Error - " + response.getStatusLine().getReasonPhrase());
        message.setData(bundle);
        handler.sendMessage(message);    #4
    }
    return result;
}
};
return responseHandler;
}
}

```

1. Require a Handler parameter
2. Get the response entity content as a String
3. Put the result value into a Bundle keyed as RESPONSE
4. Set the bundle as the data into the Message
5. Send the message via the Handler

As we discuss the `getResponseHandlerInstance` method of our helper we should note that though we find it helpful, it's really entirely optional. You can still make use of the helper class without using this method. To do so, you simply need to construct your own `ResponseHandler` and pass it in to the helper constructor – which is a perfectly plausible case. The `getResponseHandlerInstance` method simply hooks in a `Handler` via a parameter **#1**, and then parses the response as a `String` **#2**. The response `String` is sent back to the caller using the `Handler Bundle` and `Message` pattern we have seen used time and time again to pass messages between threads in our Android screens.

With the gory `HttpRequestHelper` details out of the way, and having also already seen some basic usage, we will next turn to some more involved uses of this class in the context of web service calls.

6.5 Web Services

Web services means many different things depending on the audience. To some it's a nebulous marketing term that is never pinned down, to others it's a very rigid and specific set of protocols and standards. We are going to tackle it as a general concept, but not leave it entirely undefined, and not define it to death either.

Web services is a means of exposing an API over a technology neutral network endpoint. It's a means to call a remote method or operation, not tied to a specific platform or vendor, and get a result. By this definition Plain Old XML over the network (POX) is included, and so is Representational State Transfer (REST), and, so is the Simple Object Access Protocol (SOAP) – and really so is any other method of exposing operations and data on the wire in a neutral manner.

POX, REST, and SOAP are by far the most common web services around, so they are where we will focus in this section. Each of these provides a general guideline for accessing data, and exposing operations, each in a more rigorous manner than the previous, respectively. POX is basically just exposing chunks of XML over the wire, usually over HTTP. REST is a bit more detailed in that it uses the concept of “resources” to define data, and then manipulates them with different HTTP methods using a URL style approach (much like the Android intent system in general, which we have already explored in previous chapters). And, SOAP is the most formal of them all, imposing strict rules about types of data, transport mechanisms, and security.

All of these approaches have advantages and disadvantages, and these differences are amplified on a mobile platform like Android. Though we can't possibly cover all the details here, we will touch on the differences as we discuss each of these approaches. Here we will examine the use of a POX approach to return recent posts from the `del.icio.us` API, and we will then look at using REST with the Google Data AtomPub API. Up first is what is probably the most ubiquitous type of

web service in use on the Internet today, and therefore one you will come across again and again when connecting Android applications – POX.

6.4.1 POX - Putting it together with HTTP and XML

To work with “plain old XML” over HTTP we are going to make some network calls to the popular del.icio.us online “social bookmarking” site. We will specify a username and password to login to an HTTPS resource, and return a list of “recent posts,” or bookmarks. This service returns raw XML data, and we will then parse it into a JavaBean style class and display it as seen in figure 6.4.

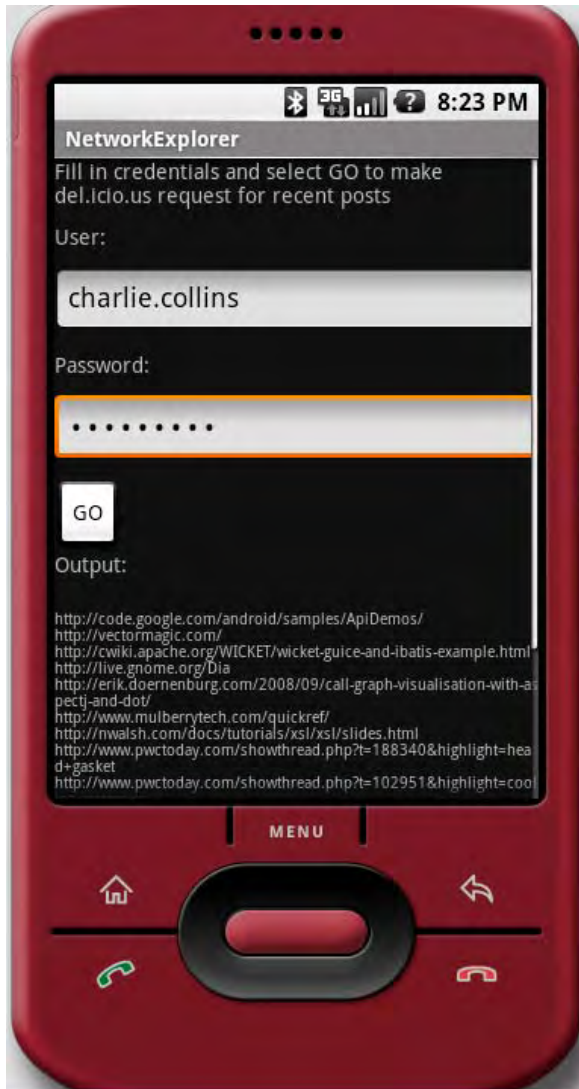


Figure 6.4 The del.icio.us recent posts screen from the NetworkExplorer application.

Listing 6.10 shows the del.icio.us login and HTTPS POST Activity code from our NetworkExplorer application.

Listing 6.10 Using the Del.icio.us HTTPS POX API with authentication from an Android Activity.

```
public class DeliciousRecentPosts extends Activity {  
  
    private static final String CLASSTAG = DeliciousRecentPosts.class.getSimpleName();  
    private static final String URL_GET_POSTS_RECENT = "https://api.del.icio.us/v1/posts/recent";    #1
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

. . . member var declarations for user, pass, output, and button (Views) omitted for brevity,

private final Handler handler = new Handler() {                                     #2
    @Override
    public void handleMessage(final Message msg) {
        progressDialog.dismiss();
        String bundleResult = msg.getData().getString("RESPONSE");
        DeliciousRecentPosts.this.output.setText(
            DeliciousRecentPosts.this.parseXMLResult(bundleResult));
    }
};

@Override
public void onCreate(final Bundle icle) {
    super.onCreate(icle);
    this setContentView(R.layout.delicious_posts);

    . . . inflate views omitted for brevity

    this.button.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            DeliciousRecentPosts.this.output.setText("");
            DeliciousRecentPosts.this.performRequest(DeliciousRecentPosts.this.user.getText()
                .toString(), DeliciousRecentPosts.this.pass.getText().toString());
        }
    });
};

. . . onPause omitted for brevity

private void performRequest(final String user, final String pass) {
    this.progressBar = ProgressDialog.show(this,
        "working . . .", "performing HTTP post to del.icio.us");

    final ResponseHandler<String> responseHandler =
        HTTPRequestHelper.getResponseHandlerInstance(this.handler);

    new Thread() {
        @Override
        public void run() {
            HTTPRequestHelper helper = new HTTPRequestHelper(responseHandler);
            helper.performPost(URL_GET_POSTS_RECENT, user, pass, null, null); #3
        }
    }.start();
}

private String parseXMLResult(String xmlString) {                                #4
    StringBuilder result = new StringBuilder();
    try {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        SAXParser sp = spf.newSAXParser();
        XMLReader xr = sp.getXMLReader();
        DeliciousHandler handler = new DeliciousHandler();
        xr.setContentHandler(handler);
        xr.parse(new InputSource(new StringReader(xmlString)));

        List<DeliciousPost> posts = handler.getPosts();
        for (DeliciousPost p : posts) {
            result.append("\n" + p.getHref());
        }
    } catch (Exception e) {
        Log.e(Constants.LOGTAG, " " + DeliciousRecentPosts.CLASSTAG + " ERROR - " + e);
    }
    return result.toString();
}

```

1. Include the del,icio.us URL for recent posts
2. Provide a handler to update the UI
3. Call the local performPost method with a username and password
4. Parse the XML String result with SAX

To utilize a POX service we need to know a little bit about it, beginning with the URL endpoint **#1**. To call the del.icio.us service we will again use a `Handler` as a “handle” to update the UI **#2**, and we will use the `HttpRequestHelper` we previously built and walked through in the last section. In this example we again have many fewer lines of code than if we did not use the helper (lines of code we would likely be repeating in different `Activity` classes). With the helper instantiated we

call the `performRequest` method with a username and password **#3**. This method, via the helper, will login to `del.icio.us`, and return an XML chunk representing the most recently bookmarked items. To turn the raw XML into useful types we then also include a `parseXMLResult` method **#4**. Parsing XML is a subject in it's own right, and therefore we will cover it in more detail in chapter 13, but the short take away with this method is that we walk the XML structure with a parser and return our own `DeliciousPost` data beans for each record. That's it, that's using POX to read data over HTTPS.

Building on the addition of XML to HTTP, above and beyond POX, is the REST “architectural principle,” which we will explore next.

6.5.2 REST

While we look at REST, we will also try to pull in another useful concept in terms of Android development; working with the various Google Data APIs (GDATA: <http://code.google.com/apis/gdata/>). We have already used the GDATA APIs for our RestaurantFinder review information in chapter 3, but there we didn't authenticate, and we didn't get into the details of networking or REST. Here we will uncover the details, as we perform two distinct tasks: authenticate and retrieve a Google ClientLogin token, and then retrieve the Google Contacts data for a specified user. Keep in mind that as we work with the GDATA APIs in any capacity, we will be using a REST style API.

The main concepts with REST are that you specify resources in a URI form, and you use different protocol methods to perform different actions. The Atom Publishing Protocol (AtomPub) defines a REST style protocol – and the GDATA APIs are an implementation of AtomPub and some extra Google add ons. As noted, the entire `Intent` approach of the Android platform is a lot like REST. A URI such as `content://contacts/1` is in the REST style. It includes a path that identifies the type of data, and a particular resource (contact number 1).

That URI does not say what to do with contact 1, however. In REST terms that is where the method of the protocol comes into the picture. For HTTP purposes REST utilizes the various methods to perform different tasks: POST (create, update – or in special cases delete), GET (read), PUT (create, replace), and DELETE (delete). True HTTP REST implementations use all the HTTP method types, and resources, to construct APIs.

In the real world though, you will find very few “true” REST implementations. It is much more common to see a REST “style” API, that doesn't typically use HTTP DELETE method (many servers, proxies, and so on have trouble with DELETE), and overloads the more common GET and POST methods with different URLs for different tasks (by encoding a bit about what is to be done in the URL, or as a header or parameter, rather than relying strictly on the method). In fact, though many people refer to the GDATA APIs as REST, they are technically only REST-like, but not true REST. That's not necessarily a bad thing though, the idea is ease of use of the API, rather than pattern purity. All in all REST is a very popular architecture or style, because it's easy yet powerful.

Listing 6.11 is a quick example that focuses on the network aspects of authentication with GDATA to obtain a ClientLogin token, and then using that token with a subsequent REST style request to obtain Contacts data by including an email address as a “resource.”

Listing 6.11 Using the Google Contacts AtomPub REST Style API with authentication from an Android Activity.

```
public class GoogleClientLogin extends Activity {

    private static final String CLASSTAG = GoogleClientLogin.class.getSimpleName();
    private static final String URL_GET_GTOKEN = "https://www.google.com/accounts/ClientLogin";
    private static final String URL_GET_CONTACTS_PREFIX = "http://www.google.com/m8/feeds/contacts/";
    private static final String URL_GET_CONTACTS_SUFFIX = "/full";
    private static final String GTOKEN_AUTH_HEADER_NAME = "Authorization";
    private static final String GTOKEN_AUTH_HEADER_VALUE_PREFIX = "GoogleLogin auth=";
    private static final String PARAM_ACCOUNT_TYPE = "accountType";
    private static final String PARAM_ACCOUNT_TYPE_VALUE = "HOSTED_OR_GOOGLE";
    private static final String PARAM_EMAIL = "Email";
    private static final String PARAM_PASSWD = "Passwd";
    private static final String PARAM_SERVICE = "service";

    Please post comments or corrections to the Author Online forum at
    http://www.manning-sandbox.com/forum.jspa?forumID=411
    Licensed to Thow Way Chiam <ken.ctw@gmail.com>
```

```

private static final String PARAM_SERVICE_VALUE = "cp"; // google contacts
private static final String PARAM_SOURCE = "source";
private static final String PARAM_SOURCE_VALUE = "manning-unlockingAndrid-1.0";

private String tokenValue;

private EditText emailAddress;
private EditText password;
private Button getContacts;
private Button getToken;
private Button clearToken;
private TextView tokenText;
private TextView output;
private ProgressDialog progressDialog;

private final Handler tokenHandler = new Handler() {          #1
    @Override
    public void handleMessage(final Message msg) {
        GoogleClientLogin.this.progressBar.dismiss();
        String bundleResult = msg.getData().getString("RESPONSE");
        String authToken = bundleResult;
        authToken = authToken.substring(authToken.indexOf("Auth=") + 5, authToken.length()).trim();
        GoogleClientLogin.this.tokenValue = authToken;          #2
        GoogleClientLogin.this.tokenText.setText(authToken);
    }
};

private final Handler contactsHandler = new Handler() {        #3
    @Override
    public void handleMessage(final Message msg) {
        GoogleClientLogin.this.progressBar.dismiss();
        String bundleResult = msg.getData().getString("RESPONSE");
        GoogleClientLogin.this.output.setText(bundleResult);
    }
};

. . . onCreate and onPause omitted for brevity

private void getToken(final String email, final String pass) {    #4
    final ResponseHandler<String> responseHandler =
        HTTPRequestHelper.getResponseHandlerInstance(this.tokenHandler);

    this.progressBar = ProgressDialog.show(this, "working . . .", "getting Google ClientLogin
token");

    new Thread() {
        @Override
        public void run() {
            HashMap<String, String> params = new HashMap<String, String>();
            params.put(GoogleClientLogin.PARAM_ACCOUNT_TYPE,
                GoogleClientLogin.PARAM_ACCOUNT_TYPE_VALUE);      #5
            params.put(GoogleClientLogin.PARAM_EMAIL, email);      #5
            params.put(GoogleClientLogin.PARAM_PASSWD, pass);      #5
            params.put(GoogleClientLogin.PARAM_SERVICE, GoogleClientLogin.PARAM_SERVICE_VALUE); #5
            params.put(GoogleClientLogin.PARAM_SOURCE, GoogleClientLogin.PARAM_SOURCE_VALUE); #5

            HTTPRequestHelper helper = new HTTPRequestHelper(responseHandler);
            helper.performPost(HTTPRequestHelper.MIME_FORM_ENCODED,
                GoogleClientLogin.URL_GET_GTOKEN, null, null, null, params); #6
        }
    }.start();
}

private void getContacts(final String email, final String token) { #7
    final ResponseHandler<String> responseHandler =
        HTTPRequestHelper.getResponseHandlerInstance(this.contactsHandler);

    this.progressBar = ProgressDialog.show(this, "working . . .", "getting Google Contacts");

    new Thread() {
        @Override
        public void run() {
            HashMap<String, String> headers = new HashMap<String, String>();
            headers.put(GoogleClientLogin.GTOKEN_AUTH_HEADER_NAME,
                GoogleClientLogin.GTOKEN_AUTH_HEADER_VALUE_PREFIX + token); #8

            String encEmail = email;
            try {
                encEmail = URLEncoder.encode(encEmail, "UTF-8"); #9
            } catch (UnsupportedEncodingException e) {
                Log.e("GoogleClientLogin", "Please post comments or corrections to the Author Online forum at
http://www.manning-sandbox.com/forum.jspa?forumID=411
Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

```

    }
    String url = GoogleClientLogin.URL_GET_CONTACTS_PREFIX + encEmail +
        GoogleClientLogin.URL_GET_CONTACTS_SUFFIX;

    HTTPRequestHelper helper = new HTTPRequestHelper(responseHandler);
    helper.performGet(url, null, null, headers);    #10
    }
    }.start();
}
}
}

```

1. Create a Handler for the token request
2. Set the tokenValue
3. Create another Handler to be used for the contacts request
4. Implement the getToken method
5. Include all the necessary parameters for obtaining a ClientLogin token
6. Perform a POST request to get the ClientLogin token, via the helper
7. Implement the getContacts method
8. Add the token as a header for subsequent API requests
9. Encode the email address part of the URL
10. Make a GET request for Contacts data

After a host of constants that represent various `String` values that we will use with the GDATA services, we then see that we have several `Handler` instances in this class, beginning with a `tokenHandler` #1. This handler updates a UI `TextView` when it receives a message, like the previous similar examples we have seen, and also updates a non UI member `tokenValue` variable that other portions of our code will use #2. The next `Handler` we have is the `contactsHandler` that will be used to update the UI after the contacts request #3.

Beyond the handlers we have the `getToken` method #4. This method includes all the required parameters for obtaining a `ClientLogin` token from the GDATA servers (<http://code.google.com/apis/gdata/auth.html>) #5. After the setup to obtain the token, we make a POST request via the request helper #6.

Once the token details are taken care of we then have the `getContacts` method #7. This method uses the token obtained via the previous method as a header #8. After you have the token you can cache it and use it with all subsequent requests (you don't need to re-obtain the token every time). Next we see that we encode the email address portion of the Contacts API URL #9, and then we make a GET request for the data – again using the `HttpRequestHelper` #10.

GDATA ClientLogin and CAPTCHA

While we have included a working `ClientLogin` example here, we have also skipped over an important part – CAPTCHA. Google may optionally require a CAPTCHA with the `ClientLogin` approach. To fully support `ClientLogin` you need to handle that response and display the CAPTCHA to the user, and then resend a token request with the user's entered CAPTCHA value. For details see the GDATA documentation.

Now that we have seen some REST style newtorking, the last thing we need to discuss with regard to HTTP and Android is SOAP. This topic comes up frequently in discussions of networking mobile devices, but sometimes the forest gets in the way of the trees in terms of framing the real question.

6.5.3 To SOAP or not to SOAP, that is the question

SOAP is a powerful protocol that has many uses. Nevertheless, we would be remiss if we didn't at least mention that while it's possible, it's not generally recommended on a small embedded device like a smart phone – regardless of the platform. The question within the limited resources

environment Android inhabits is really more one of “should” it be done, rather than “can” it be done.

Surely some experienced developers, who may have been using SOAP for years on other devices, may be snarling at this sentiment right now. To those of you in that camp we would ask you to bear with us as we try to explain for a moment. SOAP is great, this is not a dig against SOAP itself. Yet, the things that make SOAP great are its support for strong types (via XML Schema), its support for transactions, its security and encryption, its support for message orchestration and choreography, and all the related WS-* standards. These things are invaluable in many server oriented computing environments, whether or not they involve the “enterprise.” And these things are mostly worthless, and add a great deal of overhead, on a small embedded device. In fact, in many situations where people do use SOAP on embedded devices they often don't bother with the advanced features – and they just use plain XML with the overhead of an envelope at the end of the day anyway. On an embedded device it is often simpler, and more performant, to use a REST or POX style architecture and avoid the overhead of SOAP.

There are of course some situations where it makes sense to investigate using SOAP directly with Android. In the case where you need to talk to existing SOAP services that you have no control over, SOAP might make sense. Also, if you already have J2ME clients for existing SOAP services, you may be able to port those in a limited set of cases. Yet, either of these approaches really only makes it easier on you, the developer, and has either no effect, or a negative one in terms of performance on the user. Even when you are working with existing SOAP services, remember, you can often write a POX/REST style proxy for SOAP services on the server side and call **that** from Android – rather than using SOAP directly from Android.

If you feel like SOAP is still the right choice, then you can use one of several ports of the kSOAP toolkit (<http://ksoap2.sourceforge.net/>), which is specially designed exactly for SOAP on an embedded Java device. Keep in mind though, even the kSOAP documentation states “SOAP introduces some significant overhead for web services that may be problematic for mobile devices. If you have full control over the client and the server, a REST based architecture may be more adequate.” Additionally you may be able to write your own parser for simple SOAP services that don't use fancy SOAP features, and rather just use a POX approach that includes the SOAP XML portions you require (you can always roll your own, even with SOAP).

All in all in our minds the answer to the questions is **not** to SOAP on Android, even though you **can**. Our discussion of SOAP, even though we don't advocate it, rounds out our more general “web services” discussion, and that itself wraps up our networking coverage.

6.6 Summary

In this chapter we started with a brief lesson on the background of basic networking concepts, from nodes and addresses, to layers and protocols. Then, with that general background in place we next covered details concerning obtaining network status information, and finally several different ways work with the IP networking capabilities of the platform.

In terms of networking we looked at using basic sockets and the `java.net` package. Then, we also examined the included Apache `HttpClient` API. HTTP is one of the most common, and arguably most important, networking resources available to the Android platform. Using `HttpClient` we covered a lot of territory in terms of different request types, parameters, headers, authentication, and more. Beyond basic HTTP we also extended into the concepts of POX and REST, and discussed a bit of SOAP – all of which use HTTP as the transport mechanism.

Now that we have covered a good deal of the networking possibilities, and hopefully given you at least a glint of an idea of what you can do with server side APIs and integration with Android, we are next going to turn to another very important part of the Android world – telephony.

7

Telephony

In this chapter,

- Telephony background
- Making and receiving phone calls
- Capturing call related events
- Obtaining phone and service information
- Using SMS

With an Android device you can surf the web, store and retrieve data locally, access networks, access location information, use many types of applications, and, get this, you can actually make phone calls too.

After all is said and done, one of the most fundamental components of the platform is the mobile phone. Dialing out, receiving calls, sending and receiving text messages, and other related telephony services are all available. The added bonus with Android is that all of these items are accessible to developers through simple to use APIs, and built-in applications that make use of intents and services. You can use the telephony support Android provides quite easily, and you can also combine it and embed it in your own applications (as we have seen in previous examples).

In this chapter the first thing we are going to do is examine a bit of telephony background, and cover some of the terms involved with a mobile device. From there we will also cover the basic Android telephony packages which will take us through handling calls using built-in `Intent` actions, and examining the `TelephonyManager` and `PhoneStateListener` classes. The `Intent` actions are what you will use on a day to day basis to initiate phone calls in your applications. `TelephonyManager` is, on the other hand, not related to making calls, but rather used to retrieve all kinds of telephony related data – such as the state of the voice network, the device's own phone number, and Subscriber Identity Module (SIM) card details. Through `TelephonyManager` is also how you attach a `PhoneStateListener`, which can alert you when call or phone network states change.

Once we have the background and basic telephony APIs in hand we will then move on to working with another very common mobile phone feature – sending and receiving Short Message Service (SMS) messages. Android provides intents and built-in

applications for handling SMS messages, and also provides APIs that allow you to send SMS messages, and be notified when SMS messages are received.

As we explore the telephony packages and the SMS support we will also touch on a few of the emulator features that allow you to send in test calls and or messages to exercise your applications.

We are once again going to use a sample application to carry us through the concepts related to the material in this chapter. We will be building a TelephonyExplorer application here to demonstrate dialing the phone, obtaining phone and service state information, adding listeners to the phone state, and working with SMS. Our TelephonyExplorer application will have several basic screens as seen in figure 7.1.



Figure 7.1 TelephonyExplorer main screen, showing all the related activities the sample application performs.

TelephonyExplorer, as you can see from the screen shot, is not pretty, nor is it very practical outside of learning the concepts and API details involved. This application is specifically focused on touching the telephony related APIs while remaining simple and uncluttered.

Before we begin to build TelephonyExplorer, the first thing we need to do is to take a quick step back and clarify what exactly telephony is, and what some of the related terms are.

7.1 Telephony background and terms

Prior to getting into the code and APIs in this chapter, first we want to provide a short overview of telephony itself; what it is, and what the related terms are. This information is very basic, and may not be new to experienced mobile developers (who should feel free to skip to the next section), but it's still important to clarify terms, and set out some background for those that are new to these concepts.

First, telephony is a general term that refers to the details surrounding electronic voice communications over telephone networks. Our scope is of course the mobile telephone network that Android devices will participate in, specifically the Global System for Mobile Communications (GSM) network.

TELEPHONE

The term “telephone” means speech over a distance. The Greek roots are tele, which means “distant,” and phone, which means “speech.”

GSM is a cellular telephone network. Devices communicate over radio waves and specified frequencies using the cell towers that are common across the landscape. This means the GSM standard has to define a few important things, like identities for devices and “cells,” along with all of the rules for making communications possible.

We won't delve into the underlying details of GSM, but it's important to know that it's the standard that the Android stack currently uses to support voice calls – and it's the most widely used standard in the world across carriers and devices Android or otherwise. All GSM devices use a “SIM” card, or Subscriber Identity Module, to store all the important network and user settings.

A SIM card is a small, removable, and secure smart card. Every device that operates on a GSM network has a few specific unique identifiers, which are stored on the SIM card:

- **Integrated Circuit Card ID (ICCID)** – Unique number that identifies a SIM card (also known as a SIM Serial Number or SSN)
- **International Mobile Equipment Identity (IMEI)** – Unique number to identify a physical device (usually printed underneath the battery)
- **International Mobile Subscriber Identity (IMSI)** – Unique number to identify a subscriber (and the network that subscriber is on)
- **Location Area Identity (LAI)** – Unique number that identifies the region the device is in within a provider network.
- **Authentication Key (Ki)** – A 128 bit key used to authenticate a SIM card on this provider network.

These numbers are important for the obvious reasons that they are used to validate and authenticate a SIM card itself, the device it is in, and the subscriber on the network (and across networks if need be).

Along with storing unique identifiers and authentication keys SIM cards also often are capable of storing user contacts and SMS messages. This is convenient for users because they can move their SIM card to a new device and “carry along” contact and message data easily. At present there are no public APIs for interacting with the SIM card on an Android device directly, though this may become possible in the future (right now the platform handles the SIM interaction, and developers can get read only access via the telephony APIs).

The basic background for working with the Android telephony packages really is that short and simple. You need to know that you are working with a GSM network, and then you need to be aware that you may come across terms like IMSI and IMEI which are stored on the SIM. Getting at this information, and more, is done with the `TelephonyManager` class.

7.2 Accessing telephony information

Android provides a very informative manager class that supplies information about many telephony related details on the device. Using this class, `TelephonyManager`, you can access many of the GSM/SIM properties we have already discussed, and you can additionally obtain phone network state information and updates.

Attaching an event listener to the phone, in the form of a `PhoneStateListener`, which is done via the manager, is how you can make your applications aware of when phone service is and is not available, and when calls are started, in progress, or ending, and more.

Here we are going to begin several portions the `TelephonyExplorer` example application to look at both of these classes and concepts, starting with obtaining a `TelephonyManager` instance, and then using it query useful telephony information.

7.2.1 Retrieving telephony properties

The `android.telephony` package contains the `TelephonyManager` class, and has details on all of the information you can obtain using it. Here we are going to get and display a small subset of that information, to demonstrate the approach. The first `Activity`, beyond the main screen, our `TelephonyExplorer` application will have is a simple screen that shows some of the information we can obtain via `TelephonyManager`, as seen in figure 7.2.



Figure 7.2 Displaying device and phone network meta information obtained from the TelephonyManager class.

The TelephonyManager class is the information hub for telephony related data in Android. Listing 7.1 demonstrates how you obtain a reference to this class and use it to retrieve data (such as the data shown in figure 7.2).

Listing 7.1 Obtaining a reference to TelephonyManager, and making using it to retrieve information.

```
// . . . start of class omitted for brevity

final TelephonyManager telMgr =
    (TelephonyManager) this.getSystemService(Context.TELEPHONY_SERVICE);    #1

// . . . onCreate method and others omitted for brevity

public String getTelephonyOverview(final TelephonyManager telMgr) {        #2

    int callState = telMgr.getCallState();                                  #3
    String callStateString = "NA";
    switch (callState) {
    case TelephonyManager.CALL_STATE_IDLE:
        callStateString = "IDLE";
        break;
    case TelephonyManager.CALL_STATE_OFFHOOK:
        callStateString = "OFFHOOK";
        break;
    }
```

```

        case TelephonyManager.CALL_STATE_RINGING:
            callStateString = "RINGING";
            break;
    }

    GsmCellLocation cellLocation = (GsmCellLocation) telMgr.getCellLocation();
    String cellLocationString =
        cellLocation.getLac() + " " + cellLocation.getCid();

    String deviceId = telMgr.getDeviceId();
    String deviceSoftwareVersion = telMgr.getDeviceSoftwareVersion();

    String line1Number = telMgr.getLine1Number();

    String networkCountryIso = telMgr.getNetworkCountryIso();
    String networkOperator = telMgr.getNetworkOperator();
    String networkOperatorName = telMgr.getNetworkOperatorName();

    int phoneType = telMgr.getPhoneType();
    String phoneTypeString = "NA";
    switch (phoneType) {
        case TelephonyManager.PHONE_TYPE_GSM:
            phoneTypeString = "GSM";
            break;
        case TelephonyManager.PHONE_TYPE_NONE:
            phoneTypeString = "NONE";
            break;
    }

    String simCountryIso = telMgr.getSimCountryIso();
    String simOperator = telMgr.getSimOperator();
    String simOperatorName = telMgr.getSimOperatorName();
    String simSerialNumber = telMgr.getSimSerialNumber();
    String simSubscriberId = telMgr.getSubscriberId();
    int simState = telMgr.getSimState();
    String simStateString = "NA";
    switch (simState) {
        case TelephonyManager.SIM_STATE_ABSENT:
            simStateString = "ABSENT";
            break;
        case TelephonyManager.SIM_STATE_NETWORK_LOCKED:
            simStateString = "NETWORK_LOCKED";
            break;
        // . . . other SIM states omitted for brevity
    }

    StringBuilder sb = new StringBuilder();
    sb.append("telMgr - ");
    sb.append(" \ncallState = " + callStateString);

    // . . . remainder of appends omitted for brevity

    return sb.toString();
}

```

1. Get a TelephonyManager instance from the context
2. Implement a telephony information helper method
3. Obtain call state information
4. Get cell location information
5. Get device information
6. Get the phone number of the device
7. Obtain SIM related information

The Android `Context` is used, through the `getSystemService` method with a constant, to obtain an instance of the `TelephonyManager` class **#1**. Once you have a handle to the manager you can then use it as needed to obtain information. In this case we have

created a helper method to get data from the manager and return it as a `String` we later display on the screen **#2**.

The manager allows you to access phone state data such as whether or not a call is in progress **#3**, cell location information **#4**, the device ID and software version **#5**, the phone number registered to the current user/SIM **#6**, many other SIM details such as the subscriber ID (IMSI) **#7**. There are also additional properties that we are not using in this example (see the JavaDocs for complete details).

We need to note one more detail here not seen in the listing. In order for this class to work, the `READ_PHONE_STATE` permission has to be set in the manifest (without it security exceptions will be thrown when you try to read data from the manager). We have consolidated the phone related permissions into Table 7.1, in section 7.3.1.

This handle to the telephony related information, including metadata about the device, network, and SIM card, is one of the main purposes of the `TelephonyManager` class. The other main purpose of `TelephonyManager` is to allow you to attach a `PhoneStateListener`.

7.2.2 Obtaining phone state information

Obviously a phone has various states that it as a device can be in. The most basic phone states are: idle, in a call, or in the process of initiating a call. When building applications on a mobile device there are times when you need to know not only the current phone state, but also want to be alerted anytime the state changes.

In these cases you want to attach a listener to the phone and “subscribe” so that you can be notified of “published” changes. With Android this is done using a `PhoneStateListener`, which is attached to the phone through `TelephonyManager`. Listing 7.2 demonstrates a sample usage of both of these classes.

Listing 7.2 Attaching a PhoneStateListener via the TelephonyManager, and reacting to onCallStateChanged events.

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

        final TelephonyManager telMgr =
            (TelephonyManager) this.getSystemService(Context.TELEPHONY_SERVICE);           #1

        PhoneStateListener phoneStateListener = new PhoneStateListener() {
            #2
            public void onCallStateChanged(int state, String incomingNumber) {
                #3
                TelephonyManagerExample.this.telMgrOutput.setText(
                    TelephonyManagerExample.this.getTelephonyOverview(telMgr));
            }
        };
        #4
        telMgr.listen(phoneStateListener, PhoneStateListener.LISTEN_CALL_STATE);

        String telephonyOverview = this.getTelephonyOverview(telMgr);
        this.telMgrOutput.setText(telephonyOverview);
    }
```

1. Again obtain `TelephonyManager` from the context

2. Create a `PhoneStateListener`
3. Implement the `onCallStateChanged` method
4. Assign the listener to the manager

To start working with a `PhoneStateListener` you first need an instance to `TelephonyManager`, so you can later assign the listener **#1**. `PhoneStateListener` itself is an interface, so you need to create an implementation **#2**, including the `onCallStateChanged` required method, in order to use it **#3**. Once you have a `PhoneStateListener` instance (your own implementation that implements the interface), you then attach it by assigning it to the manager with the `listen` method **#4**.

In the example in listing 7.2 we are listening for any `PhoneStateListener.LISTEN_CALL_STATE` change in the phone state. This is a constant value from a list of available states that can be seen on the `PhoneStateListener` class. You can use a single value when assigning a listener with the `listen` method, as we have done here, or you can combine multiple values.

If a call state change does occur, we are simply resetting the details on the screen using the `getTelephonyOverview` method we saw for setting the initial status in listing 7.1. The action you take is defined in the `onCallStateChanged` method of your `PhoneStateListener`. You can filter further in this method too (apart from the types of events you are listening for), based on the passed in `int state`, if you need to.

To see the values in this example change while working with the emulator, you can use the SDK tools to send incoming calls or text messages, and change the state of the voice connection. The emulator includes a mock GSM modem that you can manipulate using the "gsm" command from the console. Figure 7.3 is an example session from the console that demonstrates this. For complete details see the emulator telephony documentation (<http://code.google.com/android/reference/emulator.html#telephony>).


```

ccollins@crotalus:/opt/android/tools$ telnet localhost 5554
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Android Console: type 'help' for a list of commands
OK
gsm
allows you to change GSM-related settings, or to make a new inbound phone call

available sub-commands:
list          list current phone calls
call          create inbound phone call
busy          close waiting outbound call as busy
hold          change the state of an outbound call to 'held'
accept        change the state of an outbound call to 'active'
cancel        disconnect an inbound or outbound phone call
data          modify data connection state
voice         modify voice connection state
status        display GSM status

```

Figure 7.3 An Android console session demonstrating the `gsm` command and available sub-commands.

With many of the larger telephony background details now complete, in the next few sections of this chapter we are going to cover some basic uses of the telephony APIs and other related facilities. We will examine intercepting calls, using some of the telephony utility classes, and simply making calls from your applications.

7.3 Interacting with the phone

In your day to day development you will often want to interact with the phone. This interaction may be as simple as dialing outbound calls through built-in intents, or may involve intercepting calls to modify them in some way. In this section we are going to cover these basic tasks, and we will also examine some of the phone number utilities Android provides for you “out of the box.”

One of the more common things you will do with the Android telephony support doesn't really involve the telephony APIs directly, that is making calls using the built-in intents.

7.3.1 Using Intents to make calls

As we demonstrated in chapter 4, using the `Intent.ACTION_CALL` action, and the “tel:” `Uri` is all you need to invoke the built-in dialer application and make a call. This approach will invoke the dialer application, populate the dialer with the provided telephone number (taken from the `Uri`) and initiate the call.

Along with this action you can also invoke the dialer application with the `Intent.ACTION_DIAL` action, which will again populate the dialer with the supplied phone number, but then stop short of actually initiating the call. Listing 7.4 demonstrates both of these techniques using the respective actions.

Listing 7.4 Using Intent actions to “dial” and “call” through the built-in dialer application.

```
dialintent = (Button) findViewById(R.id.dialintent_button);
dialintent.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        Intent intent = new Intent(Intent.DIAL_ACTION,      #1
                                   Uri.parse("tel:" + NUMBER)); #2
        startActivity(intent);
    }
});

callintent = (Button) findViewById(R.id.callintent_button);
callintent.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        Intent intent = new Intent(Intent.CALL_ACTION,      #3
                                   Uri.parse("tel:" + NUMBER));
        startActivity(intent);
    }
});
```

1. Usage of the `ACTION_DIAL`
2. Including the `tel:number` Uri
3. Usage of the `ACTION_CALL`

At this point the usage of intents and the Android platform design is something we have covered quite a bit. In listing 7.4 we are once again leveraging this design, to make outgoing calls to specified numbers.

Making calls using the built-in intents through the dialer application is very simple, as we have already seen in previous examples. Basically you need to set the action you want to take place, either populating the dialer with `ACTION_DIAL` **#1**, or populating the dialer **and** initiating a call with `ACTION_CALL`. In either case you also need to specify the telephone number you want to use with the intent Uri **#2**.

The only other aspect of dialing calls you need to be aware of are permissions. The correct permissions are required in your application manifest in order to be able to access and modify phone state, dial the phone, or intercept phone calls (which we will see in section 7.3.3). Table 7.1 lists the relevant phone related permissions and their purposes (for more detailed information see the security section of the Android documentation: <http://code.google.com/android/devel/security.html>).

Table 7.1 Phone related manifest permissions and purpose.

Phone related permission	Purpose
<code>android.permission.READ_PHONE_STATE</code>	Allow application to read phone state.
<code>android.permission.MODIFY_PHONE_STATE</code>	Allow application to modify phone state.

android.permission.CALL_PHONE	Initiate a phone call without user confirmation in dialer.
android.permission.CALL_PRIVILEGED	Call any number, including emergency, without confirmation in dialer.
android.permission.PROCESS_OUTGOING_CALLS	Allow application to receive broadcast for outgoing calls and modify.

Dialing the phone from an Android application is very straightforward. The built-in handling via intents and the dialer application make it almost trivial. Helping even more in terms of “making it nice for the people,” is the additional `PhoneNumberUtils` class that you can use to parse and validate phone number strings.

7.3.2 Helpful phone number related utilities

Applications running on mobile devices that support telephony get to experience the joy of dealing with a good deal of `String` formatting for phone numbers. Fortunately in the Android SDK there is a handy utility class that helps to mitigate the risks associated with this task, and standardize the way it's done - `PhoneNumberUtils`.

The `PhoneNumberUtils` class can be used to parse `String` data into phone numbers, parse alphabetical keypad digits into numbers, and determine other properties of phone numbers (such as whether or not they are global or localized). An example usage of this class is shown in listing 7.5.

Listing 7.5 Working with the `PhoneNumberUtils` class.

```

. . .

private TextView pnOutput;
private EditText pnInput;
private EditText pnInPlaceInput;
private Button pnFormat;

. . .

this.pnFormat.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {

        String phoneNumber = PhoneNumberUtils
            .formatNumber(PhoneNumberUtilsExample.this.pnInput.getText().toString());
#1

        phoneNumber = PhoneNumberUtils.convertKeypadLettersToDigits(
            PhoneNumberUtilsExample.this.pnInput.getText().toString());
#2

        StringBuilder result = new StringBuilder();
        result.append(phoneNumber);
        result.append("\nisGlobal - " +
            PhoneNumberUtils.isGlobalPhoneNumber(phoneNumber));    #|3
        result.append("\nisEmergency - " +

```

```

PhoneNumberUtils.isEmergencyNumber(phoneNumber));    #|3

        PhoneNumberUtilsExample.this.pnOutput.setText(result.toString());

        PhoneNumberUtilsExample.this.pnInput.setText("");
    }
});

```

1. Format as a phone number first
2. Convert keypad alpha characters to digits
3. Use several additional phone number utils methods

The `PhoneNumberUtils` class has a number of static helper methods for parsing phone numbers, the most simple of which is `formatNumber`. This method takes a single `String` as input and uses the default locale settings to return a formatted phone number **#1** (there are additional methods to format a number using a locale you specify, and to parse different segments of a number, and so on). Parsing a number can be combined with another helpful method, `convertKeypadLettersToDigits`, to further convert any alphabetic keypad letter characters into digits **#2**. The conversion method won't work unless it already recognizes the format of a phone number, so in this case it's important to run the format method first.

Along with these basic methods you can also check various properties of a number string, such as whether or not the number is global, and whether it represents an emergency call **#3**.

An additional way to format a phone number that is useful for any `Editable`, like the very common `EditText` (or `TextView`), is the `formatNumber` overload that edits these "in place." This method updates an `EditText` that is passed in when it is invoked. An example of using this is seen in listing 7.6.

Listing 7.6 Using the in place Editable View formatting offered by `PhoneNumberUtils`.

```

this.pnInPlaceInput.setOnFocusChangeListener(new OnFocusChangeListener() {    #1
    public void onFocusChange(View v, boolean b) {
        if (v.equals(PhoneNumberUtilsExample.this.pnInPlaceInput)
            && (b == false)) {
            PhoneNumberUtils.formatNumber(
                PhoneNumberUtilsExample.this.pnInPlaceInput.getText(),
                PhoneNumberUtils.FORMAT_NANP);    #2
        }
    }
});

```

1. Use the `OnFocusChangeListener` to invoke an update
2. Call the in place `formatNumber` method

The in place editor can be combined with a dynamic update step using various techniques, one is to make the update happen automatically when the focus changes away from a phone number field (curiously though, the in place edit does not also provide the keypad alphabetic character to number conversion automatically). To do this we have implemented an `OnFocusChangeListener` **#1**. Inside the `onFocusChange` method, which filters for the correct `View` item, we then call the `formatNumber` overload passing in the respective `Editable` and the formatting style we want to use **#2**. The "NANP" here stands for North American Numbering Plan, which includes an optional country and area code, and a 7 digit phone number.

Apart from using the phone number utilities, and making calls, you may also find a need to intercept calls.

7.3.3 Intercepting calls

There are many reasons you may want to intercept calls. For example, you may want to write an application that is aware of incoming phone calls and changes the ringer, or uses other different alerts based on the caller. Additionally, you want to write an application that catches outgoing calls and decorates or aborts them, based on certain criteria.

Intercepting outgoing calls is supported in the current Android SDK release, but unfortunately the same is not true of incoming calls. Currently incoming calls cannot be intercepted. Users can still change the ringer and other options for their contacts, but all of that is based on the built in applications, and is not something that is available to you as a developer through the APIs.

Because of the limitations in the API we will focus on what an intercept for an outgoing call looks like, which is shown in listing 7.7.

Listing 7.7 Catching and aborting an outgoing call.

```
public class OutgoingCallReceiver extends BroadcastReceiver {  
#1  
  
    public static final String ABORT_PHONE_NUMBER = "1231231234";  
  
    private static final String OUTGOING_CALL_ACTION =  
"android.intent.action.NEW_OUTGOING_CALL"; #2  
    private static final String INTENT_PHONE_NUMBER = "android.intent.extra.PHONE_NUMBER";  
#3  
  
    @Override  
    public void onReceive(final Context context, final Intent intent) {  
#4  
        if (intent.getAction().equals(OutgoingCallReceiver.OUTGOING_CALL_ACTION)) {  
#5  
  
            String phoneNumber = intent.getExtras().getString(INTENT_PHONE_NUMBER);  
#6  
            if ((phoneNumber != null)  
                &&  
                phoneNumber.equals(OutgoingCallReceiver.ABORT_PHONE_NUMBER)) {  
                Toast.makeText(context,  
                    "NEW_OUTGOING_CALL intercepted to number 123-123-1234  
                    - aborting call",  
                    Toast.LENGTH_LONG).show();  
#7  
                this.abortBroadcast();  
#8  
            }  
        }  
    }  
}
```

1. Create a broadcast receiver
2. Define a constant for the NEW_OUTGOING_CALL action
3. Define a constant for the PHONE_NUMBER Intent extra data
4. Implement the onReceive method
5. Filter the Intent for the desired action
6. Get the intent extras data
7. Show a quick message
8. Abort the intent

The first thing we do to intercept an outgoing call is to extend `BroadcastReceiver` **#1**. Our receiver defines several constants, one for the `NEW_OUTGOING_CALL` action **#2**, and one for the phone number data key `PHONE_NUMBER` **#3**.

For a `BroadcastReceiver` we have to implement the `onReceive` method **#4**. Within this method we filter on the intent action we want, `android.intent.action.NEW_OUTGOING_CALL` **#5**, and then we get the intent data using the phone number key **#6**. If the phone number matches we then send a `Toast` alert to the UI **#7** and abort the outgoing call simply by calling the `abortBroadcast` method **#8**.

Beyond dialing out, formatting numbers, and intercepting calls, another important area of the telephony support in Android is the support for sending and receiving SMS.

7.4 Working with SMS

The Short Message Service (SMS) is a hugely popular and important means of communication for mobile devices. SMS can be used to send simple text messages, and or small data chunks over voice channels on mobile networks.

Android includes a built-in SMS application that allows users to view received SMS messages, and send them (including replying to received messages). Along with the built in user-facing support, and related `ContentProvider` for interacting with the built-in system, the SDK also provides APIs for developers to be able to send and receive messages programatically.

To explore this support we are going to look at both sides of the coin, sending and receiving. The unadorned screen in figure 7.4 shows the SMS related `Activity` we will build in the `TelephonyExplorer` application.

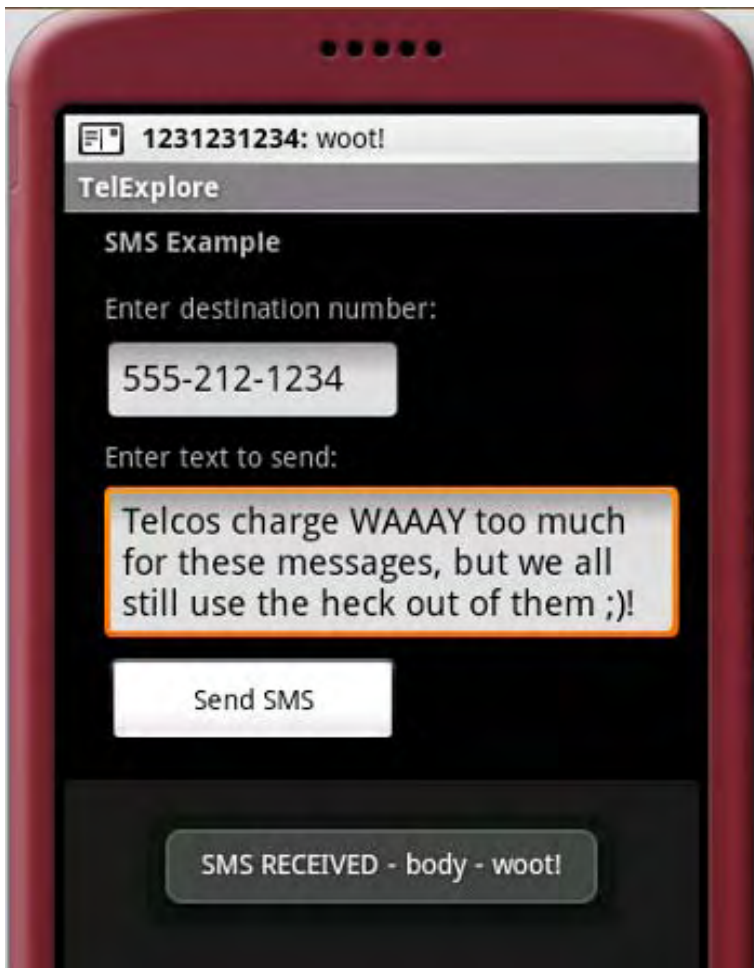


Figure 7.4 An Activity that sends SMS messages, and an example of an alert based on a received SMS message.

To get started working with SMS we will begin by sending SMS messages with the `SmsManager`.

7.4.1 Sending SMS messages

The `android.telephony.gsm` sub-package contains the `SmsManager` and `SmsMessage` classes. These are our SMS friends. The `SmsManager` is used to define many important SMS related constants, and it contains the `sendDataMessage`, `sendMultipartTextMessage`, and `sendTextMessage` methods.

In listing 7.8 we have an example from our `TelephonyExplorer` application of using the SMS manager to send a simple text message.

Listing 7.8 Using the `SmsManager` to send SMS messages.

```
// ... start of class omitted for brevity

private Button smsSend;
private SmsManager smsManager;

@Override
```



```

public void onCreate(final Bundle icle) {

    super.onCreate(icle);
    this setContentView(R.layout.smsexample);

    // . . . other onCreate view item inflation omitted for brevity

    this.smsSend = (Button) findViewById(R.id.smssend_button);

    this.smsManager = SmsManager.getDefault(); #1

    final PendingIntent sentIntent =
        PendingIntent.getActivity(
            this, 0, new Intent(this, SmsSendCheck.class), 0); #2

    this.smsSend.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {

            String dest = smsInputDest.getText().toString();
            if (PhoneNumberUtils.isWellFormedSmsAddress(dest)) { #3

                smsManager.sendTextMessage(
                    smsInputDest.getText().toString(),
                    null, smsInputText.getText().toString(),
                    sentIntent, null); #4

                Toast.makeText(SmsExample.this,
                    "SMS message sent",
                    Toast.LENGTH_LONG).show();

            } else {
                Toast.makeText(SmsExample.this,
                    "SMS destination invalid - try again",
                    Toast.LENGTH_LONG).show();
            }
        }
    });
}

```

1. Get a handle to the SmsManager
2. Create a PendingIntent to use after the message is sent
3. Check that the destination is valid
4. Send the message

The first thing we need to do in regard to working with SMS messages is obtain an instance of the `SmsManager`, which is done with the static `getDefault` method #1. The manager will be used later to send the message, before we can do that though we need to create a `PendingIntent` (which will be used as a parameter in the send method coming up).

WHAT IS A PendingIntent?

A `PendingIntent` is a specification of a future intent. These are basically a way for you to pass a future intent to another application, and allow that application to execute that intent as if it had the same permissions as your application, whether or not your application is still around when the intent is eventually invoked. Remember the `Activity` life-cycle, and the separate process logic that the platform uses. A `PendingIntent` provides a means for applications to in essence work “beyond the grave,” for a particular intent. Even after an owning application that creates a `PendingIntent` has been killed, that intent can still be run later.

A `PendingIntent` can specify an `Activity`, `Broadcast`, or `Service` that it requires. In our case we are using the `getActivity` method, which denotes an `Activity`, and then we are specifying the context, request

code (which is unused), the intent, and additional “flags” #2. The flags indicate whether or not a new instance of the referenced Activity (or Broadcast, or Service) should be created if one does not already exist.

Once we have a `PendingIntent`, we then check that the destination address is valid for SMS (using another method from `PhoneNumberUtils`) #3, and send the message using the manager's `sendTextMessage` method #4.

This send method takes in several parameters, one of which can be confusing. The signature of this method is as follows:

```
sendDataMessage(String destinationAddress, String scAddress, short destinationPort, byte[] data, PendingIntent sentIntent, PendingIntent deliveryIntent)
```

The `destinationAddress` is simple, this is the phone number you want to send the message to. The `scAddress` is the tricky one. This is not meant to be the “source” address, but rather indicates the internal service center address on the network, this should be left null in most cases (which uses the default). The `destinationPort` is also simple, it's the port. The data is the payload of the message. Finally, the `sentIntent` and `deliveryIntent` are separate `PendingIntent` instances that are fired when the message is successfully sent, and received, respectively.

Much like the permissions we saw in table 7.1 in reference to phone permissions, SMS related tasks also require manifest permissions. The SMS related permissions are shown in table 7.2.

Table 7.2 SMS related manifest permissions and purpose.

Phone related permission	Purpose
<code>android.permission.RECEIVE_SMS</code>	Allow application to monitor incoming SMS messages.
<code>android.permission.READ_SMS</code>	Allow application to read SMS messages.
<code>android.permission.SEND_SMS</code>	Allow application to send SMS messages.
<code>android.permission.WRITE_SMS</code>	Write SMS messages to the built in SMS provider (not related to sending messages directly).

Along with sending text and data messages using this basic pattern, you can also create an SMS related `BroadcastReceiver` to receive incoming SMS messages.

7.4.2 Receiving SMS messages

Receiving an SMS message programatically is done through receiving a broadcast on the Android platform. To see this in action with our `TelephonyExplorer` application we are again going to implement a receiver, as shown in listing 7.9.

Listing 7.9 Creating an SMS related BroadcastReceiver.

```
public class SmsReceiver extends BroadcastReceiver {                                #1
    public static final String SMSRECEIVED = "SMSR";
    private static final String SMS_REC_ACTION = "android.provider.Telephony.SMS_RECEIVED";
```

#2

```
@Override
public void onReceive(final Context context, final Intent intent) {

    if (intent.getAction().equals(SmsReceiver.SMS_REC_ACTION)) {

#3        StringBuilder sb = new StringBuilder();

        Bundle bundle = intent.getExtras();
        if (bundle != null) {
            Object[] pdus = (Object[]) bundle.get("pdus");

#4            for (Object pdu : pdus) {
                SmsMessage smsMessage = SmsMessage.createFromPdu((byte[]) pdu);

#5                sb.append("body - " + smsMessage.getDisplayMessageBody());

#6            }
        }
        Toast.makeText(context, "SMS RECEIVED - " + sb.toString(),
            Toast.LENGTH_LONG).show();
    }
}
```

1. Extend BroadcastReceiver
2. Define a constant for the SMS_RECEIVED action
3. Filter for the correct action in the receiver
4. Get the "pdus" from the intent Bundle
5. Create an SmsMessage from each pdu
6. Get the message body to display

To react to an incoming SMS message we once again are creating a BroadcastReceiver by extending that class **#1**. Our receiver defines a local constant for the intent action it wants to catch, in this case `android.provider.Telephony.SMS_RECEIVED` **#2**.

Once the class setup is ready we then filter for the action we want in the `onReceive` method **#3**, and get the SMS data from the intent "extras" Bundle using the key "pdus" **#4**. PDU, or Protocol Data Unit, is the term that describes the data packet in SMS messages. In this case the platform is using the String key "pdus" (we discovered this by trial and error, by getting the key Set from the Bundle and iterating it). For every "pdu" Object we then construct an `SmsMessage` by casting the data to a byte array **#5**. Once in `SmsMessage` form we can then work with the methods on that class, such as `getDisplayMessageBody` **#6**.

Sending and receiving messages in SMS form completes our exploration of the telephony APIs.

7.5 Summary

In our trip through the Android telephony related APIs we covered several important topics. We began with a brief overview of some of the telephony terms, and then we moved on to the Android specific APIs.

With the APIs we looked at accessing telephony information with the `TelephonyManager`, including device and SIM card data, and phone state. From there we also addressed hooking in a `PhoneStateListener` to get updates when the phone state changed, and reacting to such events.

Beyond retrieving the data we also looked at dialing the phone using built in intents and actions, intercepting outgoing phone calls, and several uses of the `PhoneNumberUtils` class. After the standard voice usages were covered we also addressed SMS messaging.

Here we looked at how to send and receive SMS messages using the `SmsManager` and `SmsMessage` classes.

In the next chapter we turn to the specifics of dealing with Notifications and Alerts on the Android platform.

8

Notifications and Alarms

In this chapter,

- Android Notifications
- Building a SMS Notification Application
- Alarms and the AlarmManger
- A Simple Alarm Example

Today's cell phones are expected not only to be phones but personal assistants, camera's, music and video players, instant messaging clients, as well as just about everything else a computer might do. With all these applications running on phones, applications need a way to notify users to get their attention or to take some sort of action be it in response to a Short Message Service (SMS), new voicemail, or an alarm reminding you of a new appointment.

In this chapter we are going to look at how to use Android BroadcastReceiver and the `AlarmManager` to notify users of just these sorts of events. In this chapter you will learn what a Toast is, a Notification is, how to use the Notification Manager, and how to display Notifications to the user or trigger some other action. You will also learn how to create Alarms and use the `AlarmManager` to schedule your alarm events. Before we go too deeply into how notifications work, let us first create a simple example application.

8.1 Introducing Toasts

For our example we will create a simple "Receiver" class that listens for a SMS text message and when a message arrives briefly pops up a message, called a *Toast*, to the user with the content of the message. A Toast is a simple, non-persistent, message that is designed to alert the user of some occurring event. Toasts are a great way to let a user know that a call is coming in, a SMS or email has arrived in their in box, or some other event has just happened.

To look at how we can use a Toast lets create a simple example. To build the example, first, create a new project called SMSNotifyExample in Eclipse. You can use whatever package name you like but for this chapter we will use "`com.msi.manning.chapter8`". Now that you have created the project lets first edit the AndroidManifest.xml. You will need to add some tags so that your AndroidManifest.xml looks like listing 8.1.

Listing 8.1: AndroidManifest.xml for SMSNotifyExample.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.chapter8">
    <uses-permission android:name="android.permission.RECEIVE_SMS" />      #1
    <application android:icon="@drawable/chat">                             #2
        <activity android:name=".SMSNotifyActivity" 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=".SMSNotifyExample"> #3
        <intent-filter>
            <action android:name="android.provider.Telephony.SMS_RECEIVED" />
        </intent-filter>
    </receiver>
</application>
</manifest>

```

1. Define user permissions to allow SMS messages.
2. Define a receiver, SMSNotify, with an intent filter.

The AndroidManifest.xml file needs to have specific user permissions #1 added to it to allow incoming SMS messages. Android security models default is to have no permissions associated to applications meaning applications can essentially do nothing that might cause the device or the data on the device harm. To provide Android permission you need to use one or more uses-permissions and in Chapter 9 we will go into greater detail of Androids security model.

The next part #2 of the AndroidManifest.xml file we define SMSNotifyActivity which is simply our Activity and the next class is the SMSNotifyExample class #3 which will act as our receiver. Next we will create a simple Activity class called SMSNotifyActivity like in Listing 8.2.

Listing 8.2: SMSActivity for the SMSNotifyExample.

```

package com.msi.manning.chapter8;
import android.app.Activity;
import android.os.Bundle;

public class SMSNotifyActivity extends Activity {

    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.main);
    }
}

```

As you can see there is very little to listing 8.2 in part because for this first example we will not be doing much with the Activity. Later in this chapter though we will build upon this class. Now let us create our Receiver class (see Chapter 5 for more about Intent Receiver's) which will actually listen for the SMS message and fire off an action. Listing 8.3 shows the code for our SMSNotifyExample.

Listing 8.3: A sample SMS IntentReceiver.

```

package com.msi.manning.chapter8.SMSNotifyExample2;

import android.content.Context;
import android.content.Intent;
import android.content.BroadcastReceiver; #1
import android.os.Bundle;
import android.provider.Telephony;
import android.telephony.gsm.SmsMessage;
import android.util.Log;
import android.widget.Toast; #1

public class SMSNotifyExample extends BroadcastReceiver { #2
    private static final String LOG_TAG = "SMSReceiver";

    public static final int NOTIFICATION_ID_RECEIVED = 0x1221;
}

```

```

        static final String ACTION = "android.provider.Telephony.SMS_RECEIVED"; #3

    public void onReceiveIntent(Context context, Intent intent) {

        if (intent.getAction().equals(ACTION)) {
            StringBuilder sb = new StringBuilder();

            Bundle bundle = intent.getExtras();
            if (bundle != null) {

                SmsMessage[] messages = #4
                    Telephony.Sms.Intents.getMessagesFromIntent(intent);
                for (SmsMessage currentMessage : messages){
                    sb.append("Received SMS\nFrom: ");
                    sb.append(currentMessage.getDisplayOriginatingAddress());
                    sb.append("\n---Message---\n");
                    sb.append(currentMessage.getDisplayMessageBody());

                }
            }
            Log.i(LOG_TAG, "[SMSApp] onReceiveIntent: " + sb);
            Toast.makeText(context, sb.toString(), Toast.LENGTH_LONG).show();
            #4
        }
    }

    @Override
    public void onReceive(Context context, Intent intent) {
    }
}

```

1. Import the BroadcastReceiver and Toast.
2. Extend the class as a BroadcastReceiver
3. Action fired by Android when a SMS is received.
4. Build message to share to the user.
5. Create a Toast.

Listing 8.3 should be very simple to follow. First you should take note that we are importing the BroadcastReceiver and Toast classes which we will need. Next we extend the class using BroadcastReceiver which allows the class to receive intents #2. Next we create a String #3 to hold the action that will be fired by the system when a SMS is received. After that we create a simple method to notify the user that a SMS message has been received and parse the SMS message to show who it was from and the content of the message #4. Finally we use a Toast to provide a quick message to the user #5. Toast's are transient little messages they pop up and provide the user with a quick information without interrupting what they are doing. In our code we use change two methods together using the form Toast.makeText(Context context, CharSequence text, int duration).show() where the first method contains a text view to for the user and the second method, show(), shows the message to the user. Toast allows you to set a specific view using setView but for our example we allow it to show the default which is the Android status bar.

So once you are done cutting and pasting the code, everything should automatically compile and you should be able to run the application. The application should come up and look like figure 8.1.



Figure 8.1: A simple Toast show n running in the emulator:, the SMSNotifyExample

To test our application, select the Dalvik Debug Monitor Service (DDMS) option in Eclipse. Now in the telephony actions field type a telephone number for example 17035558679. Now select SMS and type in a message in the message field provided below then select Send. Your message should be sent to the emulator, you should be able to see the emulator responding in the Eclipse console, and a message should pop up in the Android status bar on the very top of the Android screen representation as in figure 8.2.

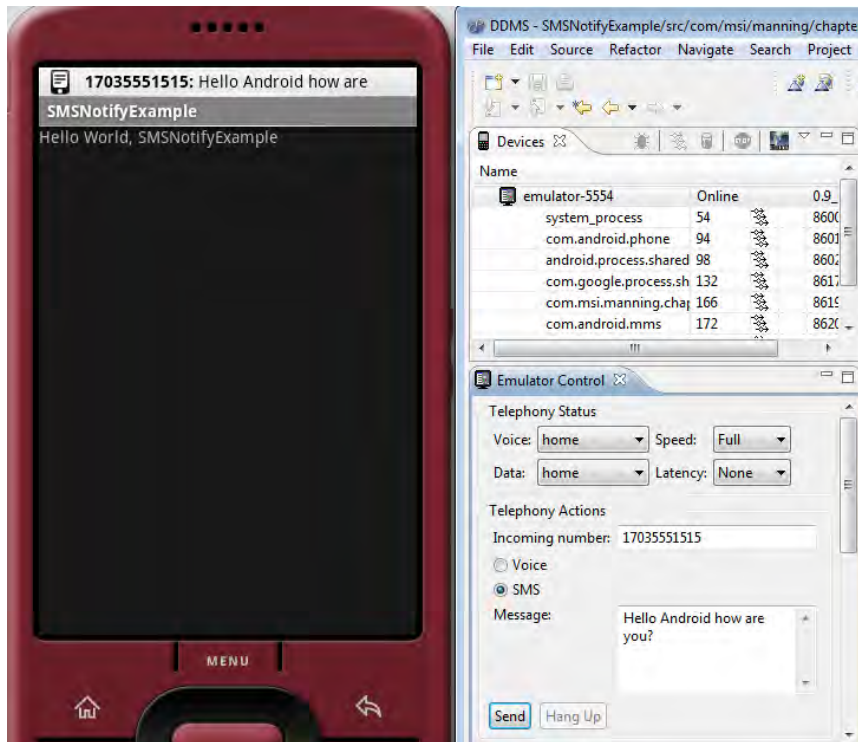


Figure 8.2: Example of a Toast message being generated from a SMS message.

So now that we have created our simple example, know how to display a short message on receiving a SMS, and know how to use the emulator to create a SMS, let us look at how to create a more persistent message that can also be used to set of LED's, play a sound, etc., to let the user know something has happened.

8.2 Introducing Notifications

In the previous section we showed how simple it is to create a quick, unobtrusive, message to the user to let them know they had received a SMS message. In this next section we are going to look at how we can create a persistent notification that not only shows up in the status bar but stays in a notification area until the user deletes it. To do that we need to use class Notification since we want to do something more complex than Toast can offer us. To do this we are going to need to use the Notification class.

As we have talked about before Notifications on Android can be many things from a pop up message, flashing LED, or the vibration of your phone but all of these actions start with and are represented by the Notifications class. The Notifications class defines how you want to represent a notification to a user and has three constructors, one public method and a number of fields. Table 8.1 provides a summary of the class:

Table 8.1 : Table of notification Fields

Access	Type	Method	Description
public	boolean	insistent	Audio and vibration notification will be repeated while True.
public	int	ledARGB	The color of the LED notification.
public	int	ledOffMS	The number of milliseconds for LED to be off between flashes.
public	int	ledOnMS	The number of milliseconds for LED to be on between flashes.
public	boolean	lights	LED enabled for notification when set to True.
public	ContentURI	sound	The sound to play.
public	RemoteViews	statusBarBalloon	View to display when the

			statusBarIcon is selected in the status bar.
public	CharSequence	statusBarBalloonText	Text to display when the statusBarIcon is selected in the status bar.
public	Intent	statusBarClickIntent	The intent to execute when the icon is clicked.
public	int	statusBarIcon	The resource id of a drawable to use as the icon in the status bar.
public	CharSequence	statusBarTickerText	Text to scroll across the screen when this item is added to the status bar.
public	long[]	vibrate	The pattern with which to vibrate.

As you can see there are a lot of fields for the Notification class since it has to describe every way you can notify a user. Using a notification is as simple as:

```
Notification notif = new Notification(
    context,           // the application context
    icon,              // the icon for the status bar
    tickerText,        // the text to display in the ticker
    when,              // the timestamp for the notification
    Title,             // the title for the notification
    TextBody,          // the details to display in the notification
    contentIntent,     // the contentIntent
    appIntent);        // the application intent
```

and to send the notification all you have to do is:

```
nm.notify(String, Notification);
```

Where **nm** is the reference to the **NotificationManager**. Now let us take our previous example and edit to change it from a Toast notification to a notification in the status bar. Before we do that let's make the application a little more interesting by adding a few more icons to our resources directory. For this example we are going to use the chat.png icon and the incoming.png icon you can find in the downloaded code for this book or you can get them from <http://www.manning.com/ablesen/>. Simply drop them in our **res/drawable** directory to have Eclipse automatically register them for us in the R class.

Now that that is done let's edit our code. First let us edit the **SMSNotifyActivity** class so that when the Activity is called it can find the Notification passed to it from the Notification Manager and then after it has run cancel it. Listing 8.4 provides the code you need for new **SMSNotifyActivity** class.

Listing 8.4: A sample SMSNotifyActivity.

```
package com.msi.manning.chapter8;

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

public class SMSNotifyActivity extends Activity {

    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.main);

        NotificationManager nm = (NotificationManager ) #1
        getSystemService(NOTIFICATION_SERVICE);
        nm.cancel(R.string.app_name); #2
    }
}
```

```
    }
}
```

Setup the NotificationManger so we can act on a Notification.

Cancel the notification.

As you can see all we did is used the **NotificationManager** #1 to look up the **Notification** and then used the **cancel()** #2 method to cancel it. We could do more here, such as setup a custom view but for now we will leave it as is.

Next we need to edit the **SMSNotifyExample** to remove the **Toast** notification and support a **Notification** to the status bar. Listing 8.5 shows the edits we need to make.

Listing 8.5: Updated SMSNotifyExample.java

```
package com.msi.manning.chapter8;

import android.app.Notification;           #1
import android.app.NotificationManager;    #1
import android.content.Context;
import android.content.Intent;
import android.content.BroadcastReceiver;
import android.os.Bundle;
import android.app.PendingIntent;
import android.provider.Telephony;
import android.telephony.gsm.SmsMessage;
import android.util.Log;

public class SMSNotifyExample extends BroadcastReceiver {
    /** TAG used for Debug-Logging */
    private static final String LOG_TAG = "SMSReceiver";
    public static final int NOTIFICATION_ID_RECEIVED = 0x1221;
    static final String ACTION =
"android.provider.Telephony.SMS_RECEIVED";
    private CharSequence from = null;      #2
    private CharSequence tickerMessage = null; #3

    @SuppressWarnings("deprecation")
    public void onReceiveIntent(Context context, Intent intent) {

        NotificationManager nm = (NotificationManager)
context.getSystemService(Context.NOTIFICATION_SERVICE);
        if (intent.getAction().equals(ACTION)) {

            StringBuilder sb = new StringBuilder();

            Bundle bundle = intent.getExtras();
            if (bundle != null) {

                SmsMessage[] messages =
Telephony.Sms.Intents.getMessagesFromIntent(intent);

                for (SmsMessage currentMessage : messages){
```

```

        sb.append("Received compressed SMS\nFrom: ");
sb.append(currentMessage.getDisplayOriginatingAddress());
        from =
currentMessage.getDisplayOriginatingAddress();
        sb.append("\n---Message---\n");
        sb.append(currentMessage.getDisplayMessageBody());
    }
}

Log.i(LOG_TAG, "[SMSApp] onReceiveIntent: " + sb);
this.abortBroadcast();

Intent i = new Intent(context, SMSNotifyActivity.class);
context.startActivity(i);

CharSequence appName = "SMSNotifyExample";
tickerMessage = sb.toString();
Long theWhen = System.currentTimeMillis();

PendingIntent.getBroadcast((Context) appName, 0, i, 0); #4
Notification notif = new Notification(
    context,
    R.drawable.incoming,
    tickerMessage,
    theWhen,
    from,
    tickerMessage,
    i);

    notif.vibrate = new long[] { 100, 250, 100, 500};
nm.notify(R.string.alert_message, notif); #5
}
}

@Override
public void onReceive(Context context, Intent intent) {
}

```

1. } Import Notification and the NotificationManger
2. Add two new variables to be used in creating a Notification
3. Create the Application Intent
4. Build the Notification
5. Broadcast the Notification

You will notice that the first change we made was to import Notification #1. Next we add a few new variables, one called **from**, and one called **tickerMessage** #2. The variable **from** will hold the information on who sent the SMS message while the **tickerMessage** will hold the actual SMS message that we want to scroll in the notification bar. We add these fields right after our Action variable like:

```

private CharSequence from = null;
private CharSequence tickerMessage = null;

```

Next we create an Application Intent #3. The Application Intent will be the Intent shown when we click on the SMS inbox. For this example it will not really do anything but it is required for building the **Notification**. You could though have it pop up editor or some other screen with a little more effort.

Once the Application Intent is set we can generate the Notification #4. To make the code easier to understand we have add some comments next to each attribute of Notification from Listing 8.5.

```
Notification notif = new Notification(
    context,                // our context
    R.drawable.incoming,    // the icon for the status bar
    tickerMessage,          // the text to display in the ticker
    theWhen,               // the timestamp for the notification
    from,                  // the title for the notification
    tickerMessage,         // the details to display in the notification
    i,                     // the content Intent
    appIntent);            // shows the inbox when you click on icon

nm.notify(R.string.app_name, notif);
```

And the last line we use the `notify()` method #5 from the `NotificationManager` to broadcast our notification to the application.

Now if you run the application, then open the DDMS, and pass a SMS message as you did earlier you should see the new notification appear in the Status bar where the message shows each line after a short interval till the message is fully display. You should also see a new icon pop up in the status bar indicating a new SMS message like in Figure 8.3.

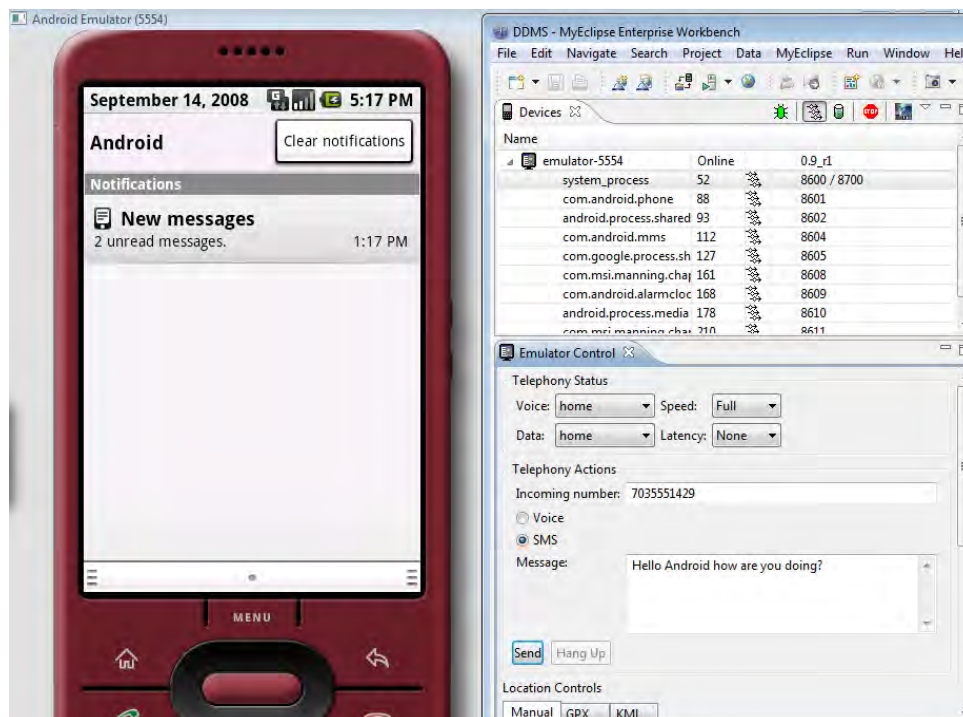


Figure 8.3: This image shows how to use the Android DDMS to send a simulated SMS message to the application.

When you have sent the message you can click on the new message icon and a bar should drop down from it. Click on the bar and drag it down to the bottom of the screen. This now opens up the default view of the SMS inbox for Android that you can see in Figure 8.4.

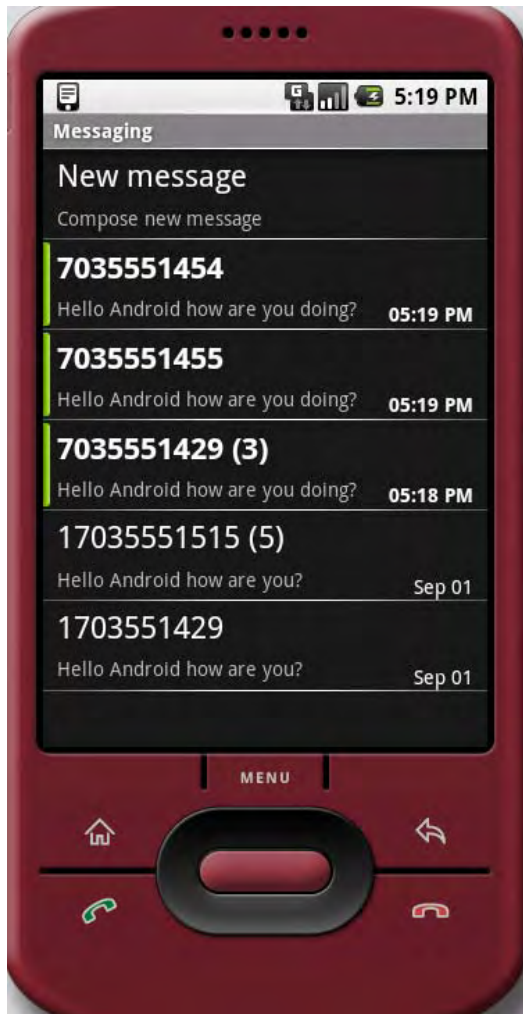


Figure 8.4: Image showing the SMS inbox after it has been expanded and clarification of how the `contentIntent` and `appIntent` are displayed.

There is a lot more you could do with this demo such as creating a better user interface or making the SMS inbox more feature rich. You could even play with notifications some more to allow it to play a sound when a message arrives but for this example we have looked at everything you need to know to start working with Notifications. In the next section we are going to look at Notifications close relative, the Alarm.

8.3 ALARMS

In Android, Alarms allow you to simply schedule your application run at some point in the future. Alarms can be used for a wide range of applications from notifying a user an appointment to something more sophisticated such as having an application start up, check for software updates, and then shutdown. Alarms work by simply registering an Intent with the Alarm and then at the time scheduled the Alarm will broadcast the Intent. Android will automatically start the targeted application even if it is not started or if Android handset is asleep.

Android manages all Alarms somewhat like the `NotificationManger` via an `AlarmManager` class. The `AlarmManager` has only three methods these being `cancel`, `set`, and `setRepeating` as showing in table 8.2.

Table 8.2: `AlarmManger` Public Methods.

Returns	Method and Description
---------	------------------------

void	cancel(Intent intent) Remove alarms with matching Intent.
void	set(int type, long triggerAtTime, Intent intent) Used to set an alarm.
void	setRepeating(int type, long triggerAtTime, long interval, Intent intent) Used to set a repeating alarm.
Void	setTimeZone(String TimeZone) Used to set the time zone for the alarm.

You instantiate the AlarmManager indirectly as you do the NotificationManager by using Context.getSystemService(Context.ALARM_SERVICE).

Setting Alarms is very easy, like most things in Android, and in the next example we will create a simple application that sets an Alarm when a button is pushed and then when the Alarm is triggered it will pass back a simple Toast to inform us that the Alarm has been fired.

8.3.1 Alarm Example

In this next example we are going to create a new Android Project called SimpleAlarm with the package com.msi.manning.chapter8.simpleAlarm, an application name of SimpleAlarm and an Activity name of GenerateAlarm. In this project we will use another open source Icon which you can find at <http://www.manning.com/ablesen/> or in the download for this chapter. Change the name of the icon to clock and add it to the res/drawable directory of the project when you create it.

Next we need to edit the AndroidManifest.xml to have a receiver #1, which we will create soon, called AlarmReceiver as in listing 8.6.

Listing 8.6: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.chapter8.simpleAlarm">
    <application android:icon="@drawable/clock">
        <activity android:name=".GenerateAlarm" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".AlarmReceiver" android:process=":remote" />    #1
    </application>
</manifest>
```

1. define the receiver.

Now that you have done this edit the string.xml in the values directory and add two new strings

```
<string name="set_alarm_text">Set Alarm</string>
<string name="alarm_message">Alarm Fired</string>
```

We will use this string to be the value of the button in our layout. Now we need to add a new button to our layout so edit the main.xml to add a new button like:

```
<Button android:id="@+id/set_alarm_button"        android:layout_width="wrap_content"
    android:layout_height="wrap_content"
        android:text="@string/set_alarm_text">
    <requestFocus />
</Button>
```

Now we are ready to create a new class that will act as the Receiver for the notification the Alarm will generate. In this case we are going to be generating a Toast style notification to let the user know that the Alarm has been triggered. Now create a new class like listing 8.7 which simply waits for the Alarm to broadcast to the AlarmReceiver and will generate a toast.

Listing 8.7: AlarmReceiver.java

```
package com.msi.manning.chapter8.simpleAlarm;
```

```

import android.content.Context;
import android.content.Intent;
import android.content.IntentReceiver;
import android.widget.Toast;

public class AlarmReceiver extends IntentReceiver {

    @Override
    public void onReceiveIntent(Context context, Intent intent)           #1
    {
        Toast.makeText(context, R.string.app_name, Toast.LENGTH_SHORT).show(); #2
    }
}

```

1. Create the onReceiveIntent method.
2. Broadcast a Toast when the Intent is received.

Next we need to edit the **SimpleAlarm** class to create a simple button widget (as discussed in chapter 3) that calls the inner class **setAlarm**. In **setAlarm** we create an **onClick** method that will schedule our alarm, call our intent, and fire off our Toast. Listing 8.8 shows what the finished file class should look like.

Listing 8.8: SimpleAlarm.java

```

package com.msi.manning.chapter8.simpleAlarm;

import android.app.Activity;
import android.app.AlarmManager;           #1
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

import java.util.Calendar;

public class GenerateAlarm extends Activity
{
    Toast mToast;

    @Override
    protected void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.main);
        Button button = (Button)findViewById(R.id.set_alarm_button);
        button.setOnClickListener(mOneShotListener);           #2
    }

    private OnClickListener mOneShotListener = new OnClickListener()
    {
        public void onClick(View v)
        {
            Intent intent = new Intent(GenerateAlarm.this, AlarmReceiver.class);
            #3

            PendingIntent appIntent = PendingIntent.getBroadcast(GenerateAlarm.this, 0,
            intent, 0);
            #3

            Calendar calendar = Calendar.getInstance();           #4

```

```

        calendar.setTimeInMillis(System.currentTimeMillis());
        calendar.add(Calendar.SECOND, 30);

        AlarmManager am = (AlarmManager) getSystemService(ALARM_SERVICE);      #5
        am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(), appIntent);  #6

        if (mToast != null) {mToast.cancel();
        }
        mToast = Toast.makeText(GenerateAlarm.this, R.string.alarm_message,
        Toast.LENGTH_LONG);
        mToast.show();
    }
};
}}

```

1. Import AlarmManager.
2. Setup the button to call mOneShotListener.
3. Create the Intent to fire when the Alarm goes off.
4. Set the time in milliseconds for when we want the Alarm to go off.
5. Create the AlarmManager.
6. Set the Alarm.

As you can see this is a pretty simple class. We first import the **AlarmManager #1** so we can work with Alarms and then create a button to trigger our Alarm **#2**. Next we create inner class for our **mOneShotListener**. We then create the Intent to be triggered when the alarm actually goes off **#3**. In the next section of code we use the **Calendar** class **#4** to help us calculate the number of milliseconds from the time of when the button is pressed which we will use to set the alarm.

Now we have everything we need done to create and set the Alarm. To do this we first create the AlarmManger **#5** and then call its **set()** method to set the Alarm **#6**. To go into a little more detail of what is going on in the application take a look at:

```

AlarmManager am = (AlarmManager) getSystemService(ALARM_SERVICE);
am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(), intent);

```

Which is where we actually create and set the Alarm by first using **getSystemService** to create the **AlarmManager**. The first parameter we pass to the **set()** method is **RTC_WAKEUP** which is an integer representing the alarm type we want to set. The AlarmManger currently (as of M5) supports four alarm types as shown in table 8.3.

Table 8.3. AlarmManager Alarm types

TYPE	Description
ELAPSED_REALTIME	Alarm time in SystemClock.elapsedRealtime() (time since boot, including sleep).
ELAPSED_REALTIME_WAKEUP	Alarm time in SystemClock.elapsedRealtime() (time since boot, including sleep), which will wake up the device when it goes off.
RTC	Alarm time in System.currentTimeMillis() (wall clock time in UTC).
RTC_WAKEUP	Alarm time in System.currentTimeMillis() (wall clock time in UTC), which will wake up the device when it goes off.

As you can see there are multiple types of alarms which you can use depending on your requirements. The **RTC_WAKEUP** for example sets the Alarm time in milliseconds and when the alarm goes off it will wake up the device from sleep mode for you as opposed to **RTC** which will not.

The next parameter we pass to the method is the amount of time in milliseconds for when we want the alarm to be triggered which we set with:

```

Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.add(Calendar.SECOND, 30);

```

The last parameter is the Intent we want to broadcast to which is our IntentReceiver. So now if you build the application and run it in the emulator you should see something like figure 8.5.



Figure 8.5: Example of the SimpleAlarm application running in the emulator.

Selecting the 'Set Alarm' button will set the alarm and after thirty seconds you should see something like Figure 8.6 displaying the Toast message.



Figure 8.6: After the Alarm runs the application shows a simple Toast message.

As you can see creating an Alarm is pretty trivial in Android but what might make more sense is for that Alarm to trigger a **Notification** in the status bar. Well to do that you would need to just add a **NotificationManger** and generate a **Notification**. To do this we have created a new method to add to Listing 8.8 called **showNotification** which takes four parameters and creates our **Notification** like this:

```
private void showNotification(int statusBarIconID, int statusBarTextID, int
detailedTextID, boolean showIconOnly) {

    Intent contentIntent = new Intent(this, SetAlarm.class);
    PendingIntent theappIntent = PendingIntent.getBroadcast(SetAlarm.this, 0,
contentIntent, 0);
    CharSequence from = "Alarm Manager";
    CharSequence message = "The Alarm was fired";

    String tickerText = showIconOnly ? null : this.getString(statusBarTextID);
    Notification notif = new Notification( statusBarIconID, tickerText,
System.currentTimeMillis());

    notif.setLatestEventInfo(this, from, message, theappIntent);

    nm.notify(YOURAPP_NOTIFICATION_ID, notif );
}
```

As you can see much of this code is very similar to the [SMSNotifyExample](#) code. To add it to our [SimpleAlarm](#) you just need to edit listing 8.8 to look like listing 8.9 where the only other things we have done is first imported the Notification and [NotificationManger](#) to the code, and the private variables `nm`, and `ApplicationID`, and finally a call to `showNotification()` right after the `Toast`.

Listing 8.9: SimpleAlarm.java

```
package com.msi.manning.chapter8.NotifyAlarm;

import java.util.Calendar;
import android.app.Activity;
import android.app.AlarmManager;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.Toast;

public class SetAlarm extends Activity
{
    private NotificationManager nm;
    private int YOURAPP_NOTIFICATION_ID;
    Toast mToast;
    @Override
    protected void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.main);

        nm = (NotificationManager) getSystemService(NOTIFICATION_SERVICE);

        Button button = (Button)findViewById(R.id.set_alarm_button);
        button.setOnClickListener(mOneShotListener);
    }

    private void showNotification(int statusBarIconID, int statusBarTextID, int
detailedTextID, boolean showIconOnly) {

        Intent contentIntent = new Intent(this, SetAlarm.class);
        PendingIntent theappIntent = PendingIntent.getBroadcast(SetAlarm.this, 0,
contentIntent, 0);

        CharSequence from = "Alarm Manager";
        CharSequence message = "The Alarm was fired";

        String tickerText = showIconOnly ? null : this.getString(statusBarTextID);
        Notification notif = new Notification( statusBarIconID, tickerText,
System.currentTimeMillis());

        notif.setLatestEventInfo(this, from, message, theappIntent);

        nm.notify(YOURAPP_NOTIFICATION_ID, notif );
    }

    private OnClickListener mOneShotListener = new OnClickListener()
    {
        public void onClick(View v)
        {
            Intent intent = new Intent(SetAlarm.this, AlarmReceiver.class);
```

```

intent, 0);

PendingIntent appIntent = PendingIntent.getBroadcast(SetAlarm.this, 0,

Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.add(Calendar.SECOND, 30);

AlarmManager am = (AlarmManager) getSystemService(ALARM_SERVICE);
am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(), appIntent);

showNotification(
    R.drawable.alarm,
    R.string.alarm_message,
    R.string.alarm_message,
    false);
    }
};

```



Figure 8.7: Alarm notification showing in the status bar.

If you run the code and click on “Set Alarm” you should now see the alarm notification in the status bar like figure 8.7. You could easily edit this code to take in parameters for time and date, have it show different intents when the icons are clicked, etc. As you can see though from this example Android alarms and the AlarmManger are very straight forward and you should be able to easily integrate them now into your applications.

8.4 Summary

In this chapter we have looked at two separate but related items which are Notifications and Alarms. We have looked at how to use the NotificationManger to generate Notifications and how the Notification class can be used to present a Notification to the user by building a simple example that displays a Notification when a SMS messages arrives in the inbox.

We have also looked at how Alarms can be set to cause an application to start or take some action in the future include waking the system up from the sleep mode. Finally we looked at how we can trigger a Notification from an Alarm. While the code presented here in these simple examples gives you a taste of what can be done with Notifications and Alarms both have very broad applications limited only by your imagination.

One other topic that we will not cover in this application is how to manage other applications that are running when a notification is triggered. Usually this involves having the active application 'pause' while the user deals with the event that triggered the notification (like a incoming email). To do this you need to understand how to manage threads. In the next chapter you will learn just that, how to manage threads, and combined with this chapter you will have the ability to create applications which you can gracefully switch between without consuming all your system resources.

9

Graphics and Animation

In this chapter:

X Drawables - XML Art

OpenGL

Animations

One of the main features of Android that you should have picked up by now is how much easier developing Android applications are than in mobile application platforms. One place where this really stands out is in the creation of visually appealing user interfaces and metaphors but there is a limit of what can be done with typical Android user interface elements (such as those discussed in Chapter 3). In this chapter we are going to look at how to create our own graphics using Android's Graphic API, develop animations, and even look at Android's support for the OpenGL standard (to see some examples of what can be done with Android's graphics platform see <http://www.omnigsoft.com/Android/ADC/readme.html>). Later in the chapter we will even look at how users can interact or manipulate the graphical environment via the Surface manager and Touch Screen interaction.

If you have ever worked with graphics in Java you will most likely find the graphics API and how graphics work in Android familiar.

9.1 Drawing Graphics in Android

In this section we are going to be looking at Android's graphical capabilities as well as some examples of how to make simple two dimensional shapes. For our examples we will be making use of the `android.graphics` package (see <http://code.google.com/android/reference/android/graphics/package-summary.html>) which provides all the low level classes and tooling need to create graphics. The graphics package supports things like `bitmap` (which holds pixels, `canvas` (what your draw calls draw on), primitives such as rectangles or text, and `paint` which you use to add color and styling.

To demonstrate the basics of drawing a shape let us do a simple example where we will draw a rectangle.

Listing 9.1 ShapeExample

```
package com.msi.manning.chapter9.SimpleShape;

import android.app.Activity;
import android.content.Context;
import android.graphics.*; #1
import android.graphics.drawable.ShapeDrawable; #1
import android.graphics.drawable.shapes.*; #1
import android.os.Bundle;
import android.view.*;

public class SimpleShape extends Activity {

    @Override
    protected void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(new SimpleView(this));
    }

    private static class SimpleView extends View { #2
        private ShapeDrawable mDrawable = new ShapeDrawable(); #3

        public SimpleView(Context context) {
            super(context);
            setFocusable(true);
            mDrawable = new ShapeDrawable(new RectShape()); #4
            mDrawable.getPaint().setColor(0xFFFF0000);
        }

        @Override protected void onDraw(Canvas canvas) { #5

            int x = 10;
            int y = 10;
            int width = 300;
            int height = 50;
            mDrawable.setBounds(x, y, x + width, y + height); #6
            mDrawable.draw(canvas); #6
            y += height + 5;

        }
    }
}
```

1. Import the graphics packages
2. Create a View
3. Create a ShapeDrawable to hold our Drawable
4. Create a new Rectangle and assign it to mDrawable
5. The onDraw method is called to draw the graphics
6. Set the boundaries and draw the shape on the canvas

As you can see drawing a new shape is pretty simple. First you need to import the necessary packages #1 including graphics, then `ShapeDrawable` which will support adding shapes to our drawing, and then shapes which supports several generic shapes including `RectShape` which we will use. Next you need to create a view #2 and then a new `ShapeDrawable` to add our Drawables too #3. Once we have a `ShapeDrawable` we can assign shapes to it. In our code we use the `RectShape` #4 but we could have used either `OvalShape`, `PathShape`, `RectShape`, `RoundRectShape`, or `Shape`. Finally we create use the `onDraw()` method to draw the `Drawable` on the `Canvas` #5. Finally we use the `Drawable's setBounds()` method to set the boundary (a rectangle) in which we will draw or rectangle using the `draw()` method #6. When you run Listing 9.1 you should see a simple red rectangle like in figure 9.1.



Figure 9.1: A simple red rectangle drawn using Androids graphics API.

Another way to do the same thing is through the use of XML. Android allows you to define shapes to draw in a XML resource file.

9.1.1 DRAWING with XML

With Android you can create simple drawings using a simple XML file approach. To do this all you need to do is create a Drawable object or objects which are defined as an XML file in your drawable directory like res/drawable. For example, the XML to create a simple rectangle would look like Listing 9.2.

Listing 9.2 simplerectangle.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#FF0000FF" />
</shape>
```

With Android XML drawable shapes the default shape is a rectangle but you can change the shape type by using the type tag and selecting a value of oval, rectangle, line, or arc. To use this XML shape you need to reference it in a layout like Listing 9.3 where your layout would reside in res/layout.

Listing 9.3xmldrawable.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <ImageView android:layout_width="fill_parent"
            android:layout_height="50dip"
            android:src="@drawable/simplerectangle" />
    </LinearLayout>
</ScrollView>
```

Finally all you need to do is create a simple Activity where you place your UI in a contentView like listing 9.5.

Listing 9.5 XMLDraw.java

```
package com.msi.manning.chapter9.XMLDraw;

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

public class XMLDraw extends Activity {

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

If you run this code it will draw a simple rectangle. You can make more complex drawings or shapes by simply stacking or ordering your XML drawables, and you can include as many shapes as you want or need depending upon space. For example, you could change your `xmldrawable` to look like Listing 9.4, which adds a number of shapes and stacks them vertically.

Listing 9.4 `xmldrawable.xml`

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

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <ImageView android:layout_width="fill_parent"
            android:layout_height="50dip"
            android:src="@drawable/shape_1" />
        <ImageView android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:src="@drawable/line" />
        <ImageView
            android:layout_width="fill_parent"
            android:layout_height="50dip"
            android:src="@drawable/shape_2" />
        <ImageView
            android:layout_width="fill_parent"
            android:layout_height="50dip"
            android:src="@drawable/shape_5" />
    </LinearLayout>
</ScrollView>
```

Now you just need to add the shapes in the `res/drawable` folder in Listings 9.5, 9.6, 9.7 and 9.8.

Listing 9.5 `shape1.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    type="oval" >
    <solid android:color="#00000000"/>
    <padding android:left="10sp" android:top="4sp" android:right="10sp"
        android:bottom="4sp" />
    <stroke android:width="1dp" android:color="#FFFFFFF"/>
</shape>
```

In Listing 9.5 we are using a different shape type: `oval`. We have also added another tag called `padding` which allows us to define padding or space between the object and other

objects in the UI. We are also using the tag `stroke`, which allows us to define the style of the line that makes up the border of the oval (see Listing 9.6).

Listing 9.6 shape2.xml

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

<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#FF0000FF" />
    <stroke android:width="4dp" android:color="#FFFFFF"
        android:strokeWidth="1dp" android:strokeDashGap="2dp" />
    <padding android:left="7dp" android:top="7dp"
        android:right="7dp" android:bottom="7dp" />
    <corners android:radius="4dp" />
</shape>
```

With this shape we are generating another rectangle, but this time (see Listing 9.7) we introduce the tag `corners` that allows us to make rounded corners with the attribute `android:radius`.

Listing 9.7 shape5.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    type="oval">
    <gradient android:startColor="#FFFF0000" android:endColor="#80FF00FF"
        android:angle="270" />
    <padding android:left="7dp" android:top="7dp"
        android:right="7dp" android:bottom="7dp" />
    <corners android:radius="8dp" />
</shape>
```

In Listing 9.8 we have created a shape of the type `line` with a size tag using the `android:height` attribute, which allows us to describe the number of pixels used on the vertical to size the line.

Listing 9.8 line.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    type="line">
    <solid android:color="#FFFFFF" />
    <stroke android:width="1dp" android:color="#FFFFFF"
        android:strokeWidth="1dp" android:strokeDashGap="2dp" />
    <padding android:left="1dp" android:top="25dp"
        android:right="1dp" android:bottom="25dp" />

    <size android:height="23dp" />
</shape>
```

Now if you run this you should see something like Figure 9.2.



Figure 9.2: Various shapes drawn using xml.

Using Android's graphics API's you can draw anything you need from simple lines or figures to overlay onto a map to custom User Interface elements such as buttons. Since a common need for applications is custom UI's, take a look at a more complex example of drawing our own buttons using the graphics API. In this next example we are going to use a number of graphics classes including `Paint`, `Path`, `Paint Style` and `RectF` to draw our buttons and add styles to them.

As you can see drawing with Android is straight forward and Android provides the ability for developers to programmatically draw anything they might need. In the next section we are going to look at what you can draw with Android's animations capabilities.

9.2 Animations

If a picture says a thousand words, then an animation must speak volumes. Android supports multiple methods of Animations including through XML like you saw in chapter 3 section X, or via Android's XML frame by frame animations, using Android graphics API, or

Android's support for OpenGL ES. In this section we are going to look at how to use the graphics API to create a very simple animation of a bouncing ball using Android's Frame by Frame Animation.

Android allows you to create simple animations by allowing you to show a set of images one after another to give the allusion of movement much like stop motion film. Android does this by setting each frame image as a drawable resource that is then shown one after the other in the background of a View. To do this you just need to define a set of resources in a XML file and then simply call `AnimationDrawable.run()`.

To demonstrate this method for creating an animation, first you will need to download the images for this chapter from the books website at {whats the URL>?}. The images for this are six representations of a ball bouncing. Next, you will need to create a new project called XMLanimation. Now create a new directory called /anim under the /res resources directory. Place all of the images for this example in the /drawable directory. Now create a new XML file called simple_animation.xml like that shown in Listing 9.9.

Listing 9.9 Simple_animation.xml

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
id="selected" android:oneshot="false">
  <item android:drawable="@drawable/ball1" android:duration="50" />
  <item android:drawable="@drawable/ball2" android:duration="50" />
  <item android:drawable="@drawable/ball3" android:duration="50" />
  <item android:drawable="@drawable/ball4" android:duration="50" />
  <item android:drawable="@drawable/ball5" android:duration="50" />
  <item android:drawable="@drawable/ball6" android:duration="50" />
</animation-list>
```

The XML file simply defines the images that make up the list of images to be displayed for the animation. The XML `<animation-list>` tag contains all of the item tags that contain the two attributes `drawable`, which describes the path to the image, and `duration`, which describes the time to show the image in nanoseconds. Now that the animation XML file has been created, edit the main.xml file to look like listing 9.10.

Listing 9.10 Simple_animation.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <ImageView android:id="@+id/simple_anim"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:layout_centerHorizontal="true"
    />
  <TextView
    android:layout_width="fill_parent"
```

```

        android:layout_height="wrap_content"
        android:text="Hello World, XMLAnimation"
    />
</LinearLayout>

```

All we have done here is added an `ImageView` tag that sets up the layout for our `ImageView`. Finally we now create the code to actually run the animation in Listing 9.11.

Listing 9.11 `xmlanimation.java`

```

package com.msi.manning.chapter9.xmlanimation;

import android.app.Activity;
import android.graphics.drawable.AnimationDrawable;
import android.os.Bundle;
import android.widget.ImageView;
import java.util.Timer;
import java.util.TimerTask;

public class XMLAnimation extends Activity
{
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.main);
        ImageView img = (ImageView)findViewById(R.id.simple_anim); #1
        img.setBackground(R.anim.simple_animation); #1

        MyAnimationRoutine mar = new MyAnimationRoutine(); #2
        MyAnimationRoutine2 mar2 = new MyAnimationRoutine2(); #2

        Timer t = new Timer(false);
        t.schedule(mar, 100);
        Timer t2 = new Timer(false);
        t2.schedule(mar2, 5000);

    }

    class MyAnimationRoutine extends TimerTask #3
    {
        MyAnimationRoutine()
        {
        }

        public void run()
        {
            ImageView img = (ImageView)findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
img.getBackground();
            frameAnimation.start(); #3
        }
    }

    class MyAnimationRoutine2 extends TimerTask #4
    {
        MyAnimationRoutine2()
        {
        }
    }
}

```

```

    public void run()
    {
        ImageView img = (ImageView)findViewById(R.id.simple_anim);
        AnimationDrawable frameAnimation = (AnimationDrawable)
img.getBackground();
        frameAnimation.stop();           #4
    }
}
}

```

1. Bind resources to the ImageView.
2. Call the subclasses MyAnimationRoutine and MyAnimatinoRoutine2 which start and stop the Animation respectively.
3. MyAnimationRoutine extends TimerTask to allow for a wait time before it starts the animation.
4. MyAnimationRoutine2 extends TimerTask to allow for a wait time before it stops the animation from running.

Listing 9.11 might be slightly confusing because of the use of the TimerTasks. Since we cannot control the animation from within the OnCreate method we need to create two subclasses which call AnimationDrawable's start and stop method. So the first sub class MyAnimationRoutine extends the TimerTask #3, then calls frameAnimation.start() method for the AnimationDrawable bound to the ImageView background. If you now run the project you should see something like figure 9.3.



Figure 9.3: Making a ball bounce using Android XML animation.

As you can see creating an Animation with XML in Android is pretty simple. You can make the animations reasonably complex as you would with any stop motion type movie but to create more sophisticated animations programmatically we need to move to using Android two and three dimensional graphic abilities and in this next section we will do just that.

9.2.1 Using Android to programmatically create an Animation

In the last section we use Android Frame by Frame animation capabilities to essentially show a series of images in a loop to give the impression of movement. In this next section we are going to animate a globe so that it moves around the screen programmatically.

To do this we are going to animate a graphic file (A PNG) and have it act like it is bouncing around inside our android viewing window. To do this we are going to create a Thread in which our animation will run and a Handler which will help communicate back to our program messages that reflect the changes in state of our animation. This same approach we will use latter in the section on OpenGL ES and you will find it the basic way to approach most complex graphics applications and animations.

9.2.1.1 ANIMATING RESOURCES

In this section we are going to look at a very simple animation technique using an image bound to a sprite and the moving that sprite around the screen to give the appearance of a bouncing ball. To get started first create a new project called bouncing ball with a BounceActivity. You can just cut and past the code in Listing 9.12 for the BounceActivity.java.

Listing 9.12 BounceActivity.java

```
package com.msi.manning.chapter9.bounceyBall;

import android.app.Activity;
import android.os.Bundle;
import android.view.Window;
import android.os.Handler;
import android.os.Message;

public class BounceActivity extends Activity {

    protected static final int GUIUPDATEIDENTIFIER = 0x101;

    Thread myRefreshThread = null;
    BounceView myBounceView = null;

    Handler myGUIUpdateHandler = new Handler() {
        public void handleMessage(Message msg) {
            switch (msg.what) {
                case BounceActivity.GUIUPDATEIDENTIFIER:
                    myBounceView.invalidate();
                    break;
            }
            super.handleMessage(msg);
        }
    };
};
```

```

@Override
public void onCreate(Bundle icle) {
    super.onCreate(icle);
    this.requestWindowFeature(Window.FEATURE_NO_TITLE);

    this.myBounceView = new BounceView(this);           #3
    this setContentView(this.myBounceView);

    new Thread(new RefreshRunner()).start();           #4
}

class RefreshRunner implements Runnable {
    // @Override
    public void run() {                                  #5
        while (!Thread.currentThread().isInterrupted()) {

            Message message = new Message();
            message.what = BounceActivity.GUIUPDATEIDENTIFIER;

            BounceActivity.this.myGUIUpdateHandler.sendMessage(message);

            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

```

1. You need to import the handler and message class which will allow you to communicate messages from the thread to Android asynchronously.
2. Create a handler.
3. Create the view.
4. Create the new thread.
5. Run the animation as long as it is not interrupted otherwise interrupt the thread.

In listing 9.12 the first thing to do is import the Handler and Message #1 classes and then create a Handler #2. A Handler allows you to send and process Messages and Runnable objects associated with a thread's Message Queue. Handlers are associated to a single thread and its message queue and we will use it to allow our objects running a thread to communicate changes in state back to the program that spawned them or vice versa.

Next we setup a view and then create the new thread #4. Finally we create a RefreshRunner inner class implementing Runnable which will run unless something interrupts the thread at which point a message is sent to the Handler to call its invalidate() method.

Now that we need to create the code that will actually do our animation and create View. Before we do this we are going to use an image of a globe which you can obtain here at the

book's web page at Manning.com or any other PNG you would like. We also want to have the Android logo as our background which you can find here (put in link). Just make sure to drop the images under res/drawable/. Next create a new Java file called BounceView and copy and paste the listing in 9.13 into your editor.

Listing 9.13 BounceView.java

```
package com.msi.manning.chapter9.bounceyBall;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Point;
import android.graphics.drawable.Drawable;
import android.view.View;

public class BounceView extends View {

    protected Drawable mySprite;                                #1
    protected Point mySpritePos = new Point(0,0);              #1

    protected enum HorizontalDirection {LEFT, RIGHT}           #2
    protected enum VerticalDirection {UP, DOWN}
    protected HorizontalDirection myXDirection =
HorizontalDirection.RIGHT;
    protected VerticalDirection myYDirection = VerticalDirection.UP;

    public BounceView(Context context) {
        super(context);

        this.setBackground(this.getResources().getDrawable(R.drawable.android));
        this.mySprite = this.getResources().getDrawable(R.drawable.world); #3
    }

    @Override
    protected void onDraw(Canvas canvas) {

        this.mySprite.setBounds(this.mySpritePos.x, this.mySpritePos.y, #4
                                this.mySpritePos.x + 50, this.mySpritePos.y + 50);

        if (mySpritePos.x >= this.getWidth() -
mySprite.getBounds().width()) {                                #5
            this.myXDirection = HorizontalDirection.LEFT;
        } else if (mySpritePos.x <= 0) {
            this.myXDirection = HorizontalDirection.RIGHT;
        }

        if (mySpritePos.y >= this.getHeight() -
mySprite.getBounds().height()) {                                #5
            this.myYDirection = VerticalDirection.UP;
        } else if (mySpritePos.y <= 0) {
            this.myYDirection = VerticalDirection.DOWN;
        }
    }
}
```

```

        if (this.myXDirection == HorizontalDirection.RIGHT) { #6
            this.mySpritePos.x += 10;
        } else {
            this.mySpritePos.x -= 10;
        }

        if (this.myYDirection == VerticalDirection.DOWN) { #6
            this.mySpritePos.y += 10;
        } else {
            this.mySpritePos.y -= 10;
        }

        this.mySprite.draw(canvas); #7
    }
}

```

1. You need to import the handler and message class which will allow you to communicate messages from the thread to Android asynchronously.
2. Create a handler.
3. Get the image file and map it to the sprite. Now when the sprite is drawn it will show the image.
4. Set the bounds of the globe.
5. Move the ball left or right up or down.
6. Check if the ball is trying to leave the screen and then move it appropriately.
7. Draw the globe.

In listing 9.13 we do all the real work of animating our image. First we create a Drawable to hold our globe image and Point which we will use to position and track our globe as we animate it #1. Next we create some enums to hold directional values for horizontal and vertical directions which we will use to keep track of the moving globe #2. Next we map the Globe to the mySprite variable and set the Android logo as the back ground for our animation #3.

Now that we have done the setup work we create a new View and set all the boundaries for the drawable #4. After that we create some simple conditional logic that detects if the globe is trying to leave the screen and if it starts to leave the screen then change its direction #5. Then we provide simple conditional logic to keep the ball moving in the direction it is going if it has not encountered the bounds of the view #6. Finally we actually draw the globe using the draw method. Now if you compile and run the project you should see the globe bouncing around in front of the Android logo like in figure 9.4.



Figure 9.4: A simple animation of a ball bouncing on the screen.

While the simple Animation that we created is not to exciting you could with very little extra work leverage the key concepts here (dealing with boundaries, moving around drawables, detecting changes, dealing with threads, etc) to create something like the google Lunar Lander example game or even a simple version of asteroids. If you want more graphics power and want to easily work with three dimensional objects for creating things like games or sophisticated animations then you should read the next section on OpenGL ES.

9.2.2 Introducing OpenGL for Embedded Systems

One of the most interesting features of Android platform is its support of the OpenGL standard for Embedded Systems or OpenGL ES. OpenGL ES is the embedded systems version of the very popular OpenGL standard which defines a cross platform and cross language API for computer graphics. The OpenGL ES API does not support the full OpenGL API and much of the OpenGL API has been stripped out allow OpenGL ES to run on a large variety of mobile phones, PDA's, video games consoles, and other embedded systems. OpenGL ES was original developed by the Kronos Group, an industry consortium, and the most current version of the standard can be found here <http://www.khronos.org/opengles/>.

OpenGL ES is a fantastic API for 2D and 3D graphics especially for graphically intensive application such as games, graphical simulations and visualizations, and all sorts of animations. Since Android also supports 3D hardware acceleration developers can make graphical intensive applications that target hardware with 3D accelerators.

Because OpenGL and OpenGL ES are such broad topics with whole books dedicated to them we will only cover the basics of working with OpenGL ES and Android. For a much deeper exploration of OpenGL ES you should check out the specification as well as look at the OpenGL ES tutorial here <http://www.zeuscmd.com/tutorials/opengles/index.php>. After reading this section on Android support for OpenGL ES you should have enough information to follow more in depth discussion of OpenGL ES as well as port your code from other languages (such as the tutorial examples) into the Android Framework. If you already know OpenGL or OpenGL ES then the OpenGL commands will be familiar and you should just pay attention to the specifics of working with OpenGL from Android.

NOTE

An excellent book on OpenGL and Java 3-D programming is Java 3D programming by Daniel Selman which can be seen here <http://www.manning.com/selman/>.

With that in mind let us apply the basics of OpenGL ES to first create an OpenGLContext and then create a Window that we could draw on. At a very basic level to use the OpenGL ES with Android you need to:

- First you need to create a custom View subclass.
- Second you need to get a handle to an OpenGLContext, which provides access to Android's OpenGL ES functionality.
- In the View's onDraw() method you will need to use the handle to the GL object and then use its methods to perform any GL functions.

So following the basic steps above lets create a simple example. First we will create a class which follows the above steps to use Android to create a blank surface to draw on. Then in the next section we will dive into using OpenGL ES commands to draw as square and then an animated cube onto the surface. To start lets create a new project called OpenGLSquare and create an activity called OpenGLSquare as in Listing 9.14.

Listing 9.14 OpenGLSquare.java

```
package com.msi.manning.chapter9.GLSquare;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;

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

import android.content.Context;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import javax.microedition.khronos.egl.*;
import javax.microedition.khronos.opengles.*;

#1
```

```

public class SquareActivity extends Activity {

    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(new DrawingSurfaceView(this));
    }

    class DrawingSurfaceView extends SurfaceView implements
    SurfaceHolder.Callback { #2

        public SurfaceHolder mHolder;

        public DrawingThread mThread; #3

        public DrawingSurfaceView(Context c) {
            super(c);
            init();
        }
        public void init() { #4

            mHolder = getHolder();
            mHolder.addCallback(this);
            mHolder.setType(SurfaceHolder.SURFACE_TYPE_GPU);
        }

        public void surfaceCreated(SurfaceHolder holder) { #5
            mThread = new DrawingThread();
            mThread.start();
        }

        public void surfaceDestroyed(SurfaceHolder holder) { #6
            mThread.waitForExit();
            mThread = null;
        }

        public void surfaceChanged(SurfaceHolder holder, int format,
int w, int h) { #7

            mThread.onWindowResize(w, h);
        }

        class DrawingThread extends Thread { #8
            boolean stop;
            int w;
            int h;

            boolean changed = true;

            DrawingThread() {
                super();
                stop = false;
                w = 0;
                h = 0;
            }
        }
    }
}

```

```

@Override
public void run() {

    EGL10 egl = (EGL10)EGLContext.getEGL(); #9
    EGLDisplay dpy =
egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);
    int[] version = new int[2];
    egl.eglInitialize(dpy, version);
    int[] configSpec = { #10
        EGL10.EGL_RED_SIZE, 5,
        EGL10.EGL_GREEN_SIZE, 6,
        EGL10.EGL_BLUE_SIZE, 5,
        EGL10.EGL_DEPTH_SIZE, 16,
        EGL10.EGL_NONE
    };
    EGLConfig[] configs = new EGLConfig[1];
    int[] num_config = new int[1];
    egl.eglChooseConfig(dpy, configSpec, configs, 1,
num_config);

    EGLConfig config = configs[0];

    EGLContext context = egl.eglCreateContext(dpy,
config, EGL10.EGL_NO_CONTEXT, null); #11
    EGLSurface surface = null;
    GL10 gl = null;

    while( ! stop ) { #12
        int W, H; boolean

updated;

        synchronized(this) {
            updated = this.changed;
            W = this.w;
            H = this.h;
            this.changed = false;
        }
        if (updated) {

            if (surface != null) {
                egl.eglMakeCurrent(dpy,
EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
                egl.eglDestroySurface(dpy,
surface);
            }

            surface =
egl.eglCreateWindowSurface(dpy, config,
mHolder, null);
            egl.eglMakeCurrent(dpy, surface,
surface, context);

            gl = (GL10) context.getGL();

            gl.glDisable(GL10.GL_DITHER);

```

```

        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,
                  GL10.GL_FASTEST);

        gl.glClearColor(1, 1, 1, 1);
        gl.glEnable(GL10.GL_CULL_FACE);
        gl.glShadeModel(GL10.GL_SMOOTH);
        gl.glEnable(GL10.GL_DEPTH_TEST);

        gl.glViewport(0, 0, W, H);

        float ratio = (float) W / H;

        gl.glMatrixMode(GL10.GL_PROJECTION);
        gl.glLoadIdentity();
        gl.glFrustumf(-ratio, ratio, -1,
1, 1, 10);
    }

    drawFrame(gl);

    egl.eglSwapBuffers(dpy, surface);

    if (egl.eglGetError() ==
EGL11.EGL_CONTEXT_LOST) {
        Context c = getContext();
        if (c instanceof Activity) {
            ((Activity)c).finish();
        }
    }

    egl.eglMakeCurrent(dpy, EGL10.EGL_NO_SURFACE,
EGL10.EGL_NO_SURFACE,
                        EGL10.EGL_NO_CONTEXT);
    egl.eglDestroySurface(dpy, surface);
    egl.eglDestroyContext(dpy, context);
    egl.eglTerminate(dpy);

}

public void onWindowResize(int w, int h) {
    synchronized(this) {
        this.w = w;
        this.h = h;
        this.changed = true;
    }
}

public void waitForExit() {
    stop = true;
    try {

```

```

        join();
    } catch (InterruptedException ex) {
    }
}

private void drawFrame(GL10 gl) {
    // do whatever drawing here.

}

}
}
}

```

1. Import the needed graphics packages
2. This handles all the creation, destruction, etc. for getting a surface and it does in asynchronously
3. This thread does the actual drawing
4. By registering as a call back, we receive events from the SurfaceHolder notifying us when a new surface is ready, or if the surface we were using was destroyed
5. Create a new thread that will draw to our new surface
6. This function is used to stop our thread when the surface is destroyed
7. This method changes the size of the window if our surface has been changed
8. Create a thread to do the actual drawing of shapes
9. Get an EGL Instance
10. Specify a configuration to use
11. Now obtain a reference to OpenGL ES context created in step 9
12. Do your actual drawing inside the drawFrame method

Now Listing 9.14 will only generate an empty white window. All of the code in listing 9.14 is essentially code you need to draw and manage any OpenGL ES visualization. Walking through Listing 9.14 we first import all our needed classes #1. Second we implement an inner class which will handle everything about managing a surface such as creating it, changing it, or deleting it by extending the class SurfaceView and implementing the SurfaceHolder interface which allows us to get information back from Android when the surface changes, for example someone resizes it #2. With Android all of this has to be done asynchronously and you cannot manage surfaces directly.

Next we create a thread to do the actual drawing #3 and then create an init method which uses the SurfaceView class's getHolder method to get access to the SurfaceView and add a callback to it via the addCallBack method #4. Now that you have added the callback we can implement the surfaceCreated #5, surfaceChanged #6, and surfaceDestroyed #7 which are all methods of the Callback class and are fired on the appropriate condition of change in the Surface's state.

Now that all the Callback methods are implemented we create a thread that will do all our drawing #8. Before we can draw anything though we need to create an OpenGL ES context #9 and then create a handler to the surface #10 so that we can then use the OpenGL context's method to act on the surface via the handle #11. Now we can finally draw something although as you will note that in the drawFrame method #12 we are actually not doing anything. If you ran the code right now all you would get is an empty window but all of the code we have generated so far will appear in some form or another in any OpenGL ES application you make on Android. Typically you would break up your code to have an Activity class to start your code, another class that would implement your custom view, another class might implement your SurfaceHolder and Callback and provide all the method for detecting changes to the surface as well as the actual drawing of your graphics in a thread, and finally whatever code that represents your graphics. In the next section we will look at how to draw a square on the surface as well as an animated cube.

DRAWING SHAPES IN OPENGL ES

In our next example we are going to be using OpenGL ES to create a simple drawing, a rectangle, using OpenGL Primitives which are all essentially pixels, polygons, and triangles. In drawing our square we are going to be using an OpenGL ES Primitive called the GL_Triangle_Strip which simply takes three vertices (or three X, Y, Z) points in an array of vertices and draws a triangle. After that the last two vertices become the first two vertices for the next triangle with the next vertex in the array being the final point. This repeats for as many vertices as there are in the array which generates something like figure 9.4 where you can see one triangle drawn and then the next one.

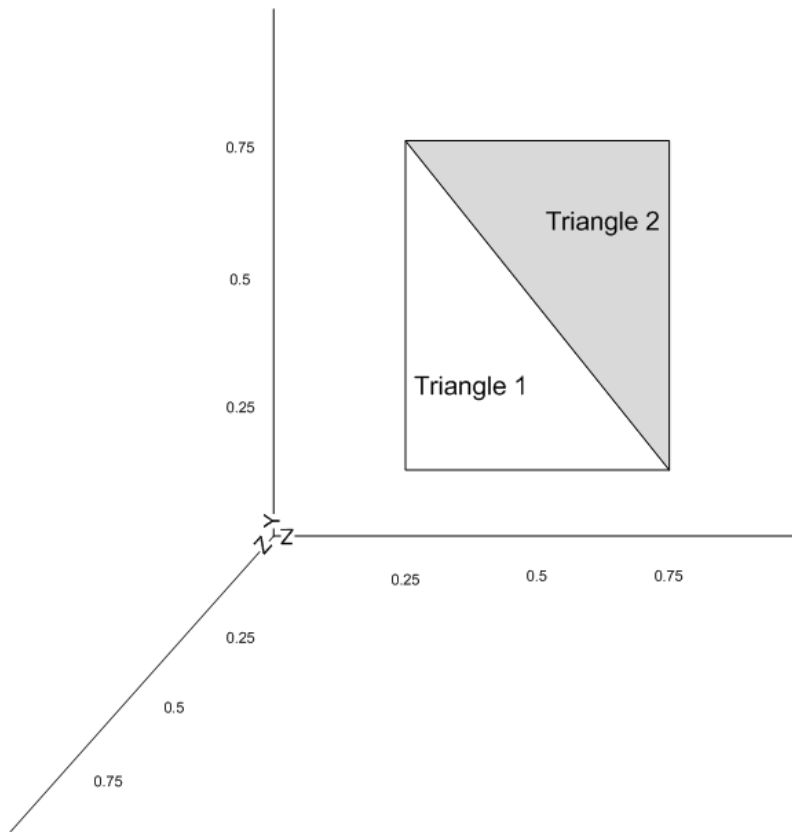


Figure 9.4: How two triangles are drawn from an array of vertices

OpenGL supports a small set of primitives, see table 9.1, from which you can build anything from 3D models of animated characters too simple geometric shapes like a rectangle.

Table 9.1 Open GL Primitives and their descriptions.

Primitive Flag	Description
GL_POINTS	A point is placed at each vertex.
GL_LINES	A line is drawn for every pair of vertices that are given.
GL_LINE_STRIP	A continuous set of lines are drawn. After the first vertex, a line is drawn between every successive vertex and the vertex before it.

GL_LINE_LOOP

This is the same as GL_LINE_STRIP except that the start and end vertices are connected as well.

GL_TRIANGLES

For every triplet of vertices, a triangle is drawn with corners specified by the coordinates of the vertices.

GL_TRIANGLE_STRIP

After the first 2 vertices, every successive vertex uses the previous 2 vertices to draw a triangle.

GL_TRIANGLE_FAN

After the first 2 vertices, every successive vertex uses the previous vertex and the first vertex to draw a triangle. This is used to draw cone-like shapes.

In Listing 9.15 we use array of vertices to define a square we will paint on our surface. To use the code just insert it directly into the code for listing 9.14 right below the commented line that says ' // do whatever drawing here'.

Listing 9.15 code for drawing a square

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT); #1

float[] square = new float[] { #2
    0.25f, 0.25f, 0.0f,
    0.75f, 0.25f, 0.0f,
    0.25f, 0.75f, 0.0f,
    0.75f, 0.75f, 0.0f };

FloatBuffer squareBuff; #3

ByteBuffer bb = #3
ByteBuffer.allocateDirect(square.length*4);
bb.order(ByteOrder.nativeOrder());
squareBuff = bb.asFloatBuffer();
squareBuff.put(square);
squareBuff.position(0);

gl.glMatrixMode(GL10.GL_PROJECTION); #4
gl.glLoadIdentity(); #4
GLU.gluOrtho2D(gl, 0.0f,1.2f,0.0f,1.0f); #5

gl.glVertexPointer(3, GL10.GL_FLOAT, 0, squareBuff); #6
gl.glEnableClientState(GL10.GL_VERTEX_ARRAY); #7

gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
gl.glColor4f(0,1,1,1);
gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4); #8
```

1. Clear the screen. This is usually the first thing you do before drawing anything.

2. Create an array that represents a square.
3. Here we generate a float buffer which will hold our square.
4. These GL commands define the type of projection (Orthographic) and initialize it to the identity matrix.
5. This command setups a two dimensional orthographic viewing region and its clipping points.
6. Set the current vertices for the drawing.
7. Drawing will be performed by a Vertex Array. All arrays are disabled by default so we enable the Vertex array here.
8. Draw our array.

As you can see this code is somewhat dense with OpenGL commands. The first thing we do is clear the screen using `glClear #1` which is something you want to do before every drawing. Then we build the array that will represent the set of vertices that will make up our square #2. As we explained before we will be using the OpenGL primitive `GL_TRIANGLE_STRIP` to create the rectangle like in figure 9.4 where the first set of three vertices is the first triangle and the last vertex represents the third or last vertex in the second triangle which reuses the last two vertices from the first triangle as its first two vertices #2. We then create a buffer to hold that same square data #3. We also tell the system that we will be using a `GL_PROJECTION` for our Matrix Mode which is simply a type of Matrix Transformation that is applied to every point in the Matrix stack #4.

The next things we do are more setup related. First we load the identity matrix and then use the `gluOrtho2D(GL10 gl, float left, float right, float bottom, float top)` command to set the clipping planes that are mapped to the lower left and upper right corners of the window #5. Now we are ready to start drawing our image. To do this we will first use `glVertexPointer(int size, int type, int stride, pointer to array)` method which sets the current vertices to use for the drawing. The method has four attributes size, type, stride, and pointer. Size specifies the number of coordinates per vertex (i.e. a 2D shape might just ignore the z axis), type defines the data type to be used (either `GL_BYTE`, `GL_SHORT`, `GL_FLOAT`, etc) #6, stride specifies the offset between consecutive vertices i.e. how many extra values exist between the end of the current vertex and the beginning of the next vertex, and the pointer is a reference to the array. While most drawing in OpenGL ES is performed by using various forms of Arrays such as the Vertex Array they are all disabled by default to save on system resources. To enable them we use the OpenGL command `glEnableClientState(array type)` which simply accepts a array type which in our case is the `GL_VERTEX_ARRAY` #7.

Finally we use the `glDrawArrays #8` function to render our arrays into the OpenGL primitives and create our simple drawing. The `glDrawArrays(mode, first, count)` has three

attributes. Mode is just what primitive to render such as `GL_TRIANGLE_STRIP`. First is the starting index of the array which we set to 0 since we want it to render all the vertices in the array. Count specifies the number of indices to be rendered and in our case that is 4.

So now if you run the code you should see a simple blue rectangle on a white surface like in Figure 9.5. Not particularly exciting but most of the code we have gone through is you would need for any OpenGL project. Now in our next example we are going to do something a little more exciting which is create a three dimensional cube with different colors on each side and then rotate it in space.

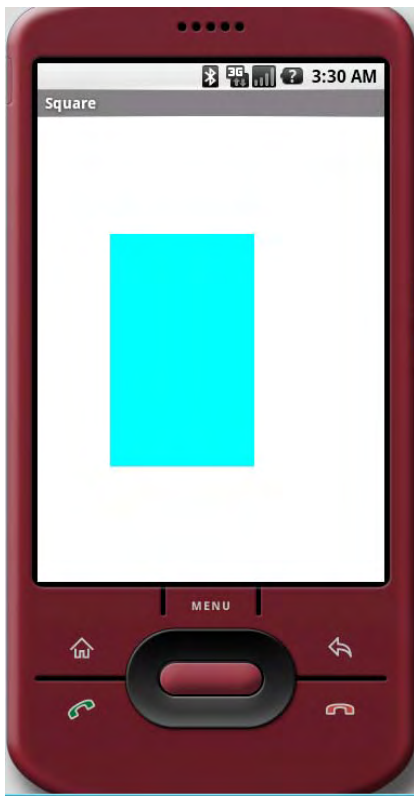


Figure 9.5: A simple square drawn on our surface using OpenGL ES.

THREE DIMENSIONAL SHAPES AND SURFACE WITH OPENGL ES

In this section we are going to use much of the code from the previous example but we are going to extend it to allow us to create a three dimensional cube that rotates. In this section we are going to be looking at not only how to build a more complex shape, a cube, but how to introduce perspective to our graphics to give the illusion to the viewer of depth.

Depth works in OpenGL by using something called a Depth Buffer which contains a depth value between 0 and 1 for every pixel. The value represents the perceived distance between objects and your viewpoint so when two objects depth values are compared the value closer to 0 will appear in front on the screen. To make use of depth in our program we need to first enable the depth bugger by passing the `GL_DEPTH_TEST` to the `glEnable` method. Next you need to use the `glDepthFunc` to define how values are compared. For our example we are going to use `GL_LEQUAL` , see table 9.2, which tells the system to show objects in front of other objects if their depth value is lower.

Table 9.2: Flags for how values in the depth buffer will be compared.

Flag	Description
<code>GL_NEVER</code>	Never passes
<code>GL_LESS</code>	Passes if the incoming depth value is less than the stored value.
<code>GL_EQUAL</code>	Passes if the incoming depth value is equal to the stored value.
<code>GL_LEQUAL</code>	Passes if the incoming depth value is less than or equal to the stored value.
<code>GL_GREATER</code>	Passes if the incoming depth value is greater than the stored value.
<code>GL_NOTEQUAL</code>	Passes if the incoming depth value is not equal to the stored value.
<code>GL_GEQUAL</code>	Passes if the incoming depth value is greater than or equal to the stored value.
<code>GL_ALWAYS</code>	Always passes.

When drawing a primitive, the depth test will take place. If the test passes, the incoming color value will replace the current one.

The default value is `GL_LESS`. We want the test to pass if the values are equal as well. This will cause objects with the same z value to display depending on the order that they were drawn in. We therefore pass `GL_LEQUAL` to the function.

One very important part of maintain the illusion of depth is the need for perspective. In OpenGL a typical perspective is represented by a view point with a near and far clipping plane and a tip, bottom, left, and right planes where objects that are closer to the far plane appear smaller like in figure 9.6.

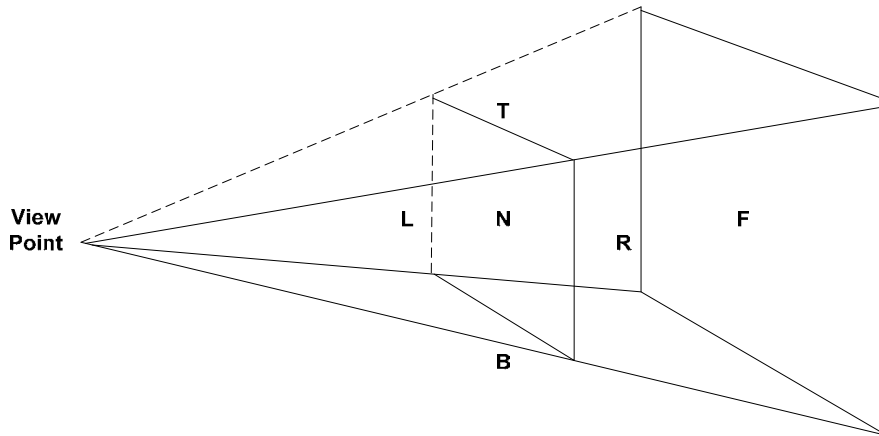


Figure 9.6: In OpenGL a perspective is made up a view point, a near (N) clipping plane, a far (F) clipping plane, Left clipping plane, Right clipping plane, Top and bottom.

OpenGL ES gives us a function called `gluPerspective(GL10 gl, float fovy, float aspect, float zNear, float zFar)` with five parameters (see table 9.3) that allows us to easily create perspective.

Table 9.3: Parameters for the `gluPerspective` function

gl **a GL10 interface**

fovy	field of view angle, in degrees, in the Y direction.
aspect	the aspect ratio that determines the field of view in the x direction. The aspect ratio is the ratio of x (width) to y (height).
zNear	distance from the viewer to the near clipping plane which is always positive.
zFar	the distance from the viewer to the far clipping plane which is always positive.

To demonstrate depth and perspective we are going to create a new project called `OpenGLCube` and just cut and paste the code from Listing 9.4 into the `OpenGLCubeActivity`.

Now add two new variables to your code like in listing 9.6 right at the beginning of the `DrawSurfaceView` inner class.

Listing 9.16 code for drawing a square

```
class DrawingSurfaceView extends SurfaceView implements
SurfaceHolder.Callback {
    public SurfaceHolder mHolder;

    float xrot = 0.0f;           #1
    float yrot = 0.0f;           #1
```

We are going to use `xrot` and `yrot` variables latter in our code to govern the rotation of our cube.

Next right before the `drawFrame` method add a new method called `makeFloatBuffer` like in listing 9.17.

Listing 9.17 code for drawing a square

```
protected FloatBuffer makeFloatBuffer(float[] arr) {
    ByteBuffer bb =
ByteBuffer.allocateDirect(arr.length*4);
    bb.order(ByteOrder.nativeOrder());
    FloatBuffer fb = bb.asFloatBuffer();
    fb.put(arr);
    fb.position(0);
    return fb;
}
```

This float buffer is essentially the same as the one in listing 9.4 but we have just abstracted it from the `drawFrame` method so we can focus on the code for rendering and animating our cube.

Next copy and paste the code in listing 9.18 into the `drawFrame` method.

Listing 9.18 code for drawing a square

```
private void drawFrame(GL10 gl, int w1, int h1) {

    float mycube[] = {                                #1
        // FRONT
        -0.5f, -0.5f,  0.5f,
        0.5f, -0.5f,  0.5f,
        -0.5f,  0.5f,  0.5f,
        0.5f,  0.5f,  0.5f,
        // BACK
        -0.5f, -0.5f, -0.5f,
        -0.5f,  0.5f, -0.5f,
        0.5f, -0.5f, -0.5f,
        0.5f,  0.5f, -0.5f,
        // LEFT
        -0.5f, -0.5f,  0.5f,
        -0.5f,  0.5f,  0.5f,
        -0.5f, -0.5f, -0.5f,
        -0.5f,  0.5f, -0.5f,
        // RIGHT
        0.5f, -0.5f, -0.5f,
        0.5f,  0.5f, -0.5f,
        0.5f, -0.5f,  0.5f,
        0.5f,  0.5f,  0.5f,
        // TOP
        -0.5f,  0.5f,  0.5f,
        0.5f,  0.5f,  0.5f,
        -0.5f,  0.5f, -0.5f,
        0.5f,  0.5f, -0.5f,
        // BOTTOM
```

```

        -0.5f, -0.5f,  0.5f,
        -0.5f, -0.5f, -0.5f,
        0.5f, -0.5f,  0.5f,
        0.5f, -0.5f, -0.5f,
    };

    FloatBuffer cubeBuff;                                #2

    cubeBuff = makeFloatBuffer(mycube);                  #2
    gl.glEnable(GL10.GL_DEPTH_TEST);                     #3
    gl.glEnable(GL10.GL_CULL_FACE);                      #3
    gl.glDepthFunc(GL10.GL_LEQUAL);                      #3
    gl.glClearDepthf(1.0f);

    gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
GL10.GL_DEPTH_BUFFER_BIT);

    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    gl.glViewport(0,0,w1,h1);
    GLU.gluPerspective(gl, 45.0f, ((float)w1)/h1, 1f, 100f);
                                                                #4

    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glLoadIdentity();
    GLU.gluLookAt(gl, 0, 0, 3, 0, 0, 0, 0, 1, 0); #5

    gl.glShadeModel(GL10.GL_SMOOTH);                      #6

    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, cubeBuff);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

    gl.glRotatef(xrot, 1, 0, 0);                          #7
    gl.glRotatef(yrot, 0, 1, 0);                          #7

    gl.glColor4f(1.0f, 0, 0, 1.0f);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4); #8
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 4, 4); #8

    gl.glColor4f(0, 1.0f, 0, 1.0f);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 8, 4);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 12, 4);

    gl.glColor4f(0, 0, 1.0f, 1.0f);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 16, 4);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 20, 4);

    xrot += 1.0f;                                          #9
    yrot += 0.5f;                                          #9

```

1. Create the sides for our cube.
2. Create our float buffer for the cube vertices.

3. Enable the depth test and set the depth test function to GL_EQUAL.
4. Define or perspective.
5. Define your view point in space.
6. Select smooth shading for the model being generated.
7. Rotate by xrot and yrot angle of degrees around vector x,y, z.
8. Draw our sixes sides in three different colors.
9. Increment our x and y rotation angles to keep our cube rotating.

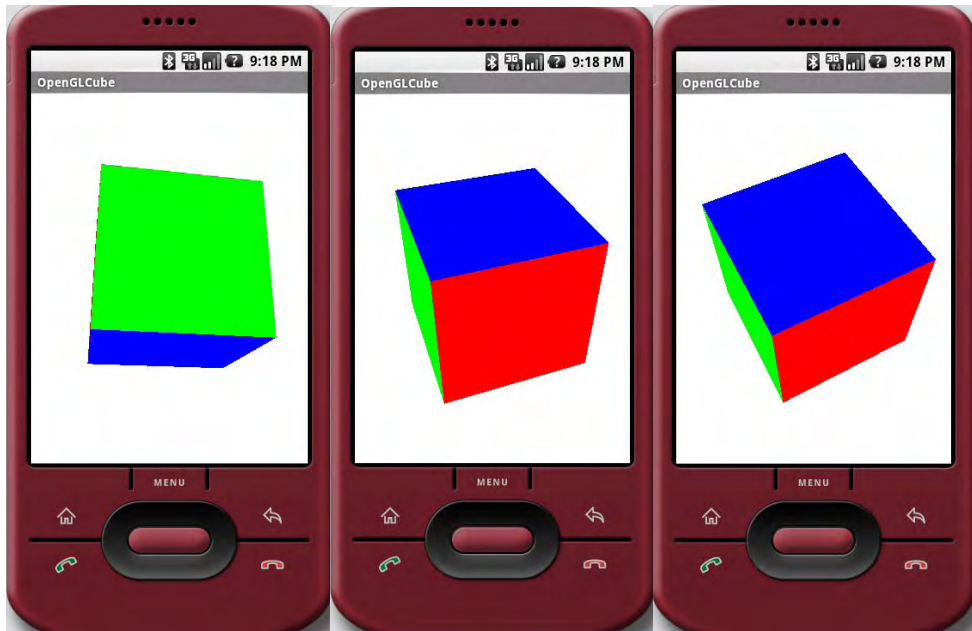


Figure 9.6: A 3D cube rotating in space.

As you can see in code listing 9.7 there is not much new code. The first thing you do is describe the vertices for a cube #1 which is built in the same way as our simple rectangle in listing 9.4 (using triangles). Next we setup the float buffer for our vertices #2 and then enable the depth function #3 and perspective function #4 to provide a sense of depth. One thing to note is that with our `gluPerspective` we passed 45.0 f (45 degrees) to give a more natural view point.

Next we use the `GLU.gluLookAt(GL10 gl, float eyeX, float eyeY, float eyeZ, float centerX, float centerY, float centerZ, float upX, float upY, float upZ)` #5 function to simply move the position of your view without having to modify the actual projection matrix directly. Once we have established our view position we turn on smooth shading for the model #6 and then rotate the cube around the X and Y axes #7. Then we actually draw the cube sides #8 and finally increment the rotation that on the next iteration of draw the cube is drawn at a slightly different angle.

NOTE

You can try experimenting with the fovy value to see how changing the angle affects the display of the cube.

9.3 Summary

In this chapter we have only lightly touched a number of topics related to Android's powerful graphics features from simple drawing, animations, and Android's implementation of the OpenGL ES standard. Graphics and visualizations is a large and complex topic but because Android uses open and well defined standards as well as supports an excellent API for graphics it should be easy for you to use Android's documentation, API, and other resources such as Manning's book on Java 3D Programming by Daniel Sleman to develop anything from a new drawing program to complex games.

In the next chapter we are going to move from graphics to working with multiple media. In chapter 11 we will explore working with audio and video to lay the ground work for making rich multi-media applications.

Chapter 10

Multimedia

Today people use their cell phones for almost everything but phone calls from chat, surfing the web, listening to music, and even watching live streaming TV. Now a day's cell phones need to support multimedia to even be considered as a usable device. In this chapter we are going to look how you can use Android to play audio files, video, take pictures, and even record sound.

Android supports multimedia by making use of the open source multimedia system called OpenCore from Packet Video Corporation. OpenCore provides the foundation for Android's media services which Android wraps in an easy to use API.

In this chapter we will take a quick look at OpenCore's architecture and features and then use it via Android's MediaPlayer API to play audio files, take a picture, play videos, and finally record video and audio from the emulator. To start of this chapter let's first look at OpenCore's multimedia architecture.

10.1 Introduction to Multimedia and OpenCore

Since the foundation of Android's multimedia platform is PacketVideo's OpenCore in this section we will quickly review OpenCore's architecture and services. OpenCore is an Java open source multimedia platform supporting:

- Interfaces for third-party and hardware media codecs, input and output devices and content policies
- Media playback, streaming, downloading and progressive playback—including 3GPP, MPEG-4, AAC and MP3 containers
- Media streaming, downloading and progressive playback—including 3GPP, HTTP and RTSP/RTP
- Video and image encoders and decoders: MPEG-4, H.263 and AVC (H.264), JPEG
- Speech codecs: AMR-NB and AMR-WB

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

- Audio codecs: MP3, AAC, AAC+
- Media recording: 3GPP, MPEG-4 and JPEG
- Video telephony based on 324-M standard
- PV test framework to ensure robustness and stability; profiling tools for memory and CPU usage

OpenCore provides all this functionality in a well laid out set of services which are diagrammed in figure 10.1

NOTE

The current Android SDK does not support video recording via the API. Video recording is still possible, but is specific to the phone vendor.

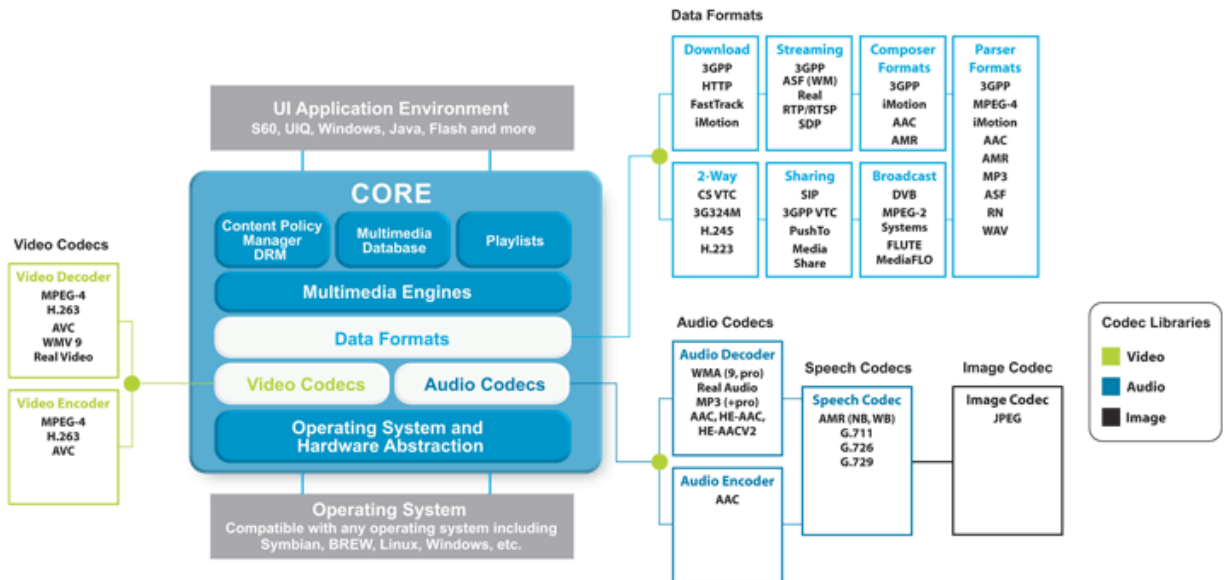


Figure 10.1: Open Core's services and architecture.

As you can see from Figure 10.1 OpenCore's architecture has excellent support for multimedia and numerous CODEC's. In the next section we are going to dive right in and use the Android API to play audio files.

10.2 Playing Audio

Probably the most basic of needs for multimedia on a cell phone is the ability to play audio files be they new ring tones, MP3's, or quick audio notes. Androids media player is pretty easy to use. At a high level all you need to do to play back a MP3 file is:

1. Put the MP3 in the res/raw directory in a project (note you can also use a URI to access files on the network or via the internet)
2. Create a new instance of the MediaPlayer and reference your MP3 by calling MediaPlayer.create()
3. Then call the MediaPlayer methods prepare() and start()

Lets do a simple example to demonstrate exactly how simepl this is. First create a new project called MediaPlayer Example with a activity called MediaPlayerActivity. Now create a new folder under res/ called raw. This is where we will store our MP3's. For this example we will use a ring tone for the game Halo 3 which you can retrieve from <http://halo.msn.com/downloads/na/default.htm>. Download the Hal0 3 Theme song (and any other MP3's) and put them in the raw directory. Next create a simple Button for the music player like in Listing 10.1.

Listing 10.1 main.xml for Media Player example

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

    <Button android:id="@+id/playsong"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Halo 3 Theme Song"
        />
</LinearLayout>
```

Next we just need to fill out our MediaPlayerActivity like Listing 10.2.

Listing 10.2 MediaPlayerActivity.java

```
package com.msi.manning.chapter10.MediaPlayerExample;

import android.app.Activity;
import android.os.Bundle;
import android.media.MediaPlayer;
import android.media.MediaPlayer.OnCompletionListener;
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

import android.view.View;
import android.widget.Button;

public class MediaPlayerActivity extends Activity {
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.main);
        Button mybutton = (Button) findViewById(R.id.playsong);
        mybutton.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
                MediaPlayer mp =
MediaPlayer.create(MediaPlayerActivity.this, R.raw.halotheme);
                mp.start();
                mp.setOnCompletionListener(new OnCompletionListener(){
                    public void onCompletion(MediaPlayer arg0) {
                    }
                });
            }
        });
    }
}

```

0

1 Set the view and a button to play a MP3

2 Get the context and then play the MP3

3 Attempt to catch any IOExceptions and if there is not then start the file3 Detect the completion of the play back

As you can see making playing back a MP3 is trivial. In listing 10.2 all we did was a view that we created in listing 10.1 and map the button, playsong, to mybutton which we then bind to the setOnClickListener() #1. Inside the listener we create the MediaPlayer instance using the create(Context context, int resourceid) method which simply takes our context and a resource id for our MP3. Finally we set the setOnCompletionListener which will on completion perform some task #3. For the moment we do nothing but you might want to change a buttons state or provide a notification to a user that the song is over or play another song and this is the method you would use.

Now if you compile the application and run it you should see something like figure 10.2. Select the button and you should hear the Halo 3 song played back in the emulator via your speakers. You can also control the volume of the playback with the volume switches on the side of the Android emulator phone visualization.



Figure 10.2: A simple red rectangle drawn using Androids graphics API.

Now that we have looked at how to play an audio file let's look at how we can play a video file.

10.3 Playing Video

Playing a video is slightly more complicated than playing audio with the media player in part because you have to provide a view surface for your video to play. Android has a `VideoView` widget that handles that for you and can be used in any layout manager and provides a number of display options including scaling and tinting. So let us get started with playing video by creating a new project called Simple Video Player. Then create a layout like listing 10.3.

NOTE:

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

Currently the emulator has some issues playing video content on some computers and operating systems. Do not be surprised if your audio or video playback is choppy.

Listing 10.3 main.xml – UI for Simple Video Player

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <VideoView android:id="@+id/video"                    #1
        android:layout_width="320px"
        android:layout_height="240px"
    />
    <Button android:id="@+id/playvideo"                  #2
        android:text="Play Video"
        android:layout_height="fill_parent"
        android:layout_width="fill_parent"
        android:paddingRight="4px"
        android:enabled="false"
    />
</LinearLayout>
```

1 Add the Video View widget and set the height and width

2 Add a button to play the video

All we have done in Listing 10.3 is added the VideoView widget and a button to initiate play back of our video.

Next we need to write our class to actually play the video. In addition to the VideoView, we also put in a Button that, when pushed, will pop up the VideoView control panel, known as the MediaController. This, by default, overlays the bottom portion of the VideoView and shows your current position in the video clip, plus offers pause, rewind, and fastforward buttons:

Listing 10.4 SimpleVideo.java

```
package com.msi.manning.chapter10.SimpleVideo;

import android.app.Activity;
import android.os.Bundle;
import android.widget.VideoView;
import android.graphics.PixelFormat;
import android.view.View;
import android.widget.Button;
import android.widget.MediaController;

public class SimpleVideo extends Activity {
    private VideoView myVideo;
    private MediaController mc;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        setContentView(R.layout.main);
        Button bPlayVideo=(Button)findViewById(R.id.playvideo);           #1

        bPlayVideo.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            mc.show();                                                       #2
        }
        myVideo=(VideoView)findViewById(R.id.video);                       #3
        myVideo.setVideoPath("/tmp/test.mp4");                             #4
        mc=new MediaController(this);                                       #5
        mc.setMediaPlayer(myVideo);
        myVideo.setMediaController(mc);                                     #6
        myVideo.requestFocus();
    }
}

```

- 1 Set the view and a button to play a MP4**
- 2 Tell Android to show the MediaController UI widget**
- 3 Use VideoView to provide a container to play the video in**
- 4 Pass the file from the SDCARD to the VideoView for playing**
- 5 Create a MediaController**
- 6 Provide a callback to the VideoView for when the video is done playing.**

In listing 10.4 we have added a button to the VideoView widget #1 and told Android to add a MediaController widget over the VideoView #2 using the show() method. Next we reference the VideoView #3 and use its setVideoPath() #4 to have it look at a Secure Digital Card (sdcard) for our test MP4. Finally we setup the MediaController #5 and use the setMediaController() #6 to perform a call back to the VideoView to notify it when a our video is finished playing. Before we can run this application though we need to learn how to setup a simulated sdcard in the emulator.

10.3.1 Emulating a SDCARD in Android

While Android phone models should offer plenty of device based storage it is more and more common for users to use removable flash based memory or SD cards. For this and several other examples in this chapter, we will make use of emulated SD card. To do this first open up a command prompt so that we can create a image of a SD card for the emulator to use. There are a variety of ways to do this including using the IDE but for the example here we will use the shell. First we need to create our SD image using the mkksdcard tool that ships with the SDK. If you have setup your environment correctly you should be able to navigate to the tools directory of the SDK and call the make image command which will create a FAT32 disc image that we can load into the emulator. The command looks like:

```
mkksdcard [-l label] <size>[K|M] <file>
```


where -l is q volume label for the image and size is the size of the image in kilobytes, flag K, or megabytes, Flag M, respectively. The argument file is path and/or filename for the disk image.

So now that you have your DOS shell open to the tools directory of the SDK type:

```
mksdcard 512M mysdcard
```

Now hit return. A 512 meg FAT32 image named mysdcard has now been created for you to load into the emulator. Now to load the SD card into the emulator we need to use the emulator -sdcard option which accepts an image file (path and filename). For our example we simply need to type:

```
emulator -sdcard mysdcard
```

NOTE

If you set your SD Card to a small number it is possible that Android applications that use the SD card will throw errors because a lack of room.

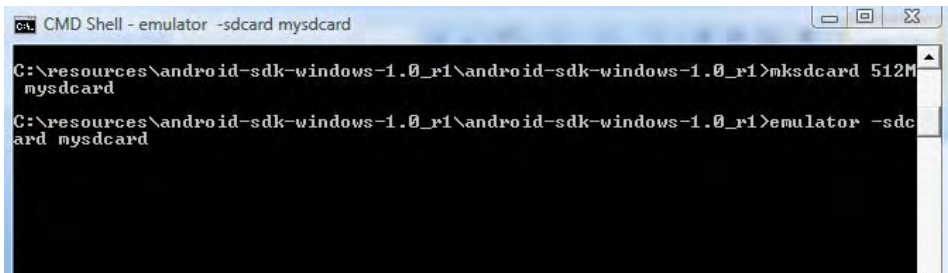


Figure 10.6: example of using the mksdcard tool to create a SD card image for the emulator.

And select return to load the 'mysdcard' image into the emulator. When you do this you should see something like figure 10.6 and the Android emulator should launch. Now we need to push the test.mp4 file that can be found with the source for this chapter at (NOTE: what is the URL for source code?) to the emulator. You can use the adb push command or the IDE. You use the adb command like:

```
Adb push <local file system> <to remote system>
```

For example

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```
Adb push test.mp4 /tmp/test.mp4
```

Once you have pushed the file to the image you can now run the SimpleVideo application by going to your IDE and just running the application while the emulator is already running. You should now see something like figure 10.4.

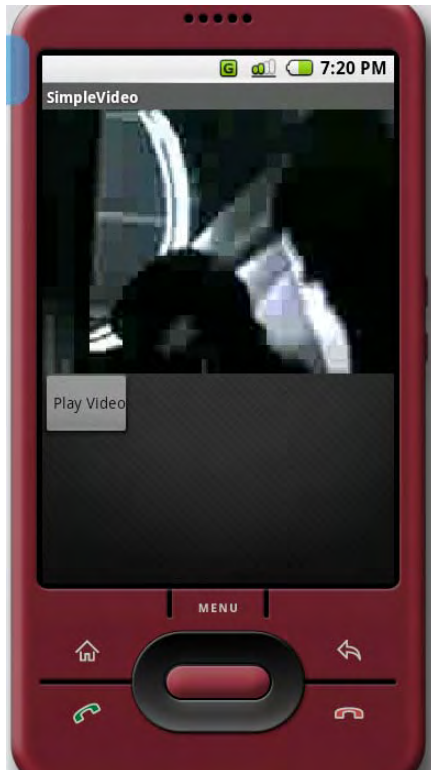


Figure 10.4: Playing a MP4 video in the Android emulator.

As you can see, the VideoView and MediaPlayer classes make working with video files pretty simple. Some things you will need to pay attention to when working with Video files is that the emulator often has problems with files larger than 1 megabyte although the current G1 phone does not.

NOTE

By default, G1 supports only MP4 and 3GP formats. There are several video converters you can use to convert videos in other formats to these standards. As Android adoption

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

grows you can expect latter updates and more players to support a greater number of formats.

Now that we have seen how simple it is to play media using Android's Media Player let us look at how we can use a phones inbuilt camera or microphone to capture images and audio.

10.4 Capturing Media

Using your cell phone to take pictures, record memos, films short videos, etc are all now features expected of any such device. In this section we are going to look at not only how to capture media from the microphone and camera, but we will also write these files to the simulated SD card image we created previously.

To get started lets first look at how to use the Android Camera class to capture images and save them to a file.

10.4.1 Understanding the Camera

A very important feature of modern cell phones is their ability to take pictures or even video using their built in camera. Some phones even support the ability to use the cameras microphone to capture audio. Android of course supports all three features and provides a variety of ways to interact with the camera. In this section we are going to look at how to interact with the camera and take photographs. In the next section we will use the camera to take Video and save it to a Secure Digital card.

For this section we will be creating a new project called 'SimpleCamera to demonstrate how to connect to work with a phone's camera to capture images. For this project we are going to be using the Camera class (<http://code.google.com/android/reference/android/hardware/Camera.html>) to tie the emulator (or phone's) camera to View. Most of the code that we create for this project will really be about showing the input from the camera but the main work for taking a picture is done by a single method called `takePicture(Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.PictureCallback jpeg)` which has three call backs that allow you to control how a picture is taken. Before we get any farther into the Camera class and how to use the camera lets create a project. For this project we will be creating two classes and since the main class is long we will break it up into two sections. For the first section look at Listing 10.5 SimpleCamera.java.

NOTE:

The Android Emulator does not allow you to actually connect to camera devices on your computer like a web cam and thus all your pictures will display a 'chess board' like the one in Figure 10.5. It is possible though to connect to a web camera and get live images and video but it requires some hacking. An excellent example on how to do this can be found at Tom Gibara's web site where he has an open source project for obtain live images from a web cam <http://www.tomgibara.com/android/camera-source>. It is

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

possible that in latter version of the SDK the emulator will support connections to cameras on the hardware the emulator is running on.

Listing 10.5 CameraExample.java

```
package com.msi.manning.chapter10.SimpleCamera;

import java.text.SimpleDateFormat;
import java.util.Date;
import android.app.Activity;
import android.content.ContentValues;
import android.content.Intent;
import android.graphics.PixelFormat;
import android.hardware.Camera;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore.Images.Media;
import android.util.Log;
import android.view.KeyEvent;
import android.view.MenuItem;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

public class SimpleCamera extends Activity implements
SurfaceHolder.Callback
{
    private Camera camera;
    private boolean isPreviewRunning = false;
    private SimpleDateFormat timeStampFormat = new
SimpleDateFormat("yyyyMMddHHmmssSS");

    private SurfaceView surfaceView;
    private SurfaceHolder surfaceHolder;
    private Uri targetResource = Media.EXTERNAL_CONTENT_URI;

    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        Log.e(getClass().getSimpleName(), "onCreate");
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        setContentView(R.layout.main);
        surfaceView = (SurfaceView)findViewById(R.id.surface);
        surfaceHolder = surfaceView.getHolder();
        surfaceHolder.addCallback(this);
        surfaceHolder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);
    }

    public boolean onCreateOptionsMenu(android.view.Menu menu) {
        MenuItem item = menu.add(0, 0, 0, "View Photos?"); #1
        item.setOnMenuItemClickListener(new
MenuItem.OnMenuItemClickListener() {
            public boolean onMenuItemClick(MenuItem item) {
                Intent intent = new Intent(Intent.ACTION_VIEW,
targetResource);
            }
        });
    }
}
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        startActivity(intent);
        return true;
    }
});
return true;
}

@Override
protected void onRestoreInstanceState(Bundle savedInstanceState)
{
    super.onRestoreInstanceState(savedInstanceState);
}

Camera.PictureCallback mPictureCallbackRaw = new
Camera.PictureCallback() { #2
    public void onPictureTaken(byte[] data, Camera c) {
        camera.startPreview();
    }
};

Camera.ShutterCallback mShutterCallback = new Camera.ShutterCallback() #3
{
}
};

```

1. Create a menu option to allow user to see pictures they have taken in Android's Photo Gallery.
2. Create a PictureCallback that provides the raw image data when a picture is taken
3. Create a ShutterCallback that allows you to control the camera shutter behavior such as playing a sound.

If you look at listing 10.5 you will see that it is pretty straight forward. First we set variables for managing a surfaceView and then setup the View. Next we create a simple menu and menu option that will float over our surface when you select the 'MENU' option on the phone while the application is running #1. This menu item will actually open up Android's picture browser and let you view the photos you took. Next we create our first PictureCallback which is called when a picture is first taken #2. This first callback captures uses the PictureCallback's only method onPictureTaken(byte[] data, Camera camera) to grab the raw image data directly from the camera. Next we create a ShutterCallback which can be used with its method onShutter() to play a sound but here we do not call the method #3.

Listing 10.6 CameraExample.java

```

public boolean onKeyDown(int keyCode, KeyEvent event) #1
{
    ImageCaptureCallback camDemo = null;
    if(keyCode == KeyEvent.KEYCODE_DPAD_CENTER) { #2
        try {
            String filename = timeStampFormat.format(new Date());
            ContentValues values = new ContentValues();
            values.put(Media.TITLE, filename);

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        values.put(Media.DESCRPTION, "Image capture by camera");
        Uri uri =
getContentResolver().insert(Media.EXTERNAL_CONTENT_URI, values);
        camDemo = new ImageCaptureCallback(
getContentResolver().openOutputStream(uri));
    } catch(Exception ex ){
    }
    }
    if (keyCode == KeyEvent.KEYCODE_BACK) {
        return super.onKeyDown(keyCode, event);
    }

    if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
        camera.takePicture(mShutterCallback, mPictureCallbackRaw,
camDemo);
        return true;
    }

    return false;
}

protected void onResume()
{
    super.onResume();
}

protected void onSaveInstanceState(Bundle outState)
{
    super.onSaveInstanceState(outState);
}

protected void onStop()
{
    super.onStop();
}
public void surfaceChanged(SurfaceHolder holder, int format, int w, int
h)
{
    if (isPreviewRunning) {
        camera.stopPreview();
    }
    Camera.Parameters p = camera.getParameters();
    p.setPreviewSize(w, h);
    camera.setParameters(p);
    camera.setPreviewDisplay(holder);
    camera.startPreview();
    isPreviewRunning = true;
}
public void surfaceCreated(SurfaceHolder holder)
{
    camera = Camera.open();
}

public void surfaceDestroyed(SurfaceHolder holder)
{
    camera.stopPreview();
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        isPreviewRunning = false;
        camera.release();
    }
}

```

1. Create a method to detect key events.
2. Check to see if the center key was depressed and write a file to the sdcard
3. Check to see if the center key was depressed and if so take a picture

Listing 10.6 is a little more complicated than 10.5 although a large amount of the code in this listing is really about managing the surface for the camera preview but as you can see the very first line is the start of a implementation of a method `onKeyDown` #1 that checks to see if the center key on the dpad was depressed. If it was depressed we setup the creation of a file and by using the `ImageCaptureCallback`, which we will define in listing 10.7, we create a `OutputStream` to write our image data to #2 including not only the image but the file name and other metadata. Next we actually call the method `takePicture()` and pass it the three callbacks `mShutterCallback`, `mPictureCallbackRaw`, and `camDemo` where `mPictureCallbackRaw` is our raw image, and `camDemo` actually writes the image to a file on the sdcard as you can see in Listing 10.7.

Listing 10.7 ImageCaptureCallback.java continued

```

package com.msi.manning.chapter10.SimpleCamera;

import java.io.OutputStream;
import android.hardware.Camera;
import android.hardware.Camera.PictureCallback;

public class ImageCaptureCallback implements PictureCallback {

    private OutputStream filoutputStream;

    public ImageCaptureCallback(OutputStream filoutputStream) {
        this.filoutputStream = filoutputStream;    #1
    }

    public void onPictureTaken(byte[] data, Camera camera) {
        try {
            filoutputStream.write(data);
            filoutputStream.flush();
            filoutputStream.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

In Listing 10.7 the class implements the `PictureCallback` interface and provides two methods. The constructor simple create a stream to write data two #1 and the second method `onPictureTaken` takes binary data and writes to the sdcard as a JPEG. Now if you build this project and start the emulator running using the sdcard image from previously in

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

this chapter you should see something like figure 10.5 when you start the SimpleCamera application from the Android menu. If you look at figure 10.5 you will notice an odd black and white checked background with a bouncing grey box. This is a test pattern that the Android emulator generates to simulate a image feed since the emulator is not actually pulling a live feed from the camera.

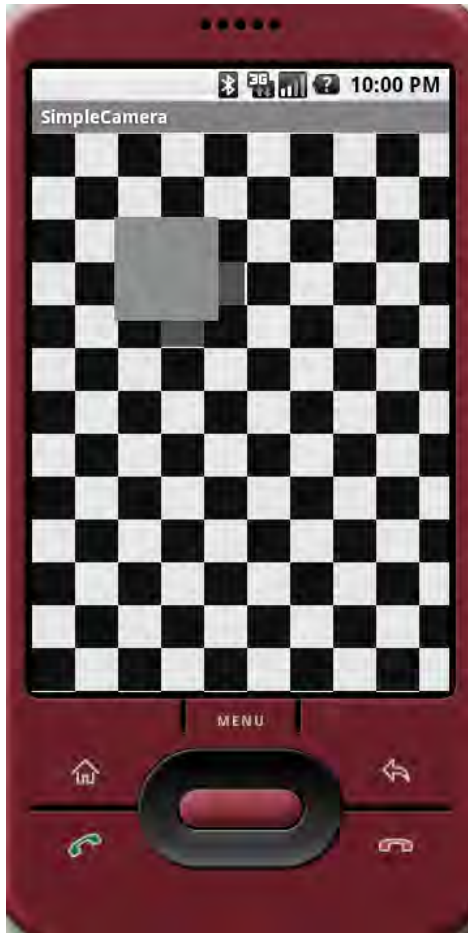


Figure 10.5: Test pattern coming from the emulator camera and being displayed in the SimpleCamera application.

Now if you depress the center button on the dpad in the emulator the application will take a picture. To see the picture simply depress the 'MENU' button which will cause a menu to

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

float on to the camera view window that has a single option 'View Pictures'. If you select view pictures you will be taken to the Android picture explorer and you should see Androids image place holders representing the number of camera captures. You can also see the JPEG files where written to the sdcard by opening up the DDMS in Eclipse and navigating to sdcard>dcim>Camera. You can see a example of what this might look like in figure 10.6.

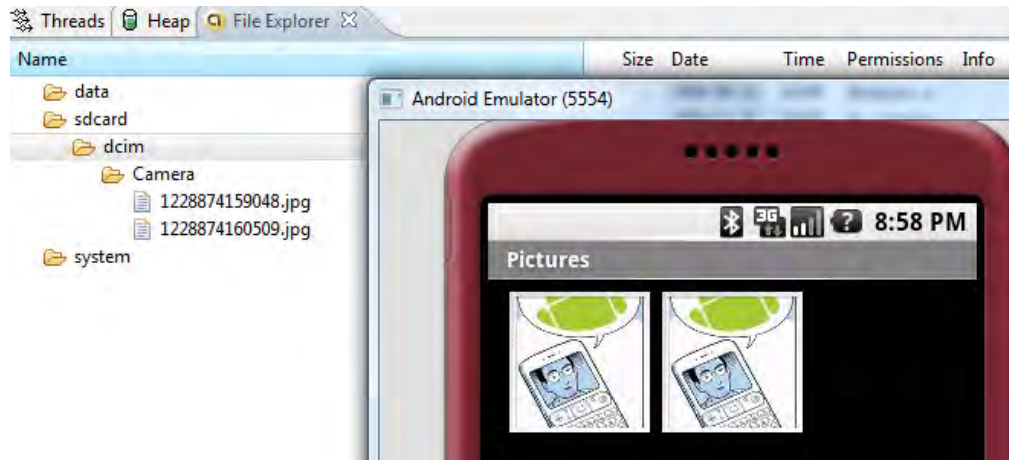


Figure 10.6: The Android emulator shows place holder images for each photo taken.

As you can see working with the camera in Android is not particularly complicated although to see how a real camera will behave you will have to test on a real handset until the emulator provides a simple way to connect to cameras on your computer. This should not stop you though from developing your camera applications and already there are a wealth of Android applications that make sophisticated use of the Camera from games to a application that use a picture of your face to unlock your phone. Now that you have taken a look at how the camera class works in Android lets look at how to capture or record Audio from a camera's microphone. In the next section we will do just that by making use of the MediaRecorder class and writing recordings to a sdcard.

10.4.2 Capturing Audio

Now that we have an idea of how the camera works in Android let us look at how we can use the onboard microphone to record audio. In this section we are actually going to make use of the Android MediaRecorder example from Google Android Developers list which you can find here <http://groups.google.com/group/android-developers/files> although the code shown here has been slightly updated.

NOTE

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

As of the time of writing the Google Android SDK 1 does not allow you to capture audio from the emulator via your computer but it is likely that latter versions of the SDK will.

In general recording either media follows the same process in Android, which is:

1. Create an instance of `android.media.MediaRecorder`.
2. Create an instance of `android.content.ContentValues` then add properties like `TITLE`, `TIMESTAMP`, and the all important `MIME_TYPE`.
3. Create a file path for the data to go to using `android.content.ContentResolver`
4. To set a Preview display on a view surface in use `MediaRecorder.setPreviewDisplay()`.
5. Set the audio source using `MediaRecorder.setAudioSource()`.
6. Set output file format using `MediaRecorder.setOutputFormat()`
7. Set your audio encoding using `MediaRecorder.setAudioEncoder()`
8. Use `prepare()` and `start()` to prepare and start your recordings.
9. Use `stop()` and `release()` to gracefully stop and clean up your recording process.

While recording media is not especially complex it is, as you can see, more complex than playing it. To really understand how to use the `MediaRecorder` let us look at an application. Go ahead and create a new application called 'SoundRecordingDemo'. Next you need to edit the `AndroidManifest.xml` and add the following line:

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

Which will allow the application to actually record the audio files and play them. Next create a class like listing 10.8.

Listing 10.8 SoundRecordingdemo.java continued

```
package com.msi.manning.chapter10.SoundRecordingDemo;

import android.app.Activity;
import android.content.ContentResolver;
import android.content.ContentValues;
import android.content.Intent;
import android.media.MediaRecorder;
import android.net.Uri;
import android.os.Bundle;
import android.os.Environment;
import android.provider.MediaStore;
import android.util.Log;
import android.view.View;
import android.widget.Button;

import java.io.File;
import java.io.IOException;

public class SoundRecordingDemo extends Activity {
```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

MediaRecorder mRecorder;
File mSampleFile = null;
static final String SAMPLE_PREFIX = "recording";
static final String SAMPLE_EXTENSION = ".mp3";

private static final String TAG="SoundRecordingDemo";

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    mRecorder = new MediaRecorder();

    Button startRecording = (Button)findViewById(R.id.startrecording);
    Button stopRecording = (Button)findViewById(R.id.stoprecording);

    startRecording.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
            startRecording();
        }
    });

    stopRecording.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v) {
            stopRecording();
            addToDB();
        }
    });
}

protected void addToDB() {
    ContentValues values = new ContentValues(3);
    long current = System.currentTimeMillis();

    values.put(MediaStore.Audio.Media.TITLE, "test_audio");           #1
    values.put(MediaStore.Audio.Media.DATE_ADDED, (int) (current /   #1
1000));
    values.put(MediaStore.Audio.Media.MIME_TYPE, "audio/mp3");       #1
    values.put(MediaStore.Audio.Media.DATA,
mSampleFile.getAbsolutePath());

    ContentResolver contentResolver = getContentResolver();

    Uri base = MediaStore.Audio.Media.EXTERNAL_CONTENT_URI;
    Uri newUri = contentResolver.insert(base, values);
    sendBroadcast(new Intent(Intent.ACTION_MEDIA_SCANNER_SCAN_FILE,
newUri)); #2
}

protected void startRecording() { #3
    mRecorder = new MediaRecorder();
    mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
    mRecorder.setOutputFormat(MediaRecorder.OutputFormat.THREE_GPP);
    mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
    mRecorder.setOutputFile(mSampleFile.getAbsolutePath());
}

```

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

        mRecorder.prepare();
        mRecorder.start();

        if (mSampleFile == null) {
            File sampleDir = Environment.getExternalStorageDirectory();

            try { mSampleFile = File.createTempFile(SAMPLE_PREFIX,
SAMPLE_EXTENSION, sampleDir);
            } catch (IOException e) {
                Log.e(TAG,"sdcard access error");
                return;
            }
        }

        protected void stopRecording() {
            mRecorder.stop();
            mRecorder.release();
        }
    }
}

```

#4

1. **Set the metadata for the audio file including the title and type of Audio file**
2. **Notify the music player that a new audio file has been created.**
3. **Start the recording of the file, define the source, define the format, set the encoder, and output the file.**
4. **Stop recording and release the MediaRecorder.**

As you can see listing 10.8 the first part of the code is just creating the buttons and button listeners to start and stop the recording. The first part of the listing you need to pay attention to is the addToDB() method. In this method we set all the metadata for the audio file we plan to save including the title, date, and type of file #1. Next we call the Intent ACTION_MEDIA_SCANNER_SCAN_FILE to notify applications like Android's Music player that a new Audio file has been created #2. This will allow us to use the Music player to look for new files in a playlist and play the file.

Now that we have finished the addToDB method we create the startRecording method which creates a new MediaRecorder and just like in the steps in the beginning of this section we set a audio source which is the microphone, we set a output format as THREE_GPP, we set the audio encoder type to AMR_NB, and then set the output file path to write the file. Next we use the methods pre pate() and start() to actually enable the recording of audio.

Finally we create the stopRecording() method to stop the MediaRecorder saving audio. We simply do this by using the methods stop() and release(). Now if you build this application and run the emulator with the sdcard image from the previous section you should be able to launch the application in from Eclipse and select the "Start Recording" button. After a few seconds press the "Stop Recording" button and if you open the DDMS you should be able to navigate to the sdcard folder and see your recordings like in figure 10.7.

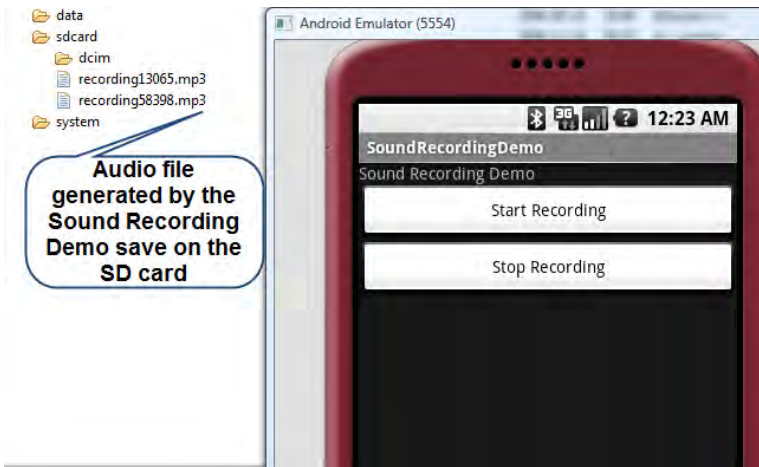


Figure 10.7: A example of audio files being saved to the SD card image in the emulator.

If you actually have music playing on your computers audio system the Android Emulator will pick it up and record it directly from the audio buffer (since it is not actually recording from a microphone) and record it. You can then easily test this by open up the Android Music player and selecting Playlists>Recently Added and it should play your recorded file and you should be able to hear anything that was playing on your computer at the time.

While Android currently only lets you record audio Google plans to soon add support for recoding video which will also make use of the MediaRecorder to allow you to record video coming in from the camera much like you would audio.

10.5 Summary

In this chapter we looked at how the Android SDK makes use of multimedia and how you can play, save, and record Video and Sound. We also looked at various features the Android Media player offers the developer from a built in VideoPlayer to a wide support for formats, encodings, and standards.

We also look at how to interact with other hardware devices attached to the phone such as a microphone and the camera. We also used the SDK to create a SD card image for the emulator to simulate SD cards and used the MediaRecorder application to record audio and save it to the SD card.

While Androids' SDK and emulator, at the time of writing, do not provide a good way to interact with a web cam or microphone on your development platform you can create real multimedia applications using the SDK now as some vendors already have on their phone platforms. Google Android currently offers you everything you need to create rich and compelling media applications and their focus on supporting industry and open standards guarantees your applications will have wide support on a variety of phones.

©Manning Publications Co. Please post comments or corrections to the Author Online forum:

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>

Licensed to Thow Way Chiam <ken.ctw@gmail.com>

In the next chapter you will learn all about how to use Android's location services to interact with GPS and Maps. By mixing what you have learned in this chapter you could even create your open GPS application that not only provides voice direction but could respond to voice commands.

Location, Location, Location

In this chapter,

- Manipulating location properties in the emulator
- Working with `LocationProvider` and `LocationManager`
- Implementing and registering `LocationListeners`
- Understanding `MapActivity` and `MapView`
- Using the Geocoder

A mobile device with accurate location awareness is very powerful. Combining location awareness with network data access is world changing – and this is where Android shines. Android isn't the only platform to support this capability, of course, but its easy to work with location API framework and open source nature does distinguish it somewhat.

From direct network queries, to triangulation with cell towers, and even the Global Positioning System (GPS), an Android powered device has access to several different `LocationProvider` instances that it can utilize to access location data. Different providers supply a varied mix of location related metrics including: latitude and longitude, speed, bearing, and altitude.

GPS is the most common location provider you will work with on the Android platform, because it is the most accurate and powerful option. Nevertheless, some devices may either not have a GPS receiver, or for one reason or another a GPS signal may not be available. In those instances the Android platform provides the capability for you to fail gracefully - to query other providers when your first choice fails. You can configure which providers are available, and hook into one or another through the `LocationManager` class.

Location awareness opens up a new world of possibilities for application development. We are just beginning to see what inventive developers can do with real time location information and faster and more reliable network data access. In this chapter we are going to build an application that combines location awareness with data from the United States' National Oceanic and Atmospheric Administration (NOAA).

Specifically we will be connecting to the National Data Buoy Center (NDBC) to retrieve data from buoys that are positioned around the coastline in North America (and a few NOAA ships). That's right, we said "data from buoys." Thanks to the NOAA-NDBC system, which polls sensors on buoys and makes that data available in RSS feeds, we can retrieve data for the vicinity, based on the current location, and display condition information such as wind speed, wave height, and temperature to our users

(though we won't cover non location related details in this chapter, such as using HTTP to pull the RSS feed data, the full source code for the application is available with the code download for this chapter). This application, which we are calling "Wind and Waves," has several main screens including an Android `MapActivity` with a `MapView`. These components are used for displaying and manipulating map information, as seen in figure 11.1.



Figure 11.1 Several screens from the Wind and Waves location aware application.

We admit that accessing buoy data has a somewhat limited audience - being important mainly for marine use cases (and in this case only working for fixed buoys in North America, and several ships that can be used as worldwide data points) - but we wanted to try to demonstrate the broad scope of possibility here, to come up with something unique. Along with being unique we also hope to make this an interesting application that exercises a good deal of the Android location related capabilities.

In addition to displaying data based on the current location we will also use this application to create several `LocationListener` instances so that we can receive updates when the user's location changes. When the location changes, and the device lets our application know, we will update our `MapView` using an `Overlay` - an object that allows us to draw on top of the map.

Outside of what our buoy application requires, here we will also pull in a few samples for working with the `Geocoder` class. This class allows you to map between a `GeoPoint` (latitude and longitude) and a place (city or postal code) or address. This is a very helpful utility, so we will cover it even though we won't be using it while on the high seas.

Before we begin building any of our example code, we will start with the basics using the built in mapping application and simulating our position within the emulator. This will allow us to mock our location for the emulator. Once we have that covered we will then move on to building Wind and Waves.

11.1 Simulating your location within the emulator

For any location aware application you will start by working with the provided SDK and the emulator. The first thing you will want to do within the emulator is set, and then update, your current location. From there you will want to progress to supplying a range of locations and times to simulate movement over a geographic area.

There are several ways you can accomplish these tasks for the emulator, either by using the Dalvik Debug Monitor Service (DDMS) tool, or by using the command line within the shell. The fastest way to get started is to send in direct coordinates through the DDMS tool.

11.1.1 Sending in your coordinates with the DDMS tool

The DDMS tool is available in two contexts, either launched on its own from the SDK "tools" subdirectory, or as the "Emulator Control" view within the Eclipse IDE (you need to have Eclipse and the Android Eclipse plugin to use DDMS within Eclipse, see chapter 2 for more details about getting the SDK and plugin setup).

The simplest way to set your location with the DDMS tool is to send direct latitude and longitude coordinates manually from the Emulator Control "Location Controls" form. This is depicted, using the straightforward "manual" approach, in figure 11.2 (note that Longitude is the top/first field, which is backwards in terms of how latitude and longitude are generally expressed).

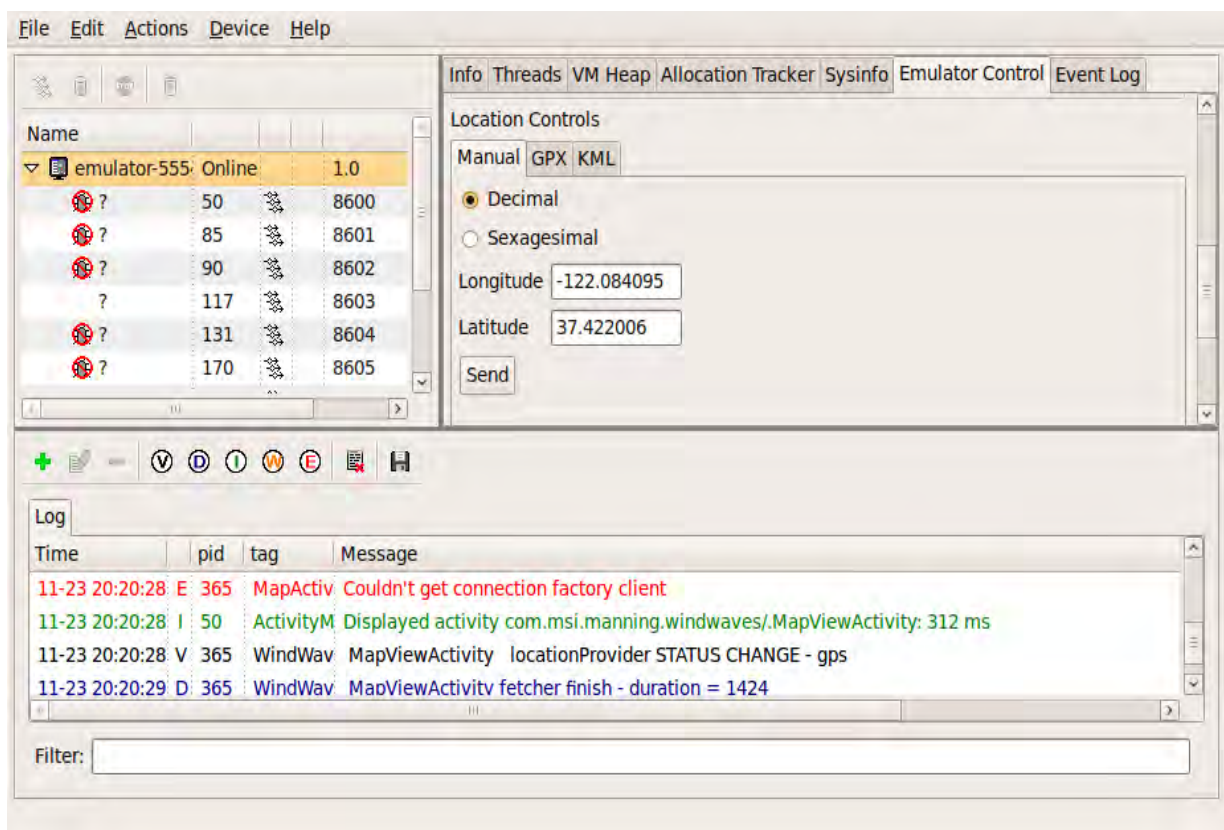


Figure 11.2 Using the DDMS tool to manual form to send direct latitude and longitude coordinates to the emulator as a mock location.

If you launch the built in Maps application, and then send in a location with the DDMS tool, you should then be able to use the menu to select “My Location” and the map will animate to the location you have specified - anywhere on the Earth.

Try this a few times to make sure you get the hang of it, for example send the decimal coordinates in table 11.1 one by one, and in between browse around with the built in map. When you supply coordinates to the emulator you will need to use the decimal form.

Table 11.1 Example coordinates for the emulator to set using the DDMS tool.

Description	Latitude Degrees	Longitude Degrees	Latitude Decimal	Longitude Decimal
Golden Gate Bridge, California	37°49' N	122°29' W	37.49	-122.29
Mount Everest, Nepal	27°59' N	86°56' E	27.59	86.56
Ayer's Rock, Australia	25°23' S	131°05' E	-25.23	131.05
North Pole	90°00' N	-	90.00	-
South Pole	90°00' S	-	-90.00	-

Though the DDMS tool requires the decimal form, longitude and latitude are more commonly expressed on maps and other tools as degrees, minutes, and seconds. Degrees are used because these coordinates represent points on the surface of the globe as measured from either the Equator (for latitude) or the Prime Meridian (for longitude). Each degree is further subdivided into 60 smaller sections, called minutes, and each minute also has 60 seconds (and it goes on from there if need be, tenths of a second, etc).

When representing latitude and longitude on a computer, the degrees are usually converted into decimal form with positive representing North and East, and negative representing South and West. (It's not personal, but if you live in the Southern and Eastern hemispheres, say in Buenos Aires, Argentina, which is 34°60' S, 58°40' W in the degree form, the decimal form is negative for both latitude and longitude, -34.60, -58.40.) If you haven't used latitude and longitude much before the different forms can be confusing at first, but they quickly become second nature after you work with them a bit.

Using the command line to send coordinates

You can also send direct coordinates from within the emulator console. If you "telnet localhost 5554" you will connect to the default emulator's console (adjust the port where necessary). From there you can use the `geo fix` command to send longitude, latitude, and optional altitude. For example: `geo fix -21.55 64.1`. Again keep in mind that the Android tools require that longitude be the first parameter, and that can be counterintuitive.

Once you have mastered setting a fixed position, the next thing you will want to be able to do is supply a set of coordinates that the emulator uses to simulate a range of movement.

11.1.2 The GPS Exchange Format (GPX)

The DDMS tool supports two formats for supplying a range of location data in file form to the emulator. The GPS Exchange Format (GPX) is the first of these and is the more expressive form in terms of working with Android.

GPX is an XML Schema (<http://www.topografix.com/GPX/1/1/>) that allows you to store waypoints, tracks, and routes. Many handheld GPS devices support and or utilize this format. Listing 11.1 is a portion of an example GPX file which shows the basics of the format.

Listing 11.1 A sample GPX file that can be used to send mock location information to the Android emulator through the DDMS tool.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1" version="1.1"                #1
  creator="Charlie Collins - Hand Rolled"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.topografix.com/GPX/1/1
    http://www.topografix.com/GPX/1/1/gpx.xsd">

  <metadata>                                                                #2
    <name>Sample Costal California Waypoints</name>
    <desc>Test waypoints for use with Android</desc>
    <time>2008-11-25T06:52:56Z</time>
    <bounds minlat="25.00" maxlat="75.00" minlon="100.00" maxlon="-150.00" />
  </metadata>

  <wpt lat="41.85" lon="-124.38">                                          #3
    <ele>0</ele>
    <name>Station 46027</name>
    <desc>Off the coast of Lake Earl</desc>
  </wpt>
  <wpt lat="41.74" lon="-124.18">
    <ele>0</ele>
    <name>Station CECC1</name>
    <desc>Crescent City</desc>
  </wpt>
  <wpt lat="38.95" lon="-123.74">
    <ele>0</ele>
```

```

        <name>Station PTAC1</name>
        <desc>Point Arena Lighthouse</desc>
    </wpt>

    . . . remainder of wpts omitted for brevity

    <trk>
        <name>Example Track</name>
        <desc>A fine track with trkpt's.</desc>
        <trkseg>
            <trkpt lat="41.85" lon="-124.38">
                <ele>0</ele>
                <time>2008-10-15T06:00:00Z</time>
            </trkpt>
            <trkpt lat="41.74" lon="-124.18">
                <ele>0</ele>
                <time>2008-10-15T06:01:00Z</time>
            </trkpt>
            <trkpt lat="38.95" lon="-123.74">
                <ele>0</ele>
                <time>2008-10-15T06:02:00Z</time>
            </trkpt>

            . . . remainder of trkpts omitted for brevity

        </trkseg>
    </trk>
</gpx>

```

#4
#5
#6

1. The root gpx element with namespace declarations
2. Including a stanza for metadata
3. Supplying individual waypoint elements - wpt
4. Supplying a track within the trk element
5. Using a track segment
6. Providing specific points within the track with trkpt

A GPX file requires the correct XML namespace **#1**, and then moves on to metadata **#2**, and individual waypoints **#3** (waypoints are named locations, and are defined using latitude and longitude). Along with individual waypoints a GPX file also supports related route information in the form of tracks **#4**, which can be subdivided further into track segments **#5**. Each track segment is made up of track points (which are basically related and ordered waypoints with an additional point in time property).

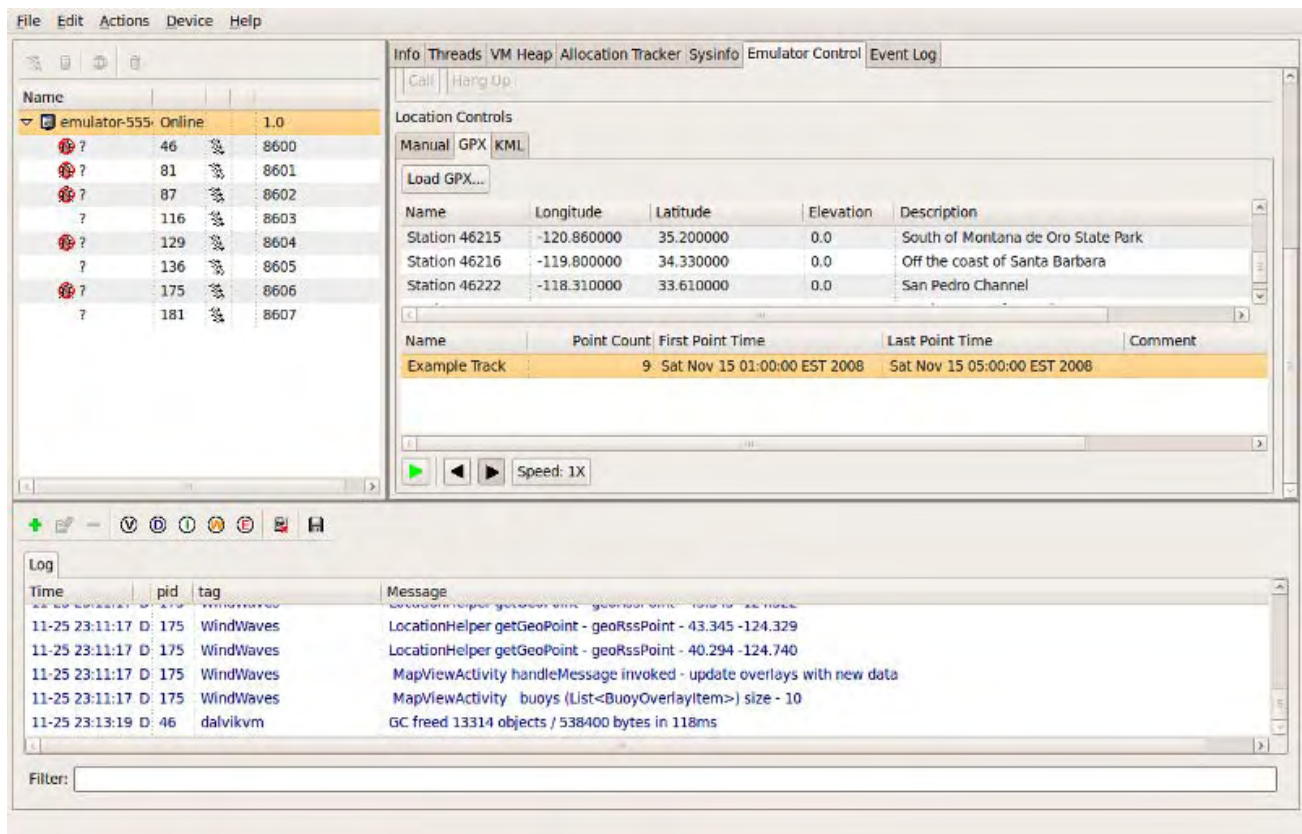


Figure 11.3 Using the DDMS tool with a GPX file to send mock location information.

When working with a GPX file in the DDMS tool you can use two different modes, as the screen shot in figure 11.3 reveals. In the top half of the GPX box individual waypoints are listed, as each is clicked that individual location is sent to the emulator. In the bottom half of the GPX box all the tracks are displayed. Tracks can be “played” forward and backward to simulate movement. As each track point is reached in the file, based on the time it defines (the times matter with GPX - and can be sped up or slowed down using the speed button) those coordinates are sent to the emulator.

GPX is very simple and extremely useful when working with mock location information for your Android applications, but it's not the only file format supported. The DDMS tool also supports a format called KML.

11.1.3 The Google Earth Keyhole Markup Language (KML)

The second format that the Android DDMS tool supports for sending a range of mock location information to the emulator is the Keyhole Markup Language (KML). KML was originally a proprietary format (created by Keyhole, which was acquired by Google), but has since been submitted to the Open Geospatial Consortium (OGC), and accepted as an international standard.

The main goal of the OGC KML is stated as:

That there be one international standard language for expressing geographic annotation and visualization on existing or future web-based online and mobile maps (2d) and earth browsers (3d).

A sample KML file for sending location data to the Android emulator is shown in listing 11.2. This file uses the same coastal location data as we saw with the previous GPX example.

Listing 11.2 A sample KML file that can be used to send mock location information to the Android emulator through the DDMS tool.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">                                #1

  <Placemark>                                                                #2
    <name>Station 46027</name>
    <description>Off the coast of Lake Earl</description>
    <Point>                                                                #3
      <coordinates>-124.38,41.85,0</coordinates>                            #4
    </Point>
  </Placemark>

  <Placemark>
    <name>Station 46020</name>
    <description>Outside the Golden Gate</description>
    <Point>
      <coordinates>-122.83,37.75,0</coordinates>
    </Point>
  </Placemark>

  <Placemark>
    <name>Station 46222</name>
    <description>San Pedro Channel</description>
    <Point>
      <coordinates>-118.31,33.61,0</coordinates>
    </Point>
  </Placemark>

</kml>
```

1. The root kml element with namespace declaration
2. Capturing location information within a Placemark
3. Using a Point for inside a Placemark
4. Supplying coordinates for a Point

KML uses a `kml` root element and again, like any self respecting XML format, requires the correct namespace declaration **#1**. KML supports many more elements and attributes than the DDMS tool is concerned with parsing. Basically in DDMS terms, all your KML files needs to have are `Placemark` elements **#2**, which contain `Point` child elements **#3** that in turn supply `coordinates` **#4**.

Figure 11.4 shows an example of using a KML file with the DDMS tool.

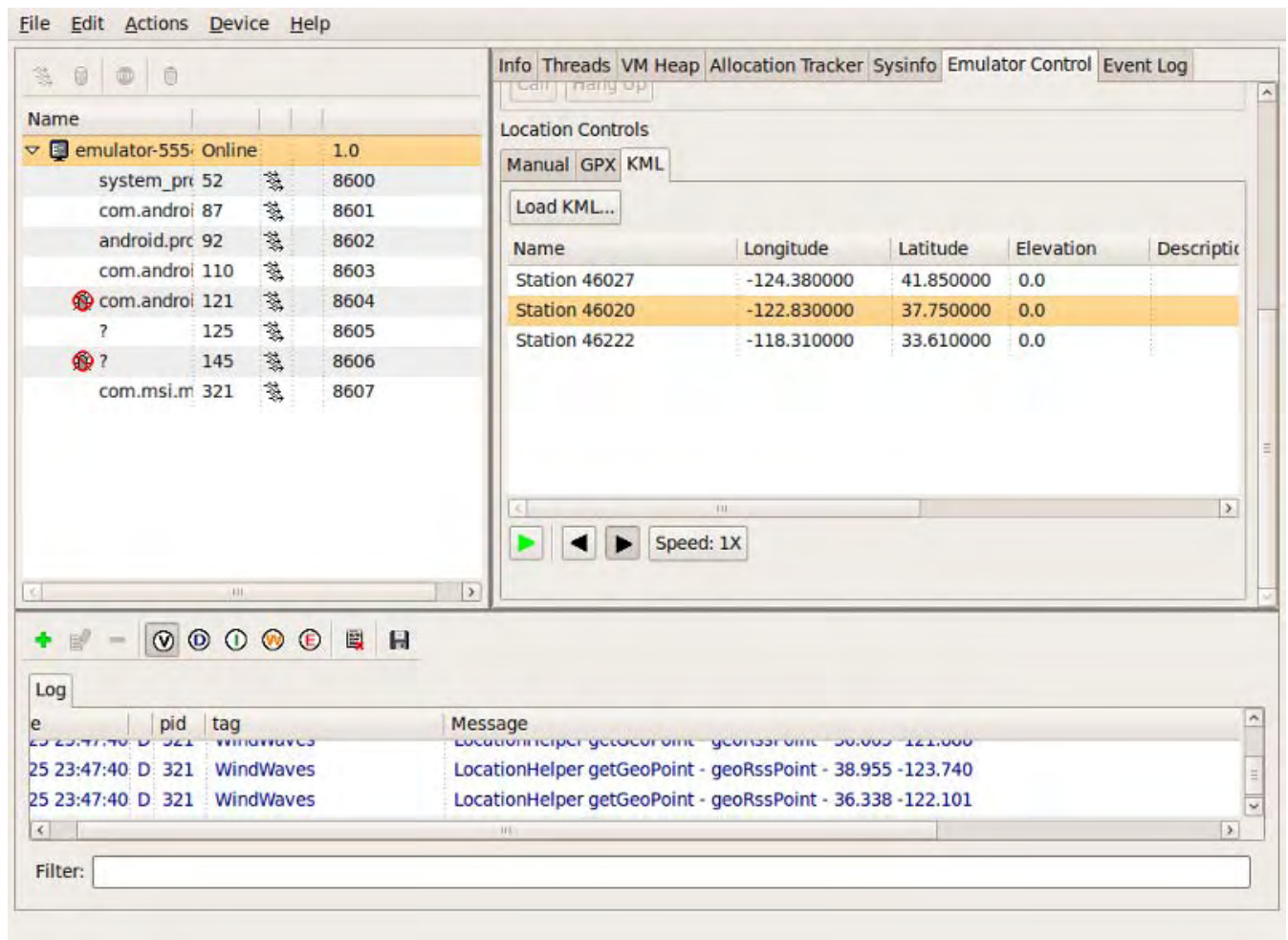


Figure 11.4 Using the DDMS tool with a KML file to send mock location information.

KML is very flexible and expressive, but it has some drawbacks when working with it in an Android emulator context. As we have noted, the DDMS parser basically just looks for the coordinate elements in the file and sends the latitude, longitude, and elevation for each in a sequence, one per second (the documentation says one Placemark per second). Timing, and other advanced features of KML are not yet supported by DDMS. Because of this we find it more valuable at present to use GPX as a debugging and testing format (where detailed timing is supported).

KML is still important though, remember it's the international standard, so it is sure to gain some traction. Also, KML is an important format for other Google applications, so you may encounter it more frequently in other contexts than GPX.

Now that we have seen how to send mock location information to the emulator, in various formats, the next thing we need to do is step out of the built in maps application and start creating our own programs that rely on location.

11.2. Using the LocationManager and LocationProvider

When building location aware applications on the Android platform there are several key classes you will use very often. A LocationProvider provides location data using several metrics, and its data is accessed through a LocationManager.

LocationManager, along with returning the available providers, also allows you to attach a LocationListener to be updated when the device location changes, and or directly fire an Intent based on the proximity to a specified latitude and longitude. The last known Location is also available directly from the manager.

The Location class is a bean that represents all the location data available from a particular snapshot in time. Depending on the provider used to populate it, a Location may or may not have all the possible data present (it might not include speed or altitude for example).

To get started working with all of these concepts the first thing we need to do is get a handle on the LocationManager.

11.2.1 Accessing location data with LocationManager

The central class that you will use to interact with location related data on Android is the LocationManager. Before you can check what providers are available, or query the last known Location, you need to get the manager from the system service. This is shown in listing 11.3, which includes a portion of the MapViewActivity that will drive our Wind and Waves application.

Listing 11.3 The start of the MapViewActivity demonstrating how to retrieve an instance of the LocationManager, and set the LocationProvider.

```
public class MapViewActivity extends MapActivity {                                     #1

    private static final int MENU_SET_SATELLITE = 1;
    private static final int MENU_SET_MAP = 2;
    private static final int MENU_BUOYS_FROM_MAP_CENTER = 3;
    private static final int MENU_BACK_TO_LAST_LOCATION = 4;

    . . . Handler and LocationListeners omitted here for brevity - shown in later listings

    private MapController mapController;
    private LocationManager locationManager;                                         #2
    private LocationProvider locationProvider;                                       #3
    private MapView mapView;
    private ViewGroup zoom;
    private Overlay buoyOverlay;
    private ProgressDialog progressDialog;
    private Drawable defaultMarker;
    private ArrayList<BuoyOverlayItem> buoys;

    @Override
    public void onCreate(final Bundle icle) {
        super.onCreate(icle);
        this setContentView(R.layout.mapview_activity);

        this.mapView = (MapView) this.findViewById(R.id.map_view);
        this.zoom = (ViewGroup) findViewById(R.id.zoom);
        this.zoom.addView(this.mapView.getZoomControls());

        this.defaultMarker =
            MapViewActivity.this.getResources().getDrawable(R.drawable.redpin);
        this.defaultMarker.setBounds(0, 0,
            this.defaultMarker.getIntrinsicWidth(),
            this.defaultMarker.getIntrinsicHeight());

        this.buoys = new ArrayList<BuoyOverlayItem>();
    }

    @Override
    public void onStart() {
        super.onStart();
        this.locationManager =
            (LocationManager) this.getSystemService(Context.LOCATION_SERVICE);      #4
        this.locationProvider =
            this.locationManager.getProvider(LocationManager.GPS_PROVIDER);          #5
    }
}
```



```

        // LocationListeners omitted here for brevity

        GeoPoint lastKnownPoint = this.getLastKnownPoint();
        this.mapController = this.mapView.getController();
#|6
        this.mapController.setZoom(10);
#|6
        this.mapController.animateTo(lastKnownPoint);
#|6
        this.getBuoyData(lastKnownPoint);
#|6
    }

    . . . onResume and onPause omitted for brevity
    . . . other portions of MapViewActivity are included in later listings in this chapter

    private GeoPoint getLastKnownPoint() {
        GeoPoint lastKnownPoint = null;

        Location lastKnownLocation =
            this.locationManager.getLastKnownLocation(LocationManager.GPS_PROVIDER);    #7

        if (lastKnownLocation != null) {
            lastKnownPoint = LocationHelper.getGeoPoint(lastKnownLocation);
        } else {
            lastKnownPoint = LocationHelper.GOLDEN_GATE;
        }
        return lastKnownPoint;
    }
}

```

1. Extend MapActivity
2. Define a LocationManager member
3. Define a LocationProvider member
4. Instantiate the LocationManager system service
5. Assign the GPS LocationProvider
6. Setup the map - we will cover this more later
7. Get the last known Location from the manager

The first thing to note with the `MapViewActivity` is that it extends `MapActivity` **#1**. Though we aren't focusing on the `MapActivity` details yet (that will be covered in section 11.3), this extension is still important to note. After we get the class started the next thing we see is that we are declaring member variables for `LocationManager` **#2** and `LocationProvider` **#3**.

In order to instantiate the `LocationManager` we then use the Activity `getSystemService(String name)` method **#4**. `LocationManager` is a "system service," so we don't directly create it, we let the system return it. After we have the `LocationManager`, we then also assign the `LocationProvider` we want to use with the manager's `getProvider` method **#5**. In this case we are using the GPS provider. We will learn more about the `LocationProvider` class in the next section.

Once we have the manager and provider in place, we also use the `onCreate` method of our Activity to instantiate a `MapController`, and set some initial state for the screen. A `MapController`, and the `MapView` it manipulates, are items we will also cover more in section 11.3.

Along with helping you setup the provider you need, `LocationManager` also supplies quick access to the last known `Location` **#7**. This method is very useful if you just need a quick "fix" on the last location, as opposed to the more involved techniques for registering for periodic location updates with a listener (a topic we will cover in section 11.2.3).

Though we don't use it in this listing, or in the Wind and Waves application at all, the `LocationManager` additionally allows you to directly register for proximity "alerts." If you need to fire an `Intent` based on proximity to a defined location, you will want to be

aware of the `addProximityAlert` method. This method lets you set the location you are concerned about with latitude and longitude, and then also lets you specify a radius, and a `PendingIntent`. If the device comes within the range, the `PendingIntent` is fired. (There is a corresponding `removeProximityAlert` method as well.)

Getting back to the main thing we will use the `LocationManager` for with Wind and Waves, we next need to look a bit more deeply at the GPS `LocationProvider`.

11.2.2 Using one or more *LocationProviders*

`LocationProvider` is an abstract class that helps define the capabilities of a given provider implementation. Different provider implementations, which are simply responsible for returning `Location` information, may be available on different devices, and in different circumstances.

So what are the “different providers,” and why are multiple providers necessary? Well, those are really context sensitive questions, meaning the answer is, “it depends.” What provider implementations are available depends on the hardware capabilities of the device – does it have a GPS receiver, for example? It also depends on the situation, even if the device has a GPS receiver, can it currently receive data from satellites, or is the user somewhere that’s not possible (an elevator, or a tunnel, etc)?

At runtime you will need to query for the list of providers available, and then use the most suitable one (or ones - it can often be advantageous to fall back to a less accurate provider if your first choice is not available or enabled). The most common provider, and the only one available in the emulator, is the `LocationManager.GPS_PROVIDER` provider (which uses the GPS receiver). Because it is the most common (and most accurate), and what is available in the emulator, this is the provider we are going to use for Wind and Waves. Keep in mind though, at runtime in a real device there will normally be multiple providers, including the `LocationManager.NETWORK_PROVIDER` provider (which uses cell tower and WiFi access points to determine location data).

In listing 11.3 we already saw how you can obtain the GPS provider directly using the `getProvider(String name)` method. Some alternatives to this direct access of a particular provider approach are shown in table 11.2.

Table 11.2 Various methods for obtaining a `LocationProvider` reference.

LocationProvider Code Snippet	Description
<pre>List<String> providers = locationManager.getAllProviders();</pre>	Get all of the providers registered on the device.
<pre>List<String> enabledProviders = locationManager.getAllProviders(true);</pre>	Get all of the currently enabled providers.
<pre>locationProvider = locationManager.getProviders(true).get(0);</pre>	A shortcut to get the first enabled provider, regardless of type.
<pre>locationProvider = this.locationManager.getBestProvider(myCriteria, true);</pre>	An example of getting a <code>LocationProvider</code> using a specified <code>Criteria</code> (you can create a <code>Criteria</code> instance and specify if bearing, or altitude, or cost, and other metrics, are required or not).

Different providers may support different location related metrics, and have different costs or capabilities. The `Criteria` class helps to define what each provider instance can handle. Among the metrics available are the following: latitude and longitude, speed, bearing, altitude, cost, and power requirements.

Another aspect of working with location data and `LocationProvider` instances that is important is permissions. Location related permissions need to be in your manifest depending on the providers you want to use. Listing 11.4 shows the Wind and Waves manifest XML file, which includes both COARSE and FINE grained location related permissions.

Listing 11.4 the Wind and Waves XML manifest file showing the COARSE and FINE location related permissions.

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

    <application android:icon="@drawable/wave_45"
        android:label="@string/app_name" android:theme="@android:style/Theme.Black">

        <activity android:name="StartActivity" android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name="MapViewActivity" />
        <activity android:name="BuoyDetailActivity" />

        <uses-library android:name="com.google.android.maps" />

    </application>

    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />    #1
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />      #2
    <uses-permission android:name="android.permission.INTERNET" />

</manifest>
```

1. Allow the NETWORK provider with ACCESS_COARSE_LOCATION
2. Allow the GPS provider with ACCESS_FINE_LOCATION

In terms of location permissions we are including both the ACCESS_COARSE_LOCATION #1, and ACCESS_FINE_LOCATION #2 permissions in our manifest. The COARSE permission corresponds to the NETWORK provider (cell and WiFi based data), and the FINE permission corresponds to the GPS provider. We aren't actually using the network provider in Wind and Waves, but we have noted that a worthwhile enhancement would be to fall back to the network provider if the GPS provider is unavailable or disabled - this permission would allow that.

Once you understand the basics of `LocationManager` and `LocationProvider`, the next step is to unleash the real power and register for periodic location updates in your application with the `LocationListener` class.

11.2.3 Receiving location updates with LocationListener

The way to keep abreast of the device location from within an Android application is to create a `LocationListener` implementation and register it to receive updates. `LocationListener` is a very flexible and powerful interface that lets you filter for many types of location events based on various properties. You have to implement the interface and register your instance to receive location data callbacks.

Listing 11.5 brings all of the pieces we have covered thus far in to scope as we create several `LocationListener` implementations for the `Wind and Waves MapViewActivity` (the parts we left out of listing 11.3), and then register those listeners using the `LocationManager` and `LocationProvider`.

Listing 11.5 The creating of `LocationListener` implementations in the `MapViewActivity` and registration of the listeners.

```
. . . start of class in Listing 11.3

private final LocationListener locationListenerGetBuoyData = new LocationListener() {
#1
    public void onLocationChanged(final Location loc) {
#2
        int lat = (int) (loc.getLatitude() * LocationHelper.MILLION);
#3
        int lon = (int) (loc.getLongitude() * LocationHelper.MILLION);
#3

        GeoPoint geoPoint = new GeoPoint(lat, lon);
#4
        MapViewActivity.this.getBuoyData(geoPoint);
#5
    }
    public void onProviderDisabled(final String s) {
    }
    public void onProviderEnabled(final String s) {
    }
    public void onStatusChanged(final String s, final int i, final Bundle b) {
    }
};

private final LocationListener locationListenerRecenterMap = new LocationListener() {
    public void onLocationChanged(final Location loc) {
        int lat = (int) (loc.getLatitude() * LocationHelper.MILLION);
        int lon = (int) (loc.getLongitude() * LocationHelper.MILLION);

        GeoPoint geoPoint = new GeoPoint(lat, lon);
        MapViewActivity.this.mapController.animateTo(geoPoint);
#6
    }
    public void onProviderDisabled(final String s) {
    }
    public void onProviderEnabled(final String s) {
    }
    public void onStatusChanged(final String s, final int i, final Bundle b) {
    }
};

@Override
public void onStart() {
    super.onStart();
    this.locationManager = (LocationManager)
this.getSystemService(Context.LOCATION_SERVICE);
    this.locationProvider =
this.locationManager.getProvider(LocationManager.GPS_PROVIDER);

    if (locationProvider != null) {
        this.locationManager.requestLocationUpdates(
            locationProvider.getName(), 3000, 185000, this.locationListenerGetBuoyData); #7
        this.locationManager.requestLocationUpdates(
            locationProvider.getName(), 3000, 1000, this.locationListenerRecenterMap); #8
    } else {
        Log.e(Constants.LOGTAG, " " + CLASSTAG + " NO LOCATION PROVIDER AVAILABLE");
        Toast.makeText(this, "Wind and Waves cannot continue, the GPS location provider
            is not available at this time.", Toast.LENGTH_SHORT).show();
        this.finish();
    }
}

. . . remainder of repeated code omitted (see listing 11.3)
```

```
}
```

1. Create a `LocationListener` with an anonymous instance
2. Implement the `onLocationChanged` method
3. Get latitude and longitude from the `Location`
4. Create a `GeoPoint` from the latitude and longitude
5. Update the map pins (the buoy data) based on the new location
6. Move the map to the new location
7. Register the `locationListenerGetBuoyData`
8. Register the `locationListenerRecenterMap`

When implementing the `LocationListener` interface it is often practical to use an anonymous inner class **#1**. For our `MapViewActivity` we have created two `LocationListener` implementations because we want to register them both using different settings, as we will see momentarily.

Within the first listener, `locationListenerGetBuoyData`, we see how the `onLocationChanged` method is implemented **#2**. In that method we get the latitude and longitude from the `Location` sent in the callback **#3**. We use the data to create a `GeoPoint` by multiplying the latitude and longitude by 1 million (1e6) **#4**. The 1e6 format is necessary because `GeoPoint` requires “microdegrees” for coordinates.

After we have the data we then update the map (using a helper method that resets a map `Overlay`, the details of which we will cover in the next section) **#5**. In the second listener, `locationListenerRecenterMap`, we perform a different task - we center the map **#6**.

The reason we are using two listeners becomes crystal clear when we see how listeners are registered with the `requestLocationUpdates` method of the `LocationManager` class. Here we are registering the first one, `locationListenerGetBuoyData`, to fire only when the new device location is a long way away from the previous one (185000 meters, we chose this number to stay just under 100 nautical miles, which is the radius we will use to pull buoy data for our map - we don't need to redraw the buoy data on the map if the user moves less than 100 nautical miles) **#7**. And, we are registering the second one, `locationListenerRecenterMap`, to fire more frequently (so the map view re-centers if the user stays inside of our application but moves more than 1000 meters) **#8**. Using separate listeners like this allows us to fine tune the event processing (rather than having to build in our own logic to do different things based on different values with one listener).

Register Location listeners carefully

The time parameter to the `requestLocationUpdates` method should be used carefully. Getting location updates too frequently (less than 60000 ms per the documentation) can wear down the battery and make the application to “noisy.” In this sample we have used an extremely low value for the time parameter for debugging purposes (3000 ms). You should never use a value lower than the recommended 60000ms in production code.

Though our implementation here works, and is the most straightforward example, keep in mind that our registration of `LocationListener` instances could be made even more robust by implementing the `onProviderEnabled` and `onProviderDisabled` methods. Using those methods, and different providers, you can see how you could provide useful messages to the user, and also provide a graceful fall back through a set of providers (if GPS becomes disabled, try network, and so on).

With `LocationManager`, `LocationProvider`, and `LocationListener` instances in place, the next thing we need to address is more detail concerning the `MapActivity` and `MapView` we are using.

11.3. Working with maps

We have seen the start of the `MapViewActivity` our Wind and Waves application will use in the previous sections. There we covered the supporting classes and handling of registering to receive location updates. With that structure in place, we now will focus on the map details themselves.

The `MapViewActivity` screen will look like the screen shot in figure 11.5, where several map `Overlay` classes are used on top of a `MapView` within a `MapActivity`.



Figure 11.5 The `MapViewActivity` from the `Wind and Waves` application showing a `MapActivity` with `MapView`.

In order to use the `com.google.android.maps` package on the Android platform, and to support all the concepts related to a `MapView` you are required to use a `MapActivity`.

11.3.1 Extending `MapActivity`

A `MapActivity` is the gateway to the Android Google Maps like API package, and other useful map related utilities. There are several details behind creating and using a `MapView` that we as developers are fortunate enough not to have to worry about, because `MapActivity` handles them for us.

We will learn more about `MapView`, which is what we really care about as developers building map applications, in the next section, but it's important to first understand what a `MapActivity` is, and why it's necessary. At its core, a `MapActivity` supports a `MapView` (a `MapActivity` is the only place a `MapView` can be used), and manages all the network and file system intensive setup and tear down tasks needed for supporting the same.

The `MapActivity` `onResume` method automatically sets up network threads for various map related tasks, and caches map section tile data on the file system, for example,

And, the `onPause` method cleans these up. Without this class all of these details would be extra housekeeping that any `Activity` wishing to include a `MapView` would have to repeat each time.

There isn't a lot you will need to actually do with regard to `MapActivity` in code. Extending this class (as we did in listing 11.3), and making sure to use only one instance per process (more than one and "unexpected" results may occur), and including a special manifest element to enable the `com.google.android.maps` package is all you need. You may have noticed the curious "uses-library" element in the Wind and Waves manifest in listing 11.4.

```
<uses-library android:name="com.google.android.maps" />
```

The `com.google.android.maps` package, where `MapActivity`, `MapView`, `MapController`, and other related classes such as `GeoPoint` and `Overlay` all reside, is "not a standard package in the Android library" per the documentation. This manifest element is required to pull in support for the maps package.

Once you have the `uses-library` element, and have a basic `Activity` that extends `MapActivity`, the details come in inside the `MapView` and related `Overlay` classes.

11.3.2 Using a MapView

A `MapView` is a miniature version of many of the Google Maps API concepts in the form of a `View` for your Android application. A `MapView` displays tiles of a map which it obtains over the network as the map is moved and zoomed, much like the web version of Google Maps.

Many of the concepts from the standard Google Maps API are also present in Android through the `MapView`. For instance, `MapView` supports a plain map mode, a satellite mode, a street view mode, and a traffic mode. When you want to write something on top of the map, from a straight line between two points, to "push pin" markers, or full on images or anything else you use an `Overlay`.

Examples of several of these concepts can be seen in the `MapViewActivity` screen shots for the Wind and Waves application, such as what is shown in figure 11.5. That same `MapViewActivity` is shown again in figure 11.6, switched into satellite mode, and zoomed in several levels (the "B" pins are buoy markers).



Figure 11.6 The `MapViewActivity` from the `Wind and Waves` application using satellite mode and zoomed in on a position near San Francisco.

The `com.google.android.maps` package supports a good deal of the Google Maps API concepts, but isn't identical, of course. We have already seen the `MapView` we will use for the `Wind and Waves` application declared and instantiated in listing 11.3. Here we will discuss the use of this class inside our `Activity` to control, position, zoom, populate, and overlay our map.

Before we can use a map at all though, we have to get a Google Maps API key and declare it in our layout file. Listing 11.6 shows the `MapActivity` layout file we are using with a special `android:apiKey` attribute.

Listing 11.6 A `MapView` layout file showing the inclusion of the Google Maps API key.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:gravity="center_horizontal" android:padding="10px">
```



```

<com.google.android.maps.MapView                                #1
    android:id="@+id/map_view"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:enabled="true"
    android:clickable="true"
    android:apiKey="05lSygx-ttd-J5GXfsIB-dlpNtggca4I4DMYVqQ" />    #2

<LinearLayout
    android:id="@+id/zoom"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentBottom="true"
    android:layout_centerInParent="true">
</LinearLayout>

</RelativeLayout>

```

1. Defining a MapView element in XML
2. Include the apiKey attribute

A `MapView` can be declared in XML just like other `View` components **#1**. In order to use the Google Maps network resources though, a `MapView` requires an API key **#2**. Obtaining a map key is done via a special Google Maps Android key registration web page: <http://code.google.com/android/maps-api-signup.html>.

Before you can register for a key you first need to get the “MD5 fingerprint” of the certificate that is used to sign your application. This sounds tricky, but it's really very simple. When you are working with the emulator the SDK has a “Debug Certificate” that is always in use. To get the MD5 fingerprint for this certificate you can use the following command (on Mac and Linux, on Windows adjust for the user's home directory and the slashes) :

```

cd ~/.android
keytool -list -keystore ./debug.keystore -storepass android -keypass android

```

Getting a key for a production application involves the same process, but you need to use the actual certificate your APK file is signed with (rather than the debug.keystore file). The Android documentation has a good deal of additional information about obtaining a maps key (<http://code.google.com/android/toolbox/apis/mapkey.html>).

The Maps Key Conundrum

One issue with the maps key process is that you need to declare the key in the layout file. Because there can only be one `MapActivity` and hence one `MapView` per application/process, it would seem more logical to declare the key in the application manifest, or in an environment variable or properties file, but none of those is the case. With the key in the layout file you have to remember to update the key between debug (emulator) and production modes, **and** if you debug on different development machines you will also have to remember to switch keys by hand.

Once you have a `MapActivity` with a `MapView`, and have setup your view in the layout file, complete with map key, you can then make full use of the map. Several of the listings we have already seen up to this point are using the `MapView` we have declared in the Wind and Waves application. In listing 11.7 we are repeating a few of the map related lines of code we have already shown, and we are bringing in additional related items, to consolidate all the map related concepts in one listing.

Listing 11.7 Portions of code from different areas in the Wind and Waves MapViewActivity that demonstrate working with maps.

```
. . . from onCreate
this.mapView = (MapView) this.findViewById(R.id.map_view);           #1
this.zoom = (ViewGroup) findViewById(R.id.zoom);                     #2
this.zoom.addView(this.mapView.getZoomControls());                  #3

. . . from onStart
this.mapController = this.mapView.getController();                  #4
this.mapController.setZoom(10);                                     #5
this.mapController.animateTo(lastKnownPoint);                       #6

. . . from onOptionsItemSelected
case MapViewActivity.MENU_SET_MAP:
    this.mapView.setSatellite(false);                               #7
    break;
case MapViewActivity.MENU_SET_SATELLITE:
    this.mapView.setSatellite(true);
    break;
case MapViewActivity.MENU_BUOYS_FROM_MAP_CENTER:
    this.getBuoyData(this.mapView.getMapCenter());                 #8
    break;
```

1. Inflate the MapView from layout
2. Include another View for the zoom controls
3. Get the zoom controls from the MapView
4. Get the MapController
5. Set the initial zoom level using the controller
6. Animate to a given GeoPoint using the controller
7. Set the map satellite mode
8. Get coordinates from the map center

MapView is a ViewGroup and you can declare it in XML and inflate it just like other view components **#1**. Because it is a ViewGroup you can also combine and attach other elements to it. Beyond the MapView itself, we are also using a separate additional ViewGroup for the zoom controls **#2**, and attaching the controls from the map to it **#3**.

Next we get a MapController from the MapView **#4**, and then use the controller to set the initial zoom level **#5**, and animate to a specified GeoPoint **#6**. The controller is what you use to zoom and move the map. Also, when the user chooses to do so via the menu, we are setting the mode of the map from plain to satellite, and vice versa **#7**. Along with manipulating the map itself, we can also get data back from it, such as the coordinates of the map center **#8**.

Above and beyond manipulating the map, and getting data from the map, you also have the ability to draw items on top of the map using any number of Overlay instances.

11.3.3 Placing data on a map with an Overlay

The small “B” push pin style icons on the MapViewActivity for the Wind and Waves application that we have seen in several figures up to this point are drawn on the screen at specified coordinates using an Overlay.

Overlay is a generalized base class for different specialized implementations. You can roll your own Overlay by extending the class, or you can use the included MyLocationOverlay. The MyLocationOverlay class lets you display a user's current location with a compass, and also has some other useful features like including a LocationListener for convenient access to position updates.

Another common use case for a map (in addition to showing you where you are) is the need to place multiple marker items on it - the ubiquitous push pins. We have this exact requirement for the Wind and Waves application. We need to create buoy markers for the location of every buoy using data we get back from the NBDC feeds.

Android provides built in support for this with the `ItemizedOverlay` base class, and the `OverlayItem`.

An `OverlayItem` is a simple bean that includes a title, a text “snippet,” a drawable marker, and coordinates using a `GeoPoint` (and a few other properties, but you get the idea). Listing 11.8 is the buoy data related `BuoyOverlayItem` class that we are using for Wind and Waves.

Listing 11.8 The `BuoyOverlayItem` class that extends `OverlayItem` and supplies the necessary properties.

```
public class BuoyOverlayItem extends OverlayItem {                                #1

    public final GeoPoint point;
    public final BuoyData buoyData;

    public BuoyOverlayItem(final GeoPoint point, final BuoyData buoyData) {
        super(point, buoyData.title, buoyData.dateString);                    #2
        this.point = point;
        this.buoyData = buoyData;                                            #3
    }
}
```

1. Extend `OverlayItem`
2. Call the superclass constructor with required properties
3. Include an extra `BuoyData` property

We extend `OverlayItem` to bring in all the necessary properties of an item to be drawn on the map: a location, a title, a snippet, and so on **#1**. In the constructor we make the call to the superclass with the required properties **#2**, and we also assign additional elements our subclass supports. In this case we are adding a `BuoyData` member (itself a bean with name, water temperature, wave height, and so on type properties) **#3**.

After we have the individual item class prepared the next thing we need is a class that extends `ItemizedOverlay` and uses a `Collection` of the items to display them on the map one by one. Listing 11.9, the `BuoyItemizedOverlay` class, shows how this works.

Listing 11.9 The `BuoyItemizedOverlay` class that demonstrates using an `ItemizedOverlay` for a collection of `OverlayItem` instances.

```
public class BuoyItemizedOverlay extends ItemizedOverlay<BuoyOverlayItem> {      #1

    private final List<BuoyOverlayItem> items;                                #2
    private final Context context;

    public BuoyItemizedOverlay(final List<BuoyOverlayItem> items,
        final Drawable defaultMarker, final Context context) {
        super(defaultMarker);
        this.items = items;                                                    #3
        this.context = context;
        this.populate();
    }

    @Override
    public BuoyOverlayItem createItem(final int i) {
        return this.items.get(i);                                              #4
    }

    @Override
    protected boolean onTap(final int i) {
        final BuoyData bd = this.items.get(i).buoyData;                      #5

        LayoutInflater inflater = LayoutInflater.from(this.context);
        View bView = inflater.inflate(R.layout.buoy_selected, null);
        TextView title = (TextView) bView.findViewById(R.id.buoy_title);

        . . . rest of view inflation omitted for brevity

        new AlertDialog.Builder(this.context)
```

```

        .setView(bView)
        .setPositiveButton("More Detail", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface di, int what) {
                Intent intent = new Intent(BuoyItemizedOverlay.this.context,
BuoyDetailActivity.class);
                BuoyDetailActivity.buoyData = bd;
                BuoyItemizedOverlay.this.context.startActivity(intent);
            }
        })
        .setNegativeButton("Cancel", new DialogInterface.OnClickListener() {
            public void onClick(DialogInterface di, int what) {
                di.dismiss();
            }
        })
        .show();

    return true;
}

@Override
public int size() {
    return this.items.size();
}

@Override
public void draw(Canvas canvas, MapView mapView, boolean b) {
    super.draw(canvas, mapView, false);
}
}

```

#6

#7

1. Extend ItemizedOverlay
2. Include a Collection of OverlayItem elements
3. Provide the drawable marker to the super class
4. Implement the createItem method
5. When an item is "tapped" get data and display
6. Implement the size method
7. Other methods such as draw are also available

The `BuoyItemizedOverlay` class extends `ItemizedOverlay` #1, and includes a Collection of `BuoyOverlayItem` elements #2. In the constructor we pass the `Drawable` marker to the parent class #3. This marker is what is drawn on the screen in the overlay to represent each point on the map.

`ItemizedOverlay` takes care of many of the details we would have to tackle ourselves if we weren't using it (if we were just making our own `Overlay` with multiple points drawn on it), such as the drawing of items, and focus and event handling. For every element in the Collection of items it holds it invokes the `onCreate` method #4, and it supports facilities like `onTap` #5, where we can react when a particular overlay item is selected by the user. In our code we inflate some views and display an `AlertDialog` with information about the respective buoy when a `BuoyOverlayItem` is tapped. From the alert the user can navigate to more detailed information if they choose.

The **size** method tells the `ItemizedOverlay` how many elements it needs to process #6, and even though we aren't doing anything special with it in our case, there are also methods like `onDraw` that can be customized if necessary #7.

When working with a `MapView` you create the `Overlay` instances you need and then you add them on top of the map. *Wind and Waves* is using a separate `Thread` to retrieve the buoy data in the `MapViewActivity` (the data retrieval code is not shown, but is included in the code download for this chapter), and then when ready we send a `Message` to a `Handler` to add the `BuoyItemizedOverlay` to the `MapView`. These details are seen in listing 11.10.

Listing 11.10 The Handler *Wind and Waves* uses to add overlays to the `MapView`.

```

private final Handler handler = new Handler() {
    @Override
    public void handleMessage(final Message msg) {

        MapViewActivity.this.progressBar.dismiss();

        if (MapViewActivity.this.mapView.getOverlays().contains(
            MapViewActivity.this.buoyOverlay)) {
            MapViewActivity.this.mapView.getOverlays().remove(
                MapViewActivity.this.buoyOverlay);           #1
        }

        // add buoys itemized overlay
        MapViewActivity.this.buoyOverlay = new BuoyItemizedOverlay(
            MapViewActivity.this.buoys,
            MapViewActivity.this.defaultMarker,
            MapViewActivity.this);                             #2
        MapViewActivity.this.mapView.getOverlays().add(
            MapViewActivity.this.buoyOverlay);                 #3
    }
};

```

1. Removing the Overlay if already present
2. Creating the BuoyItemizedOverlay
3. Adding the Overlay to the MapView

A MapView contains a Collection of Overlay elements, and as such you can remove previous elements if you need to. We use the remove method to cleanup any existing BuoyOverlayItem class **#1** before we create **#2** and add a new one **#3** - this way we aren't simply adding more items on top of each other, rather we are resetting the data.

The built in Overlay subclasses have handled our requirements here perfectly, which is very helpful. The ItemizedOverlay and OverlayItem classes have allowed us to complete the Wind and Waves application without having to make our own Overlay subclasses directly. Keep in mind though, if you need to, you can go to that level and implement your own draw, tap, touch, and so on methods within your custom Overlay.

With our sample application now complete, and providing us with buoy data using a MapActivity and MapView, we next need to address one additional maps related concept that we haven't yet encountered, but is nonetheless very important - geocoding.

11.4 Converting places and addresses with Geocoder

Geocoding is described in the documentation as converting a "street address or other description of a location" into latitude and longitude coordinates. Reverse geocoding is the opposite, converting latitude and longitude into an address.

We aren't using geocoding in the Wind and Waves application because it's obviously not as useful in the ocean as it is with landmarks, cities, addresses, and so on. Nevertheless, geocoding is an invaluable tool to have at your disposal when working with coordinates and maps. To demonstrate geocoding listing 11.11 includes a new single Activity application, GeocoderExample, which demonstrates the concepts.

Listing 11.11 A short Geocoder example that does address to coordinate conversion, and coordinate to address reverse conversion.

```

@Override
public void onCreate(final Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    this setContentView(R.layout.main);
    this.input = (EditText) this.findViewById(R.id.input);
    this.output = (TextView) this.findViewById(R.id.output);
    this.button = (Button) this.findViewById(R.id.geocode_button);
    this.isAddress = (CheckBox) this.findViewById(R.id.checkbox_address);
}

```

```

        this.button.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                GeocodeExample.this.output.setText(
                    GeocodeExample.this.performGeocode(
                        GeocodeExample.this.input.getText().toString(),
                        GeocodeExample.this.isAddress.isChecked()));
            }
        });
    }

    private String performGeocode(final String in, final boolean isAddr) {
        String result = "Unable to Geocode - " + in;
        if (this.input != null) {
            Geocoder geocoder = new Geocoder(this);
#1
            if (isAddr) {
                try {
#2
                    List<Address> addresses = geocoder.getFromLocationName(in, 1);

                    if (addresses != null) {
                        result = addresses.get(0).toString();
                    }
                } catch (IOException e) {
                    Log.e("GeocodExample", "Error", e);
                }
            } else {
                try {
                    String[] coords = in.split(",");
                    if ((coords != null) && (coords.length == 2)) {
                        List<Address> addresses = geocoder.getFromLocation(
                            Double.parseDouble(coords[0]),
                            Double.parseDouble(coords[1]),
                            1);
#3
                        result = addresses.get(0).toString();
                    }
                } catch (IOException e) {
                    Log.e("GeocodExample", "Error", e);
                }
            }
        }
        return result;
    }
}

```

1. Instantiating a Geocoder with Context
2. Getting an Address from a String location name
3. Getting an Address from coordinates

In Android terms you create a `Geocoder` by constructing it with the `Context` of your application **#1**. You then use a `Geocoder` to covert either `String` instances that represent place names into `Address` objects with the `getFromLocationName` method **#2**, or latitude and longitude coordinates into `Address` objects with the `getFromLocation` method **#3**.

Figure 11.7 is an example of our simplified `GeocoderExample` in use. In this case we are converting a `String` representing a place (Wrigley Field in Chicago) into an `Address` object which contains latitude and longitude coordinates.



Figure 11.7 A Geocoder usage example that demonstrates turning an address String into an Address object which provides latitude and longitude coordinates.

The GeocoderExample application shows how useful the `Geocoder` is. For instance, if you have data that includes address string portions, or even just place descriptions, it's easy to covert that into latitude and longitude numbers for use with `GeoPoint` and `Overlay`, and so on.

Geocoding rounds out our look at the powerful location and mapping related components of the Android platform.

11.5 Summary

Location, location, location, as they say in real estate, could also be the mantra for the future of mobile computing. Arguably one of the most important features of the Android platform is the support for readily available location information and the inclusion of smart mapping APIs and other location related utilities.

In this chapter we explored the location and mapping capabilities of the Android platform by building an application that setup a `LocationManager` and `LocationProvider`, to which we attached several `LocationListener` instances so that we could receive location updates. We then combined a somewhat unique data source (the National Data Buoy Center) with our location awareness to build an draggable, zoomable, interactive map. To build the map we used a `MapActivity`, with `MapView` and `MapController`, and created an `ItemizedOverlay` to display individual `OverlayItem` elements . In addition to covering the bulk of the Android map and location related API details we also looked at some related utilities like the `Geocoder`.

In the next few chapters of the book that make up the final section we are going to move past the introduction and tutorial stages and work on some larger applications that pull in many of the pieces we have learned about from previous chapters.

12

Putting it all together – A Field Service Application

Now that Android and its core technologies have been introduced and examined up to this point, it is high time to put together a more comprehensive application. In this chapter we are going to put much of what we have learned into a composite application, leveraging skills gained throughout the book. In addition to an in-depth Android application, this chapter's sample application works with a custom web site application which manages data for use by a mobile worker. The aim of the application is to demonstrate a more complex application involving real-world requirements. All of the source code for the server-side application is available for download from the book's companion site. After reading through this chapter and getting familiar with the sample application, you will be ready to strike out on your own and build useful Android applications. Many of the code samples are presented and explained, however if you are in need of more background information on a particular topic, please refer back to earlier chapters where the Android APIs are more fully presented.

If this application is going to represent a useful real world application, we need to put some flesh on it. Beyond just being helpful to understand the application, this definition process will hopefully get you thinking about the kinds of impact a mobile application can have on our economy. This chapter's sample application is a "Field Service Application". Pretty generic name perhaps, but it will prove to be an ample vehicle for demonstrating some key elements required in mobile applications as well as demonstrate the power of the Android platform for building useful applications quickly.

Our application's target user is a fleet technician who works for a national firm which makes its services available to a number of contracted customers. One day, our technician, who we will call a "mobile worker", is replacing a hard drive in the computer at the local fast food restaurant and the next day he is perhaps installing a memory upgrade in a piece of pick and place machinery at a telephone system manufacturer. If you have ever had a piece

of equipment serviced at your home or office and thought the technician's "uniform" did not really match the job he was doing, you have experienced this kind of service arrangement. This kind of technician is often referred to as "hands and feet". He has basic mechanical or computer skills and is able to follow directions reliably, often guided by the manufacturer of the equipment being serviced at the time. Thanks to workers like this, companies can extend their reach to a much broader geography than their internal staffing levels would ever allow. For example, a small manufacturer of retail music sampling equipment might contract with such a firm for providing tech support to retail locations across the country.

Due to our mythical technician's varied schedule and arguable lack of experience in a particular piece of equipment, it is important to equip him with as much relevant and timely information as possible. However, he cannot be burdened with thick reference manuals or specialized tools. So, with a toolbox containing a few hand tools and of course an Android equipped device, our fearless hero is counting on us to provide him with an application that enables him to do his job. And remember, this is the person who restores the ice cream machine to operation at the local Dairy Barn or perhaps fixes the farm equipment's computer controller so the cows get milked on time. You see, you never know where a computer will be found in today's world!

If built well, this application can enable the efficient delivery of service to customers in many industries, where we live, work and play. Let's get started and see what this application must be able to accomplish.

12.1 Field Service Application Requirements

We have established that our mobile worker will be carrying two things – a set of hand tools and an Android device. Fortunately, in this book we are not concerned with the applicability of the hand tools in his toolbox, leaving us free to focus on the capabilities and features of a Field Service Application running on the Android platform. In this section, we are going to define the high level application requirements.

12.1.1 Basic Requirements

Before diving into the bits and bytes of data requirements and application features, it is helpful to enumerate some basic requirements and assumptions about our Field Service Application, or FSA for short. Here are a few items which come to mind for such an application:

The mobile worker is dispatched by a "home-office/dispatching authority" which takes care of prioritizing and distributing Job Orders to the appropriate technician.

The mobile worker is carrying an Android device which has full data service, i.e. a device capable of browsing rich web content. The application needs to access the Internet for data transfer as well.

The home-office dispatch system and the mobile worker share data via a wireless Internet connection on an Android device; a laptop computer is not necessary, or even desired.

A business requirement is the proof of completion of work, most readily accomplished with the capture of a customer's signature. Of course, an electronic signature is preferred.

The home-office desires to receive job completion information as soon as possible, as this accelerates the invoicing process, which improves cash flow.

The mobile worker is also eager to perform as many jobs as possible as he is paid by the job, not by the hour, so getting access to new Job information as quickly as possible is a benefit to the mobile worker.

The mobile worker needs information resources in the field and can use as much information as possible about the problem he is being asked to resolve. The mobile worker may even have to place orders for replacement parts while in the field.

The mobile worker will require navigation assistance as he is likely covering a rather large geographic area.

Lastly, it is important that the application be simple to use with a minimum number of requirements for the mobile worker – i.e. it needs to be intuitive.

There are likely additional requirements for such an application, but this list is adequate for our purposes. One of the most glaring omissions from our list is security.

Security in this kind of an application comes down to two fundamental aspects. The first is physical security of the Android device. Our assumption is that the device itself is locked and only the authorized worker is using it. A bit naïve perhaps, but there are more important topics we need to cover in this chapter. If this bothers you, just assume there is a sign-in screen with a password field which pops up at the most inconvenient times, forcing you to tap in your password on a very small keypad. Feel better now? The second security topic is the secure transmission of data between the Android device and the dispatcher. This is most readily accomplished through the use of a Secure Sockets Layer (SSL) connection whenever required.

The next step in defining this application is to examine the data flows and discuss the kind of information which must be captured to satisfy the functional requirements.

12.1.2 Data Model

Throughout this chapter, the term Job refers to a specific task or event that our mobile worker engages in. For example, a request to replace a hard drive in a computer at the book store is a Job. A request to upgrade the firmware in the fuel injection system at the refinery is likewise a Job. The home office dispatches one or more Jobs to the mobile worker on a regular basis. There are certain data elements in the Job which are helpful to the mobile worker to accomplish his goal of completing the Job. This information comes from the home office. Where the home office gets this information is not our concern in this application.

In this chapter's sample application, there are only two pieces of information the mobile worker is responsible for submitting to the dispatcher. The first requirement is that the mobile worker communicates to the home office that a Job has been "closed", i.e. completed. The second requirement is the collection of an electronic signature from the customer, acknowledging that the Job has in fact been completed. Figure 12.1 depicts these data flows.

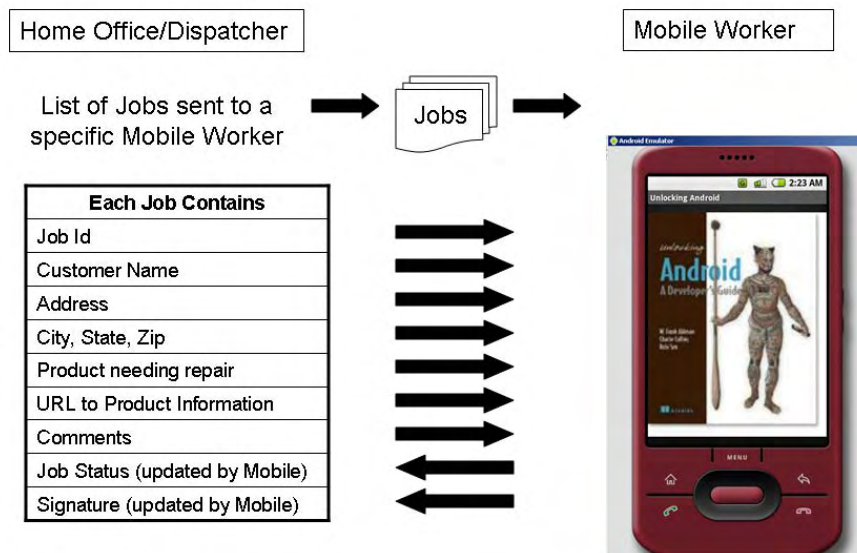


Figure 12.1 depicts data flows between the Home Office and a Mobile Worker

Of course there are additional pieces of information that may be helpful here, such as customer phone number, anticipated duration of the Job, replacement parts required in the repair (including tracking numbers), any observations about the condition of related equipment and much more. While these are very important to a real world application, these pieces of information are extraneous to the goals of this chapter and are left as an exercise for you to extend the application for your own learning and use.

The next objective is to determine how data is stored and transported.

12.1.3 Application Architecture and Integration

Now that we know which entities are responsible for the relevant data elements, and in which direction they flow, let's look at how the data is stored and exchanged. We will be

deep into code before too long, but for now we will discuss the available options and continue to examine things from a requirements perspective, building to a proposed architecture.

At the home office, the dispatcher must manage data for multiple mobile workers. The best tool for this purpose is a relational database. The options here are numerous, but we will make the simple decision to use MySQL, a popular open source database. Not only are there multiple mobile workers, but the organization we are building this application for is quite distributed, with employees in multiple markets and time-zones. Because of the distributed nature of the dispatching team, it has been decided to host the MySQL database in a datacenter where it is accessed via a browser-based application. For this sample application, the dispatcher system is super simple and written in PHP.

Data storage requirements on the mobile device are quite modest. At any point in time a given mobile worker may have only a half a dozen or so assigned Jobs. Jobs may be assigned at any time, so the mobile worker is encouraged to refresh the list of jobs periodically. Though we learned about how to use SQLite in Chapter 5, we have little need for sharing data between multiple applications and don't need to build out a `ContentProvider`, so the decision has been made to use an XML file stored on the file system to serve as a persistent store of our assigned Job list.

The Field Service Application uses HTTP to exchange data with the home office. Again, PHP is used to build the transactions for exchanging data. While more complex and sophisticated protocols can be employed such as Simple Object Access Protocol (SOAP), this application simply requests an XML file of assigned Jobs and submits an image file representing the captured signature. This architecture is depicted in Figure 12.2.

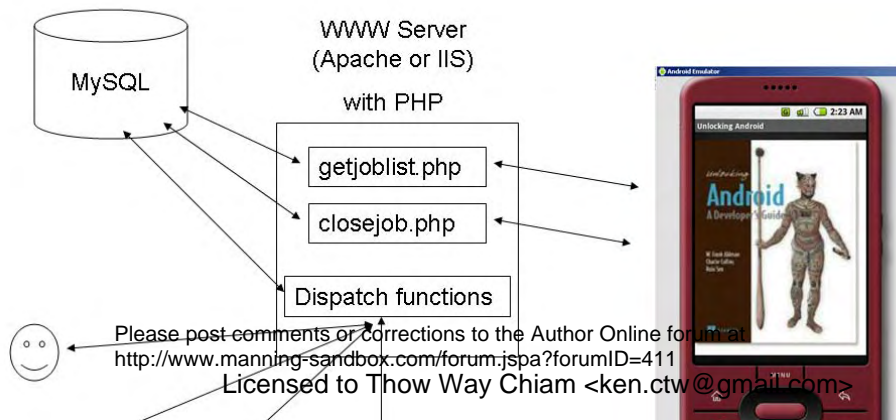


Figure 12.2. Field Service Application and Dispatchers both leverage PHP transactions

The last item to discuss before diving into the code is to discuss configuration. Every mobile worker needs to be identified uniquely. This way, the Field Service Application can retrieve the correct Job list and the dispatchers can properly assign Jobs to workers in the field. Similarly, the mobile application may need to communicate with different servers, depending on locale. For example, a mobile worker in the United States might use a server located in Chicago, but a worker in the United Kingdom may need to use a server in Cambridge. Because of these requirements, it has been decided that both the mobile worker's identifier and the server address need to be readily accessed within the application. Remember, these fields would likely be secured in a deployed application, but for our purposes they are easy to access and not secured.

We have identified the functional requirements, defined the data elements necessary to satisfy those objectives, and selected the preferred deployment platform. It is time to examine the Android application.

12.2 Android Application Tour

Have you ever downloaded an application's source code, excited to get access to all of that code, but once you did, it was a little overwhelming? You want to make your own changes, to put your own spin on the code, but you unzip the file into all of the various subdirectories and you just don't know where to start? Before we jump directly into examining the source code, we need to pay a little attention to the architecture, in particular the flow from one screen to the next.

12.2.1 Application Flow

In this section we will examine the application flow to better understand the relation between the application's functionality, user interface and the classes used to deliver this functionality. Doing this process up front helps ensure that the application delivers the needed functionality and assists in defining which classes we require when it comes time to start coding, which is soon! Figure 12.3 shows the relation between the high level classes in

the application, which are implemented as Android Activities as well as interaction with other services available in Android.

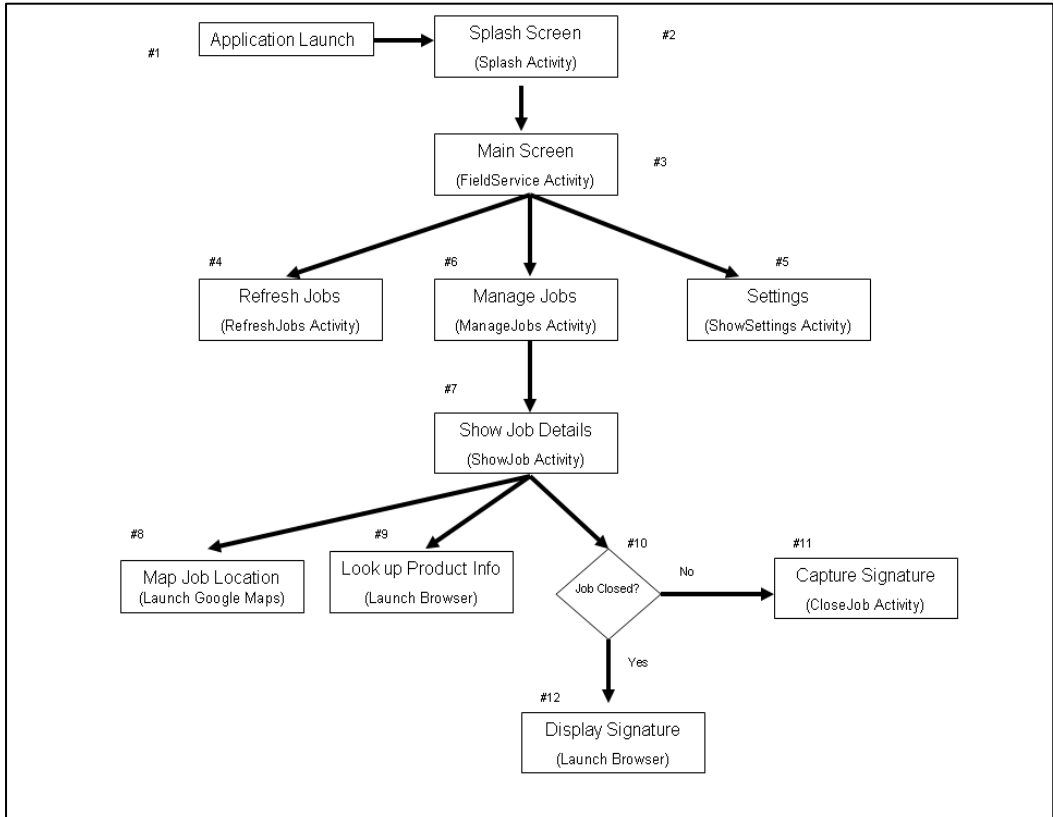


Figure 12.3. Application Flow

The application is selected from the application launch screen on the Android device.

The application splash screen displays. Why? Some applications require setup time to get data structures initialized for example. As a practical matter, such time consuming behavior is discouraged on a mobile device, however it is an important aspect to application design, so it is included in this sample application.

The main screen displays the currently configured User and Server settings, along with three easy-to-hit-with-your-finger buttons.

The Refresh Jobs button initiates a download procedure to fetch the currently available Jobs for this mobile worker from the configured server. The download includes a `ProgressDialog`, which is discussed later in this chapter.

The Settings button brings up a screen which allows the configuration of the User and Server settings.

Selecting Manage Jobs lets our mobile worker review the available Jobs assigned to him and proceed with further steps specific to a chosen Job.

Selecting a Job from the list of Jobs on the Manage Jobs screen brings the Show Job Details screen up with the specific Job information listed. This screen lists the available information about the job and presents three additional buttons.

The Map Job Location button initiates a geo query on the device using an Intent. The default handler for this Intent is the Maps application.

Because our mobile worker may not know much about the product he is being asked to service, each Job includes a product information url. Selecting this button brings up the built-in browser and takes the mobile worker to a (hopefully) helpful Internet resource. This may be an online manual, or perhaps an instructional video.

The behavior of the third button depends on the current status of the Job. If the Job is still "OPEN" this button is used to initiate the close-out or completion of this Job.

When the close procedure is selected, the application presents an empty canvas upon which the customer can take the stylus (assuming a touch screen capable Android device of course!) and sign that the work is complete. A menu choice on that screen presents two options: "Sign & Close" or Cancel. If the Sign-and-Close option is selected, the application submits the signature as a JPEG image to the server and the server marks the Job as "CLOSED". Additionally, the local copy of the Job is marked as closed. The Cancel button causes the Show Job Detail screen to be restored.

If the Job being viewed has already been closed, this third button causes the browser window to be opened to a page displaying the previously captured signature.

Now that we have a pretty good feel for what our requirements are and how we are going to tackle the problem from a functionality and application flow perspective, let's examine the code which delivers this functionality.

12.2.2 Code Road Map

The source code for this application consists of twelve Java source files, one of which is the R.java file, which you will recall is automatically generated based on the resources in the application. This section presents a quick introduction to each of these files. No code is explained yet, we just want to know a little bit about each file and then it will be time to jump into the application, step by step. Table 12.1 lists the source files in the Android Field Service application.

Table 12.1 The source files used to implement the Field Service Application

Source File Name	Description
Splash.java	Activity provides splash screen functionality
ShowSettings.java	Activity provides management of User Name and Server

	URL address
FieldService.java	Activity provides the main screen of the application
RefreshJobs.java	Activity interacts with Server to obtain updated list of Jobs
ManageJobs.java	Activity provides access to list of Jobs
ShowJob.java	Activity provides detailed information on a specific Job such as an address lookup or initiate the signature caption process
CloseJob.java	Activity collects electronic signature and interacts with the Server to upload images and mark Jobs as "CLOSED"
R.java	Automatically generated source file representing identifiers in the resources
Prefs.java	Helper class encapsulating SharedPreferences
JobEntry.java	Class which represents a Job. Includes helpful methods used when passing JobEntry objects from one Activity to another.
JobList.java	Class representing the complete list of JobEntry objects. Includes methods for marshalling and un-marshalling to non-volatile storage.
JobListHandler.java	Class used for parsing XML document containing Job data.

The application also relies on layout resources to define the visual aspect of the user interface. In addition to the layout xml files, an image used by the Splash activity is placed in the drawable subfolder of the res folder along with the stock Android icon image. This icon is used for the home application launch screen. Figure 12.4 depicts the resources used in the application.

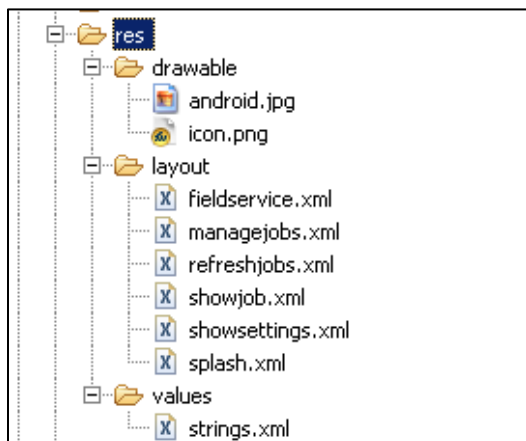


Figure 12.4 Resources used in the application

In an effort to make navigating the code as easy as possible, the resource files are presented in Table 12.2. Note that each of these is clearly seen in Figure 12.4, which is a screen shot from our project open in Eclipse.

Table 12.2. Resource files used in the sample application

File Name	Description
android.jpg	Image used in the Splash Activity.
icon.jpg	Image used in the application launcher.
fieldservice.xml	Layout for main application screen, FieldService Activity
managejobs.xml	Layout for the list of Jobs, ManageJobs Activity
refreshjobs.xml	Layout for the screen shown when refreshing Job list, RefreshJobs Activity.
showjob.xml	Layout for Job detail screen, ShowJob Activity
showsettings.xml	Layout for configuration/settings screen, ShowSettings Activity
splash.xml	Layout for splash screen, Splash Activity
strings.xml	Strings file containing extracted strings. Ideally all text is contained in a strings file for ease of localization. This application's file contains only the application title.

A quick examination of the source files in this application tells us that we have more than one Activity in use. In order to enable navigation between one Activity and the next, our application must inform Android of the existence of these Activity classes. If you recall from Chapter 1, this “registration” step is accomplished with the AndroidManifest.xml file.

12.2.3 AndroidManifest.xml.

Every Android application requires a manifest file to let Android properly “wire things up” when an Intent is handled and needs to be dispatched. Let’s look at the AndroidManifest.xml file used by our application, which is presented in Listing 12.1.

Listing 12.1. The Field Service Application’s AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.UnlockingAndroid">
    <application android:icon="@drawable/icon">
        <activity android:name=".Splash" android:label="@string/app_name">
            1
            <intent-filter>
                2
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER"
            />
            </intent-filter>
        </activity>
```

```

        <activity android:name=".FieldService" >                                3
        </activity>
        <activity android:name=".RefreshJobs" >                                3
        </activity>
        <activity android:name=".ManageJobs" >                                3
        </activity>
        <activity android:name=".ShowJob" >                                    3
        </activity>
        <activity android:name=".CloseJob" >                                    3
        </activity>
        <activity android:name=".ShowSettings" >                                3
        </activity>

    </application>
    <uses-permission android:name="android.permission.INTERNET"></uses-
permission> 4
</manifest>

```

1. The entry point for the application is the Splash Activity class.
2. The Splash class has an intent-filter causing it to be visible in the main application launcher.
3. Additional Activity classes are listed here. Without a presence in the manifest file, they cannot be launched.
4. Because our Field Service Application requires Internet access to communicate with the Home Office, the user-permission android.permission.INTERNET must be present.

12.3.1 Android Code

After a rather long introduction and stage-setting for this chapter, it is finally time to look at the source code for the Field Service Application. The approach is to largely follow the application flow, step by step. So, without further delay, let's start with the Splash screen.

12.3.2 Splash Activity

We are all very familiar with a splash screen for a software application. It acts like a curtain while important things are taking place behind the scenes. Ordinarily splash screens are visible until the application is ready – this could be a very brief amount of time or perhaps much longer in the case where quite a bit of housekeeping is necessary. As a rule, a mobile application should focus on economy and strive to have as little resource consumption as possible. The splash screen in this sample application is meant to demonstrate how such a feature may be constructed - we don't actually need one for housekeeping in this application. But that's ok, we can learn in the process. There are two code snippets of interest to us, the implementation of the Activity as well as the layout file which defines what the user interface looks like. First, examine the layout file in Listing 12.2.

Listing 12.2. splash.xml defines the layout of the application's splash screen.

```

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

```

```

    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <ImageView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:scaleType="fitCenter"
    android:src="@drawable/android"
    />
</LinearLayout>

```

1. The splash.xml layout contains a single `ImageView`, set to fill the entire screen.
2. The source image for this view is defined as the drawable resource named `android`. Note that this is simply the name of the file (minus the file extension) in the drawable folder as shown earlier.

Now we must use this layout in an Activity. Aside from the referencing of an image resource from the layout, this is really not that interesting. Figure 12.5 shows the splash screen running on the Android Emulator.

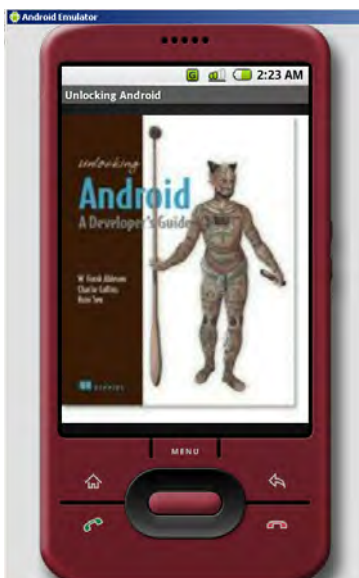


Figure 12.5. Splash screen

What is of interest is the code that creates the splash page functionality. This code is shown in Listing 12.3.

Listing 12.3. Splash.java implements splash screen functionality.

```
package com.msi.manning.UnlockingAndroid;

// multiple imports omitted for brevity, see full source code

public class Splash extends Activity
{
    @Override
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);

        setContentView(R.layout.splash);
        Handler x = new Handler();
        x.postDelayed(new splashhandler(), 2000);
    }
    class splashhandler implements Runnable
    {
        public void run()
        {
            startActivity(new Intent(getApplicationContext(),FieldService.class));
            Splash.this.finish();
        }
    }
}
```

1. Associate the splash layout with this Activity's View.
2. Setup a Handler which is used to close down the splash screen after a period of time. Note that the arguments to the `postDelayed` method are a class which implements the `Runnable` interface and time in milliseconds. In this snippet of code, the screen will be shown for 2000 milliseconds, i.e. two seconds.
3. The class `splashhandler` is invoked when it is time for the splash screen to close.
4. The `FieldService` Activity is instantiated with a call to `startActivity`. Note that an `Intent` is not used here - we explicitly specify which class is going to satisfy our request.
5. Terminate the `Splash` Activity with a call to `finish()`.

The `Splash` screen is happily entertaining our mobile worker each time he starts the application. Let's move on to the main screen of the application.

12.3.3 FieldService Activity, Part 1

The goal of the `FieldService` Activity is to put the functions the mobile worker requires directly in front of him and make sure they are easy to access. A good mobile application is

often one that can be used with one hand, such as the five way navigation buttons, or in some case a thumb tapping on a button. Additionally, if there is helpful information to display, do not hide it. For example, it is helpful for our mobile worker to know that he is configured to obtain Jobs from a particular server. Figure 12.6 demonstrates the Field Service application conveying a very simple, yet easy to use "home screen".

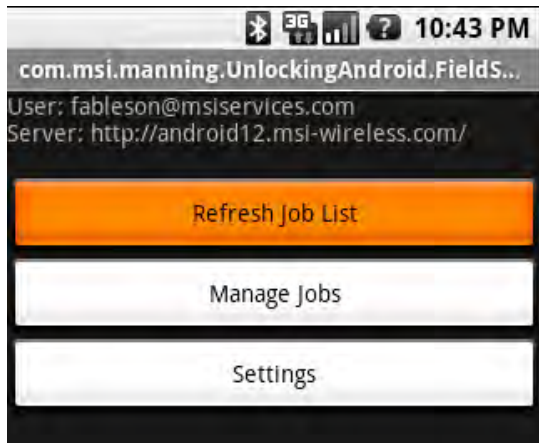


Figure 12.6. The home screen. Less is more.

Before reviewing the code in `FieldService.java`, we need to take a quick break to discuss how the User and Server settings are managed. This is important because these settings are used throughout the application and as we saw in the `fieldservice.xml` layout file, we need to access those values to display to our mobile worker on the home screen.

12.3.3.1 PREFS CLASS

As you learned in Chapter 5, there are a number of means for managing data. Because we need to persist this data across multiple invocations of our application, the data must be stored in a non-volatile fashion. This application employs private `SharedPreferences` to accomplish this. Why? Well, despite the fact that we are largely ignoring security for this sample application, using private `SharedPreferences` means that other applications cannot casually access this potentially important data. For example, we presently only use an identifier (let's call it an email address for simplicity) and a server url in this application. However, we might also include a password or a pin in a production-ready application, so keeping this data private is a good practice.

The `Prefs` class can be described as a helper or wrapper class. This class wraps the `SharedPreferences` code and exposes simple getter and setter methods, specific to this application. This implementation knows something about what we are trying to accomplish, so it adds value with some default values as well. Let's have a quick look at Listing 12.5 to see how our `Prefs` class is implemented.

Listing 12.5 Prefs class provides storage and retrieval for small and useful information

```
package com.msi.manning.UnlockingAndroid;

// multiple imports omitted for brevity, see full source code

public class Prefs
{
    private SharedPreferences _prefs = null;           1
    private Editor _editor = null;                   2
    private String _useremailaddress = "Unknown";    3
    private String _serverurl = "http://android12.msi-
wireless.com/getjoblist.php";                       3

    public Prefs(Context context)                     4
    {
        _prefs =
context.getSharedPreferences("PREFS_PRIVATE",Context.MODE_PRIVATE);
        _editor = _prefs.edit();

    }
    public String getValue(String key,String defaultvalue)    5
    {
        if (_prefs == null) return "Unknown";
        return _prefs.getString(key,defaultvalue);
    }
    public void setValue(String key,String value)             5
    {
        if (_editor == null) return;
        _editor.putString(key,value);
    }
    public String getEmail()                                  6
    {
        if (_prefs == null) return "Unknown";
        _useremailaddress = _prefs.getString("emailaddress","Unknown");
        return _useremailaddress;
    }
    public void setEmail(String newemail)                    7
    {
        if (_editor == null) return;
        _editor.putString("emailaddress",newemail);
    }
    ... (abbreviated for brevity)
    public void save()                                       8
    {
        if (_editor == null) return;
        _editor.commit();
    }
}
```

1. `_prefs` is a private reference to a `SharedPreferences` object
2. `_editor` is used to manipulate data when setting values as seen further on in this listing.
3. We have default values for User and Server values. Note how these are specific to our application. If we were to re-use this `Prefs` class, likely through the tried and true copy-paste reuse pattern, we would customize it to properly reflect the data elements of interest.

4. The `Prefs()` constructor does the necessary housekeeping so we can establish our private `SharedPreferences` object, including using a passed-in `Context` instance. The `Context` class is necessary because the `SharedPreferences` mechanism relies upon a `Context` for segregating data.
5. Even though we have tailored this class to be specific to this application, there also exists “generic” `getValue()` and `setValue()` methods.
6. `getEmail()` and `getServer()` access the `_prefs` instance directly to read data.
7. `setEmail()` and `setServer()` employ the `Editor` instance, `_editor`, to store values with the `putString()` method.
8. The `save()` method invokes a `commit()` on the `Editor` which persists the data to the `SharedPreferences` store.

Now that we have some feel for how this important preference data is stored, let's return to examine the code of `FieldService.java`.

12.3.4 *FieldService Activity, Part 2*

Recall that the `FieldService.java` file implements the `FieldService` class which is essentially the “home screen” of our application. This code does the primary dispatching for the application. Much of the programming techniques in this file have been seen earlier in the book, however please take note of the use of `startActivityForResult` and the `onActivityResult` methods as you read through the code, as seen in Listing 12.6.

Listing 12.6 `FieldService.java` implements `FieldService` Activity

```
package com.msi.manning.UnlockingAndroid;

// multiple imports trimmed for brevity, see full source code

public class FieldService extends Activity
{
    final int ACTIVITY_REFRESHJOBS = 1;
    final int ACTIVITY_LISTJOBS = 2;
    final int ACTIVITY_SETTINGS = 3;
    Prefs myprefs = null;
    @Override
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.fieldservice);
        myprefs = new Prefs(this.getApplicationContext());
        RefreshUserInfo();
        final Button refreshjobsbutton = (Button)
findViewById(R.id.getjobs);
        refreshjobsbutton.setOnClickListener(new Button.OnClickListener()
        {
            public void onClick(View v)
            {
                try
```



```

        {
            startActivityForResult(new
Intent(v.getContext(),RefreshJobs.class),ACTIVITY_REFRESHJOBS); 6
        }
        catch (Exception e)
        {
        }
    }
    });
    // show jobs
    // similar to refresh jobs code
    // settings
    // similar to refresh jobs code
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data) 7
{
    switch (requestCode)
    {
        case ACTIVITY_REFRESHJOBS:
            break;
        case ACTIVITY_LISTJOBS:
            break;
        case ACTIVITY_SETTINGS:
            RefreshUserInfo(); 8
            break;
    }
}
private void RefreshUserInfo() 9
{
    try
    {
        final TextView emaillabel = (TextView)
findViewById(R.id.emailaddresslabel);
        emaillabel.setText("User: " + myprefs.getEmail() + "\nServer: "
+ myprefs.getServer() + "\n");
    }
    catch (Exception e)
    {
    }
}
}

```

1. Three constants are defined for use with the `startActivityForResult()` and `onActivityResult()` call/return construct.
2. Define an instance of our `Prefs` class.
3. Connect the user interface defined in `fieldservice.xml` with this Activity.
4. Initialize our `Prefs` class instance, `myprefs`.
5. A call to `RefreshUserInfo()` updates the display with the currently stored User and Server configuration values.

6. Connect an instance of a Button to handle the RefreshJobs functionality. Note the use of `startActivityForResult()`, in particular, the last parameter. The code is omitted for brevity, but the other Activities are launched in a similar fashion.
7. When an Activity which was invoked by `startActivityForResult` completes, the method `onActivityResult` is invoked. Data may be passed back within the returning Intent, named data.
8. When the Settings Activity completes, we call `RefreshUserInfo()`. This way we can update the display to reflect any changes made by the user when changing the settings.
9. `RefreshUserInfo()` updates the display by getting a reference to the `TextView` and setting its text with `setText()`.

Because the settings are so important to this application, the next section covers the management of the user and server values.

12.3.5 Settings

When the 'Settings' button is selected from the main application screen an Activity is started which allows the user to configure their User Id (email address) and the Server url. The screen layout is very basic, as seen in Listing 12.7 and shown graphically in Figure 12.7.

Listing 12.7 showsettings.xml contains the user interface elements for the Settings screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Email Address"
    />
    <EditText
        android:id="@+id/emailaddress"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:autoText="true"
    />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Server URL"
    />
    <EditText
        android:id="@+id/serverurl"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:autoText="true"
```

```

/>
<Button android:id="@+id/settingsave"
        android:text="Save Settings"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:enabled="true"
        />
</LinearLayout>

```

3

1. A TextView and EditText combine to allow entry of the User ID/Email Address.
2. A TextView and EditText combine to allow entry of the Server address.
3. A Button is used to request that the new settings be saved.

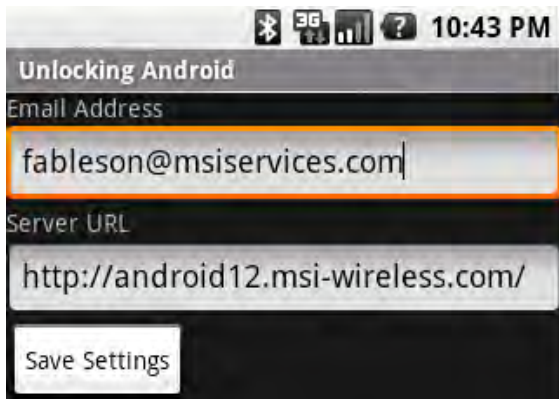


Figure 12.7. Settings Screen in use.

The source code behind the Settings screen is also very basic. Note the use of the `PopulateScreen()` method which makes sure the `EditView` controls are populated with the current values stored in the `SharedPreferences`. Note also the use of the `Prefs` helper class to retrieve and then save the values, as seen in Listing 12.8

Listing 12.8. ShowSettings.java implements the code behind the settings screen

```
package com.msi.manning.UnlockingAndroid;
```

```

// multiple imports trimmed for brevity, see full source code

public class ShowSettings extends Activity
{
    Prefs myprefs = null;
    @Override
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.showsettings);           1
        myprefs = new Prefs(this.getApplicationContext()); 2
        PopulateScreen();                                 3
        final Button savebutton = (Button) findViewById(R.id.settingssave);
        savebutton.setOnClickListener(new Button.OnClickListener() 4
    {
        public void onClick(View v)
        {
            try
            {
                final EditText email = (EditText)
findViewById(R.id.emailaddress);           5
                if (email.getText().length() == 0)
                {
                    // display dialog, see full source code
                    return;
                }

                final EditText serverurl = (EditText)
findViewById(R.id.serverurl);           5
                if (serverurl.getText().length() == 0)
                {
                    // display dialog, see full source code
                    return;
                }
                myprefs.setEmail(email.getText().toString()); 6
                myprefs.setServer(serverurl.getText().toString()); 6
                myprefs.save();
                finish();                                     7
            }
            catch (Exception e)
            {
            }
        }
    });
}

private void PopulateScreen()                8
{
    try
    {
        final EditText emailfield = (EditText)
findViewById(R.id.emailaddress);
        final EditText serverurlfield = (EditText)
findViewById(R.id.serverurl);
        emailfield.setText(myprefs.getEmail());
        serverurlfield.setText(myprefs.getServer());
    }
    catch Exception e)

```

```

    }
}

```

1. Connect this Activity to its layout, using the identifier in the automatically generated `R` class.
2. Initialize an instance of the `Prefs` class.
3. Request the screen to be populated by calling the private method `PopulateScreen()`.
4. Wire up the Button's listener method.
5. Perform some very basic data entry validation
6. Using the `Prefs` helper class, set the values for the email address and the server url.
7. Call `finish()` to end this Activity.
8. The `PopulateScreen()` method loads the `EditText` fields, ready for user input.

Once the settings are in order, it is time to focus on the core of the application, managing Jobs for our mobile worker. In order to get the most out of looking at the higher level functionality of downloading (Refreshing) and managing Jobs, we need to take a quick look at the core data structures in use in this application.

12.3.6 Data Structures

Data structures represent a key element of any software project and in particular, one consisting of multiple tiers, such as this field service application. Job data is exchanged between an Android application and the server, so the elements of the Job are central to our application. In Java, we implement these data structures as classes, which include helpful methods in addition to the data elements. XML data shows up in many locations in this application, so we will start there.

12.3.6.1 JOB XML DESCRIPTION

The same xml format is used as persistent storage by the Android application and for the transmission of Job data from the server to Android. There is nothing particularly fancy in the xml document structure, just a collection of Jobs.

Perhaps the most straight-forward means of describing an XML document is through a Document Type Definition, or DTD. The DTD representing the XML used in this application is shown in Listing 12.9.

Listing 12.9 DTD for joblist.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT joblist ((job+))>
1
  <!ELEMENT job ((id, status, customer, address, city, state, zip, product,
producturl, comments))>
2
  <!ELEMENT zip (#PCDATA)>

```

```

<!ELEMENT status (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT producturl (#PCDATA)>
<!ELEMENT product (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT customer (#PCDATA)>
<!ELEMENT comments (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT address (#PCDATA)>

```

1. The joblist is the top level of the xml file and contains one or more job elements.
2. The job element contains a number of simple data fields.

Listing 12.10 shows a sample xml document containing a joblist with a single job entry.

Listing 12.10 XML document containing data for Field Service Application

```

<?xml version="1.0" encoding="UTF-8" ?>
<joblist>
<job>
<id>22</id>
<status>OPEN</status>
<customer>Big Tristan's Imports</customer>
<address>2200 East Cedar Ave</address>
<city>Flagstaff</city>
<state>AZ</state>
<zip>86004</zip>
<product>UnwiredTools UTCIS-PT</product>
<producturl>http://unwiredtools.com</producturl>
<comments>Requires tuning - too rich in the mid range RPM. Download
software from website before visiting.</comments>
</job>
</joblist>

```

Now that we have a feel for what the Job data looks like, we need to see how the data is handled in our Java classes.

12.3.6.2 JOBENTRY

The individual Job is used throughout the application and is therefore essential to understand. In our application, we define the JobEntry class to manage the individual Job, shown in listing 12.11.

Listing 12.11 JobEntry.java

```

package com.msi.manning.UnlockingAndroid;

import android.os.Bundle;           1

public class JobEntry
{
    private String _jobid="";        2
    private String _status = "";
    private String _customer = "";
    private String _identifier = "";
    private String _address = "";

```

```

private String _city = "";
private String _state = "";
private String _zip = "";
private String _product = "";
private String _producturl = "";
private String _comments = "";

    JobEntry()
    {
    }
    // get/set methods omitted for brevity
    public String toString() 3
    {
        return this._jobid + ": " + this._customer + ": " +
this._product;
    }
    public String toXMLString() 4
    {
        StringBuilder sb = new StringBuilder("");
        sb.append("<job>");
        sb.append("<id>" + this._jobid + "</id>");
        sb.append("<status>" + this._status + "</status>");
        sb.append("<customer>" + this._customer + "</customer>");
        sb.append("<address>" + this._address + "</address>");
        sb.append("<city>" + this._city + "</city>");
        sb.append("<state>" + this._state + "</state>");
        sb.append("<zip>" + this._zip + "</zip>");
        sb.append("<product>" + this._product + "</product>");
        sb.append("<producturl>" + this._producturl + "</producturl>");
        sb.append("<comments>" + this._comments + "</comments>");
        sb.append("</job>");
        return sb.toString() + "\n";
    }

    public Bundle toBundle() 5
    {
        Bundle b = new Bundle();
        b.putString("jobid", this._jobid);
        b.putString("status", this._status);
        b.putString("customer", this._customer);
        b.putString("address", this._address);
        b.putString("city", this._city);
        b.putString("state", this._state);
        b.putString("zip", this._zip);
        b.putString("product", this._product);
        b.putString("producturl", this._producturl);
        b.putString("comments", this._comments);
        return b;
    }
    public static JobEntry fromBundle(Bundle b) 6
    {
        JobEntry je = new JobEntry();
        je.set_jobid(b.getString("jobid"));
        je.set_status(b.getString("status"));
        je.set_customer(b.getString("customer"));
        je.set_address(b.getString("address"));
        je.set_city(b.getString("city"));

```

```

        je.set_state(b.getString("state"));
        je.set_zip(b.getString("zip"));
        je.set_product(b.getString("product"));
        je.set_producturl(b.getString("producturl"));
        je.set_comments(b.getString("comments"));
        return je;
    }
}

```

1. This application relies heavily upon the `Bundle` class for moving data from one Activity to another. This will be explained in more detail later in this chapter.
2. A `String` member exists for each element in the job. For example, `jobid`, `customer`, etc.
3. The `toString()` method is rather important as it is used when displaying Jobs in the `ManageJobs` Activity, discussed later in the chapter.
4. The `toXML()` method generates an XML representation of this `JobEntry`, complying to the `job` element defined in the previously presented DTD.
5. The `toBundle()` method takes the data members of the `JobEntry` class and packages them into a `Bundle`. This `Bundle` is then able to be passed between Activities.
6. The `fromBundle()` static method returns a `JobEntry` when provided with a `Bundle`. The `toBundle()` and `fromBundle()` work together to assist in the passing of `JobEntry` objects (at least the data portion thereof) between Activities.

After understanding the `JobEntry` class, we need to look at the `JobList` class, which is a class used to manage a collection of `JobEntry` objects.

12.3.6.3 JOBList

When interacting with the server or presenting the available Jobs to manage on the Android device, the field service application works with an instance of the `JobList` class. This class, like the `JobEntry` class, has both data members and helpful methods. The `JobList` class contains a typed `List` data member, which is actually implemented using a `Vector`. This is the only data member of this class, as seen in Listing 12.12. The methods of interest are described in the listing.

Listing 12.12 JobList.java code listing.

```

package com.msi.manning.UnlockingAndroid;

import java.util.List;
import org.xml.sax.InputSource;
import android.util.Log;
// additional imports omitted for brevity, see source code

public class JobList
{
    private Context _context = null;
    private List<JobEntry> _joblist;
    JobList(Context context)
    {

```



```

        _context = context;
        _joblist = new Vector<JobEntry>(0);
    }
    int addJob(JobEntry job)                                5
    {
        _joblist.add(job);
        return _joblist.size();
    }
    JobEntry getJob(int location)                            5
    {
        return _joblist.get(location);
    }
    List<JobEntry> getAllJobs()                              6
    {
        return _joblist;
    }
    int getJobCount()
    {
        return _joblist.size();
    }
    void replace(JobEntry newjob)                            7
    {
        try
        {
            JobList newlist = new JobList();
            for (int i=0;i<getJobCount();i++)
            {
                JobEntry je = getJob(i);
                if (je.get_jobid().equals(newjob.get_jobid()))
                {
                    newlist.addJob(newjob);
                }
                else
                {
                    newlist.addJob(je);
                }
            }
            this._joblist = newlist._joblist;
            persist();
        }
        catch (Exception e)
        {
        }
    }
    void persist()                                          8
    {
        try
        {
            FileOutputStream fos =
_context.openFileOutput("chapter12.xml", Context.MODE_PRIVATE);
            fos.write("<?xml version=\\"1.0\\" encoding=\\"UTF-8\\"
?>\n".getBytes());
            fos.write("<joblist>\n".getBytes());
            for (int i=0;i<getJobCount();i++)
            {
                JobEntry je = getJob(i);
                fos.write(je.toXMLString().getBytes());
            }
        }
    }

```

```

        }
        fos.write("</joblist>\n".getBytes());
        fos.flush();
        fos.close();
    }
    catch (Exception e)
    {
        Log.d("CH12",e.getMessage());
    }
}
static JobList parse(Context context)
{
    try
    {
        FileInputStream fis =
context.openFileInput("chapter12.xml");

        if (fis == null)
        {
            return null;
        }
        InputSource is = new InputSource(fis);
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        XMLReader xmlreader = parser.getXMLReader();
        JobListHandler jlHandler = new JobListHandler(null /* no
progress updates when reading file */);
        xmlreader.setContentHandler(jlHandler);
        xmlreader.parse(is);
        fis.close();
        return jlHandler.getList();
    }
    catch (Exception e)
    {
        return null;
    }
}
}

```

1. The list of jobs is implemented as a Vector, which is a type of List.
2. The XML structure containing Job information is parsed with the SAX parser, so we need to be sure to import those packages.
3. This class, as other classes in this application, relies on the android.util.Log class to assist in debugging the application by outputting trace data to the Log.
4. JobEntry objects are stored in the typed List object named _joblist.
5. addJob() and getJob() are helper methods for managing the list.
6. getAllJobs() returns the list of JobEntry items. Note that generally speaking the application uses the getJob() method for individual JobEntry management, however the getAllJobs() method is particularly useful when we get to displaying the full list of Jobs in the ManageJobs Activity, discussed later in this chapter.

7. The `replace()` method is used when we have closed a `Job` and need to update our local store of `Jobs`. Note that after it has updated the local list of `JobEntry` items, it calls the `persist()` method.

8. The `persist()` method is responsible for writing out to the file system an XML representation of the entire list of `JobEntry` items. Note that this method invokes the `toXMLString()` method on each `JobEntry` in the list. The `openFileOutput` method creates a file within the application's private file area. This is essentially a helper method to ensure we get a file path to which we have full read/write privileges.

9. The `android.util.Log` class's static methods are useful for displaying debug information, as seen here in the exception handler. In a production application, additional code would be necessary in the catch block to properly recover from an error.

10. The `parse` method obtains an instance of a `FileInputStream` to gain access to the file, and then creates an instance of an `InputStream` which is required by the SAX XML parser. In particular, take note of the `JobListHandler`. SAX is a "callback" parser, meaning that it invokes a user-supplied method to process events in the parsing process. It is up to the `JobListHandler` (in our example) to process the data as appropriate.

We have one more class to go before we can jump back to the higher level functionality of our application. The next section takes a quick tour of the `JobListHandler`, which is responsible for putting together a `JobList` from an XML data source.

12.3.6.4 JOBLISTHANDLER

As presented already, our application uses an XML data storage structure. This XML data can come from either the Server, or from a local file on the file system. In either case, the application must parse this data and transform it into a useful form. This is accomplished through the use of the SAX XML parsing engine and the `JobListHandler`, which is shown in Listing 12.13. The `JobListHandler` is used by the SAX parser for our XML data, regardless of the data's source. Depending on where the data comes from dictates how the SAX parser is setup and invoked in this application. The `JobListHandler` behaves slightly different depending on whether the class's constructor includes a `Handler` argument or not. If the `Handler` is provided, the `JobListHandler` will pass messages back for use in a `ProgressDialog`. If the `Handler` argument is null, this status message passing is bypassed. When parsing data from the server, the `ProgressDialog` is employed, however the parsing of a local file is done quickly and without user feedback. The rationale for this is simple – the network connection may be slow and we need to show progress information to the user. An argument could be made for always showing the progress of the parse operation, however this approach gives us opportunity to demonstrate some more conditionally operating code.

Listing 12.13 JobListHandler.java

```
package com.msi.manning.UnlockingAndroid;
```

```

// multiple imports omitted for brevity, see full source code

public class JobListHandler extends DefaultHandler
{
    Handler phandler = null;
    JobList _list;
    JobEntry _job;
    String _lastElementName = "";
    StringBuilder sb = null;
    Context _context;

    JobListHandler(Context c, Handler progresshandler)           1
    {
        _context = c;
        if (progresshandler != null)
        {
            phandler = progresshandler;                          2
            Message msg = new Message();
            msg.what = 0;
            msg.obj = (Object)("Processing List");
            phandler.sendMessage(msg);
        }
    }

    public JobList getList()                                     3
    {
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object)("Fetching List");
        if (phandler != null) phandler.sendMessage(msg);
        return _list;
    }

    public void startDocument() throws SAXException              4
    {
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object)("Starting Document");
        if (phandler != null) phandler.sendMessage(msg);
        _list = new JobList(_context);
        _job = new JobEntry();
    }

    public void endDocument() throws SAXException                5
    {
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object)("End of Document");
        if (phandler != null) phandler.sendMessage(msg);
    }

    public void startElement(String namespaceURI, String localName, String
qName, Attributes atts) throws SAXException
    {
        try
        {
            sb = new StringBuilder("");                          6
            if (localName.equals("job"))

```

```

        {
            Message msg = new Message();
            msg.what = 0;
            msg.obj = (Object)(localName);
            if (phandler != null) phandler.sendMessage(msg);
            _job = new JobEntry();
        }
    }
    catch (Exception ee)
    {
    }
}

public void endElement(String namespaceURI, String localName, String
qName) throws SAXException
{
    if (localName.equals("job"))
    {
        // add our job to the list!
        _list.addJob(_job);
        Message msg = new Message();
        msg.what = 0;
        msg.obj = (Object)("Storing Job # " + _job.get_jobid());
        if (phandler != null) phandler.sendMessage(msg);
        return;
    }
    // portions of the code omitted for brevity
}

public void characters(char ch[], int start, int length)
{
    String theString = new String(ch,start,length);
    Log.d("CH12","characters[" + theString + "]");
    sb.append(theString);
}
}

```

1. The `JobListHandler` constructor takes a single argument of a `Handler`. This value may be null. If null, Message passing is omitted from operation. When reading from a disk file, this `Handler` argument is null. When reading data from the Server over the Internet, with a potentially slow connection, the Message passing code is utilized to provide feedback for the user in the form of a `ProgressDialog`. The `ProgressDialog` code is seen later in this chapter in the discussion of the `RefreshJobs` Activity.
2. A local copy of the `Handler` is setup when using the `ProgressDialog` as described in #1.
3. The `getList()` method is invoked when parsing is complete to return a copy of the `JobList` which was constructed during the parse process.
4. When the `startDocument()` callback method is invoked by the SAX parser, the initial class instances are established.
5. The `endDocument()` method is invoked by the SAX parser when all of the document has been consumed. This is an opportunity for the `Handler` to perform additional clean-up, as necessary. In our example, a message is posted to the user by sending a `Message`.

6. For each element in the XML file, the SAX parser follows the pattern: `startElement` is invoked, `characters()` is invoked (one or more times) and then `endElement` invoked. In the `startElement` method, we initialize our `StringBuilder` and evaluate the element name. If the name is "job", we initialize our class-level `JobEntry` instance.

7. In the `endElement()` method, the element name is evaluated. If the element name is "job", the `JobListHandler` adds this `JobEntry` to the `JobList` data member, `_joblist` with a call to `addJob()`. Also in the `endElement()` method, the data members of the `JobEntry` instance (`_job`) are updated. See full source code.

8. The `characters()` method is invoked by the SAX parser whenever data is available for storage. The `JobListHandler` simply appends this string data to a `StringBuilder` instance each time it is called. It is possible that the `characters` method is invoked more than once for a particular element's data. That is the rationale behind using a `StringBuilder` instead of a single `String` variable.

After this lengthy but important look into the data structures and the accompanying explanations, it is time to return to the higher level functionality of the application.

12.4 *Digging Deeper Into the Code*

Most of the time our mobile worker is using this application, he will be reading through comments, looking up a Job address, getting product information and the other aspects of working on a specific Job. However, without a list of Jobs to work on, our mobile worker will be sitting idle, not earning a dime! Therefore, the first thing to review is the fetching of new Jobs. This is also a good time to discuss gathering the list of Jobs, coming on the heels of the review of the `JobListHandler`.

12.4.1 *RefreshJobs*

The Refresh Jobs Activity performs a simple yet vital role in the field service application. Whenever requested, the RefreshJobs Activity attempts to download a list of new Jobs from the server. The user interface is super simple – just a blank screen with a `ProgressDialog` informing the user of the application's progress, as seen in Figure 12.8.

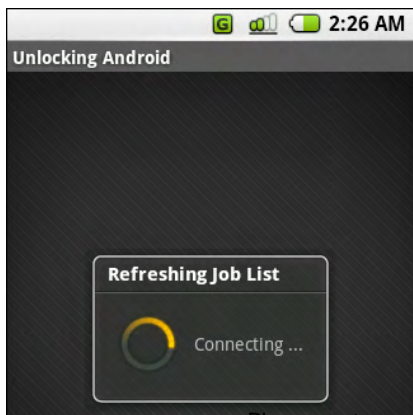


Figure 12.8. The ProgressDialog in use during RefreshJobs.

The code listing for RefreshJobs is shown in Listing 12.14. The code is rather straightforward, as most of the heavy lifting is done in the JobListHandler. This code's responsibility is to fetch configuration settings, initiate a request to the Server and put a mechanism in place for showing progress to the user.

Listing 12.14. RefreshJobs.java

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source

public class RefreshJobs extends Activity
{
    Prefs myprefs = null;
    Boolean bCancel = false;
    JobList mList = null;
    ProgressDialog progress;
    Handler progresshandler;
    @Override
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.refreshjobs);
        myprefs = new Prefs(this.getApplicationContext());
        myprogress = ProgressDialog.show(this, "Refreshing Job List",
        "Please Wait",true,false);
        progresshandler = new Handler()
        {
            @Override
            public void handleMessage(Message msg)
            {
                switch (msg.what)
                {
                    case 0:
                        myprogress.setMessage("" + (String) msg.obj);
                        break;
                    case 1:
                        myprogress.cancel();
                        finish();
                        break;
                    case 2: // error occurred
                        myprogress.cancel();
                        finish();
                        break;
                }
                super.handleMessage(msg);
            }
        };
        Thread workthread = new Thread(new DoReadJobs());
        workthread.start();
    }
}
```

```

    }
    class DoReadJobs implements Runnable
    {
        public void run()
        {
            InputSource is = null;
            Message msg = new Message();
            msg.what = 0;
            try
            {
                // Looper.prepare();
                msg.obj = (Object) ("Connecting ...");
                progresshandler.sendMessage(msg);
                URL url = new URL(myprefs.getServer() +
"getjoblist.php?identifier=" + myprefs.getEmail());
                is = new InputSource(url.openStream());
                SAXParserFactory factory = SAXParserFactory.newInstance();
                SAXParser parser = factory.newSAXParser();
                XMLReader xmlreader = parser.getXMLReader();
                JobListHandler jlHandler = new
JobListHandler(progresshandler);
                xmlreader.setContentHandler(jlHandler);
                msg = new Message();
                msg.what = 0;
                msg.obj = (Object) ("Parsing ...");
                progresshandler.sendMessage(msg);
                xmlreader.parse(is);
                msg = new Message();
                msg.what = 0;
                msg.obj = (Object) ("Parsing Complete");
                progresshandler.sendMessage(msg);
                msg = new Message();
                msg.what = 0;
                msg.obj = (Object) ("Saving Job List");
                progresshandler.sendMessage(msg);
                jlHandler.getList().persist();
                msg = new Message();
                msg.what = 0;
                msg.obj = (Object) ("Job List Saved.");
                progresshandler.sendMessage(msg);
                msg = new Message();
                msg.what = 1;
                progresshandler.sendMessage(msg);
            } catch (Exception e)
            {
                Log.d("CH12", "Exception: " + e.getMessage());
                msg = new Message();
                msg.what = 2;
                msg.obj = (Object) ("Caught an error retrieving Job data: "
+ e.getMessage());
                progresshandler.sendMessage(msg);
            }
        }
    }
}

```


1. Declaration of a `ProgressDialog`, used to display progress information to the user. There are a number of ways to display progress in Android. This is perhaps the most straight-forward approach.
2. Declaration of a `Handler` used to process `Message` instances. While the `Handler` itself is defined as an anonymous class, the code requires a reference to it for passing to the `JobListHandler` when parsing.
3. The user identifier and server url address are accessible via the `Prefs` class.
4. The `ProgressDialog` is instantiated. The arguments are: Context, Title of Dialog, Initial Textual Message, Indeterminate and Cancelable. Using true for the Indeterminate parameter means that we are not providing any clue as to when the operation will complete such as percentage remaining, just an indicator that something is still happening.
5. A new `Handler` is defined to process messages sent from the parsing routine which is introduced momentarily. An important class which has been mentioned but thus far not described is the `Message` class. This class is used to convey information between different threads of execution. The `Message` class has some generic data members which may be used in a flexible manner. The first member of interest is the "what" member, which acts as a simple identifier, allowing recipients to easily jump to desired code based on the value of the what member. The most typical (and used here) approach is to evaluate the what data member via a switch statement.
6. In this application, a `Message` received with its what member equal to 0 represents a textual update message to be displayed on the `ProgressDialog`. The textual data itself is passed as a `String` casted to an `Object` and stored in the `obj` data member of the `Message`. This interpretation of the what member is purely arbitrary. We could have used 999 as the value meaning textual update, for example.
7. A what value of 1 or 2 indicates that the operation is complete and this `Handler` can take steps to initiate another thread of execution. For example, a value of 1 indicates successful completion so the `ProgressDialog` is canceled (dismissed would work here also) and the `RefreshJobs` Activity is completed with a call to `finish()`. The value of 2 for the what member has the same function as a value of 1, but is provided here as an example of handling different result conditions, for example a failure. In a production-ready application, this step should be fleshed out to perform an additional step of instruction to the user and/or a retry step.
8. Any `Message` not explicitly handled by the `Handler` instance should be passed along to the `super` class. In this way system messages may be processed.
9. When communicating with a remote resource such as a remote web server in our case, it is a good idea to perform the communications steps in a thread other than the primary GUI

thread. A new Thread is created based on the DoReadJobs class, which implements the Runnable Java interface.

10. A new Message object is instantiated and initialized. This step takes place over and over throughout the run method of the DoReadJobs class. It is important to not re-use a Message object as they are literally passed and enqueued. It is possible for them to "stack up" in the receiver's queue so re-using a Message object will lead to missing or corrupt data at best and Thread synchronization issues or beyond at worst.

11. Why are we talking about a commented out line of code? Great question – because it caused so much pain in the writing of this application! A somewhat odd and confusing element of Android programming is the Looper class. This class provides static methods to assist Java Threads to interact with Android. Threads by default do not have a "message loop", so presumably Messages just don't go anywhere when "sent". The first call to make is `Looper.prepare()` which creates a Looper for a Thread which does not already have one established. Then by placing a call to the `loop()` method, the flow of Messages takes place. Prior to implementing this class as a Runnable interface, I experimented with performing this step in the same thread and attempted to get the ProgressDialog to work properly. All this said, if you run into funny thread/Looper messages on the Android Emulator, have a look at adding a call to `Looper.prepare()` at the beginning of your Thread and then `Looper.loop()` to help Messages flow.

12. An example of using a Message to communicate back to our handler. These Messages carry updates for the ProgressDialog.

13. Use the `myprefs` instance to help construct a URL to fetch jobs.

14. Create a new `InputSource` from the URL stream. This is required for the SAX parser. This method reads data from the network directly into the parser without a temporary storage file.

15. Note that the instantiation of the `JobListHandler` takes a reference to the `progresshandler`. This way the `JobListHandler` can (optionally) propagate messages back to the user during the parse process.

16. After the parse is complete, the `JobListHandler` returns a `JobList` object which is then persisted to store the data to the local file system.

17. This parsing step is complete, so let the Handler know by passing a Message with the what field set to a value of 1.

18. If an exception occurs, pass a message with what set to 2, indicating an error.

Congratulations, your Android application has now constructed a URL with persistently stored configuration information (user and server), and successfully connected over the Internet to fetch XML data. That data has been parsed into a `JobList` containing

`JobEntry` objects, whilst providing our patient mobile worker with feedback and then subsequently storing the `JobList` to the file system for later use. Now we want to work with those `Jobs`, because after all, those `Jobs` have to be completed for our mobile worker friend to make a living!

12.4.2 *ManageJobs*

The `ManageJobs` Activity presents a scrollable list of `Jobs` for review and action. At the top of the screen is a simple summary indicating the number of `Jobs` in the list and each individual `Job` is enumerated in a `ListView`.

If you recall earlier, we mentioned the importance of the `JobEntry`'s `toString()` method:

```
public String toString()
{
    return this._jobid + ": " + this._customer + ": " +
this._product;
}
```

This method generates the `String` which is used to represent the `JobEntry` in the `ListView`, as shown in Figure 12.9.

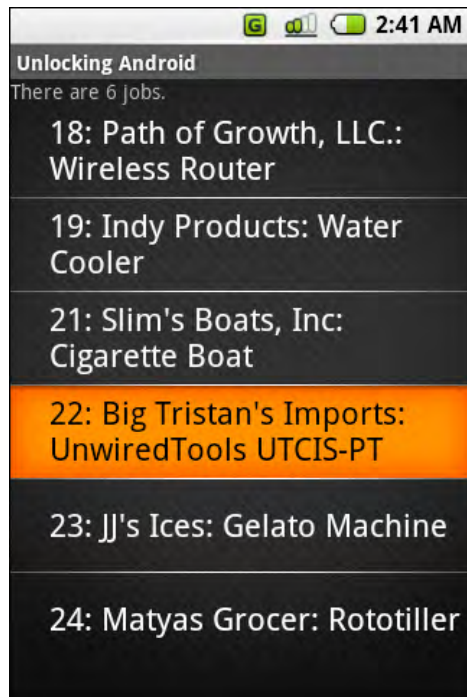


Figure 12.9. *Mange Jobs* Activity lists downloaded `Jobs`.

The layout for this Activity's View is rather simple, just a TextView and a ListView, as seen in Listing 12.15.

Listing 12.15 managejobs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/joblistview"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:scrollbars="vertical"
>
    <TextView        android:id="@+id/statuslabel"
        android:text="list jobs here "
        android:layout_height="wrap_content"
        android:layout_width="fill_parent"
    />
    <ListView        android:id="@+id/joblist"
        android:layout_height="fill_parent"
        android:layout_width="fill_parent"
    />
</LinearLayout>
```

The code in Listing 12.16 for the ManageJobs Activity connects a JobList to the GUI as well as reacts to the selection of a particular Job from the ListView. In addition, this class demonstrates taking the result from another, synchronously-invoked Activity and processing it according to its specific requirement. For example, when a Job is completed and Closed, that JobEntry is updated to reflect its new status.

Listing 12.16 ManageJobs.java implements the ManageJobs Activity

```
package com.msi.manning.UnlockingAndroid;

// multiple imports omitted for brevity, see full source

public class ManageJobs extends Activity implements OnItemClickListener
{
    final int SHOWJOB = 1;
    Prefs myprefs = null;
    JobList _joblist = null;
    ListView jobListView;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.managejobs);
        myprefs = new Prefs(this.getApplicationContext());
        TextView tv = (TextView) findViewById(R.id.statuslabel);    1
        _joblist = JobList.parse(this.getApplicationContext());    2
        if (_joblist == null)
        {
            _joblist = new JobList(this.getApplicationContext());    3
        }
    }
}
```

```

        if (_joblist.getJobCount() == 0)                                4
        {
            tv.setText("There are No Jobs Available");
        }
        else
        {
            tv.setText("There are " + _joblist.getJobCount() + "
jobs.");
        }
        jobListView = (ListView) findViewById(R.id.joblist);          5
        ArrayAdapter<JobEntry> adapter = new
ArrayAdapter<JobEntry>(this,
        android.R.layout.simple_list_item_1, _joblist.getAllJobs());
        6
        jobListView.setAdapter(adapter);                              7
        jobListView.setOnItemClickListener(this);                      8
        jobListView.setSelection(0);
    }
    public void onItemClick(AdapterView parent, View v, int position, long
id)                                9
    {
        JobEntry je = _joblist.getJob(position);                      10
        Log.i("CH12", "job clicked! [" + je.get_jobid() + "]");
        Intent jobintent = new Intent(this, ShowJob.class);           11
        Bundle b = je.toBundle();                                     12
        jobintent.putExtras(b);                                       13
        startActivityForResult(jobintent, SHOWJOB);                  14
    }
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent
data)
    {
        switch (requestCode)
        {
            case SHOWJOB:
                if (resultCode == 1)                                    15
                {
                    Log.d("CH12", "Good Close, let's update our list");
                    JobEntry je = JobEntry.fromBundle(data.getExtras()); 16
                    _joblist.replace(je);                                17
                }
                break;
        }
    }
}

```

1. Connect TextView instance to the layout. This will allow the method to update the label with a count of the number of jobs in the list.
2. Parse the persistent xml file into a JobList. Note the Context argument is required to allow the JobList class access to the private file area for this application.
3. If the parse fails for whatever reason, initialize the JobList instance to a new, empty list. This is a somewhat simplistic way to handle the error without the GUI falling apart.

4. Selectively update the status label with the # of jobs in the list.
5. Get a reference to the `ListView`.
6. Create an instance of the `ArrayAdpater` class which combines a built-in Android layout with our `JobList`. Note the use of the `getAllJobs()` method.
7. Associate the `ArrayAdapter` with the `ListView`.
8. Tell the `ListView` to pass any click events to this class. The `ManageJobs` class implements the `OnClickListener` interface.
9. The `onItemClicked()` method is required by the `OnClickListener` interface.
10. When an item is clicked, fetch it from the `JobList` with a call to `getJob()`, passing in the index to the list. Note that the `ListView` and the `JobList` have a simple one-to-one index relationship.
11. Prepare an `Intent`, which is used to explicitly start an instance of the `ShowJob` Activity.
12. We need to pass the selected `Job` to the `ShowJob` Activity, but we cannot casually pass an `Object` from one Activity to another. We don't want the `ShowJob` Activity to have to parse the list of jobs again, otherwise we could simply pass back an "index" to the selected `Job` by using the integer storage methods of a `Bundle`. Perhaps we could store the currently selected `JobEntry` (and `JobList` for that matter) in a "global" data member of the `Application` object, should we have chosen to implement one? However, if you recall back to Chapter 1 when we discussed the ability of Android to dispatch `Intents` to any Activity registered on the device, we want to keep the ability open to an application other than our own to perhaps pass a `Job` to us. If that were the case, using a global data member of an `Application` object would never work! Never mind for the moment the likelihood of such a step being low, particularly considering how the data is stored in this application. This chapter's sample application is an exercise of evaluating some different mechanisms one might employ to solve data movement around Android. The chosen solution is to package the data fields of the `JobEntry` in a `Bundle` to move a `JobEntry` from one Activity to another. In the strictest sense, we are not really moving a `JobEntry` object, but rather a representation of a `JobEntry`'s data members. The net of this long discussion is that this method creates a new `Bundle` by using the `toBundle()` method of the `JobEntry`.
13. Add this `Bundle` to the `Intent`.
14. Start the `ShowJob` Activity synchronously. Note the use of the last parameter, `SHOWJOB`.
15. The `onActivityResult` callback method is invoked when the `ShowJob` Activity has finished. The result code is checked for success, which in this case is a value equaling 1.

16. Using the static method of the `JobEntry` class, `fromBundle()`, a new `JobEntry` instance is created.

17. The `JobList` is updated to reflect the changes to the `JobEntry`.

Now that we can view and select the Job of interest, it is time to look at just what we can do with that Job. Before diving in to the next section, be sure to review the `ManageJobs` code carefully to understand how the `JobEntry` information is passed between the two Activities.

12.4.3 *ShowJob*

The `ShowJob` Activity is arguably the most interesting element of the entire application, and is certainly the screen most useful to the mobile worker carrying around his Android capable device and toolbox. To help in the discussion of the different features available to the user on this screen, take a look at Figure 12.10.

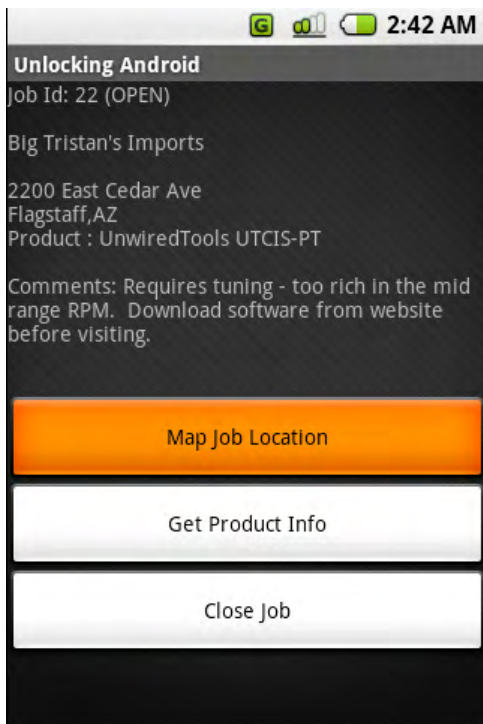


Figure 12.10. An example of a Job shown in the `ShowJob` Activity.

Again, the layout is very straightforward, however this time we have some Buttons and we will be changing the textual description depending on the condition of a particular Job's status. A `TextView` is used to present Job details such as address, product requiring service

and comments. The third Button will have the text property changed, depending on the status of the Job. If the Job's status is marked as Closed, the functionality of the third button will change.

To support the functionality of this Activity, first the code needs to launch a new Activity to show a map of the Job's address as shown in Figure 12.11.

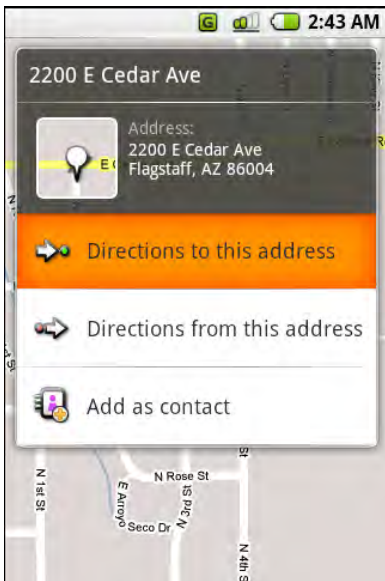


Figure 12.11. Viewing a Job address in the Maps application.

The second button, Get Product Info, launches a browser window to assist the user in learning more about the product he is being called upon to work with. Figure 12.12 shows this in action.

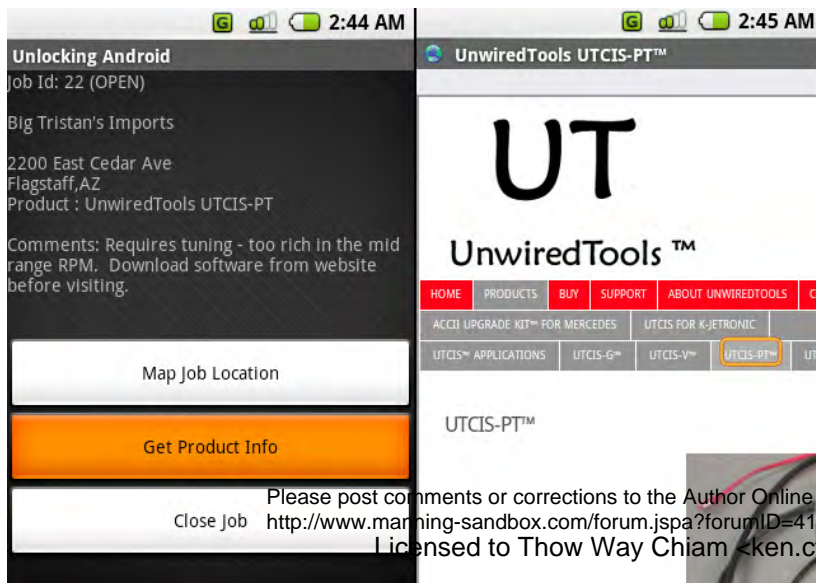


Figure 12.12. Get Product Info takes the user to a web page specific to this Job.

The third requirement is to allow the user to “close” the job or to view the signature if it is already closed, the details of which are covered in the next section on the CloseJob Activity.

Fortunately, the steps required for the first two operations are quite simple with Android – thanks to the Intent. Listing 12.18 and the accompanying descriptions show you how.

Listing 12.18 ShowJob.java

```
package com.msi.manning.UnlockingAndroid;

// multiple imports omitted for brevity, see full source

public class ShowJob extends Activity
{
    Prefs myprefs = null;
    JobEntry je = null;
    final int CLOSEJOBTASK = 1;
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        setContentView(R.layout.showjob);
        myprefs = new Prefs(this.getApplicationContext());           1
        StringBuilder sb = new StringBuilder();
        String details = null;
        Intent startingIntent = getIntent();                          2
        if (startingIntent != null)
        {
            Bundle b = startingIntent.getExtras();                    3
            if (b == null)
            {
                details = "bad bundle?";
            }
            else
            {
                je = JobEntry.fromBundle(b);
                sb.append("Job Id: " + je.get_jobid() + " (" +
je.get_status()+ ") \n\n");
                sb.append(je.get_customer() + "\n\n");
                sb.append(je.get_address() + "\n" + je.get_city() + ", "
+ je.get_state() + "\n" );
                sb.append("Product : " + je.get_product() + "\n\n");
                sb.append("Comments: " + je.get_comments() + "\n\n");
                details = sb.toString();
            }
        }
    }
}
```

```

    }
}
else
{
    details = "Job Information Not Found.";           4
    TextView tv = (TextView) findViewById(R.id.details);
    tv.setText(details);
    return;
}
TextView tv = (TextView) findViewById(R.id.details);    5
tv.setText(details);
Button bmap = (Button) findViewById(R.id.mapjob);
bmap.setOnClickListener(new Button.OnClickListener()    6
{
    public void onClick(View v)
    {
        // clean up data for use in GEO query
        String address = je.get_address() + " " + je.get_city() + " " +
je.get_zip();
        String cleanAddress = address.replace(", ", "");
        cleanAddress = cleanAddress.replace(' ', '+');
        try
        {
            Intent geoIntent = new
Intent("android.intent.action.VIEW",android.net.Uri.parse("geo:0,0?q=" +
cleanAddress));
            startActivity(geoIntent);
        }
        catch (Exception ee)
        {
        }
    }
});
Button bproductinfo = (Button) findViewById(R.id.productinfo);
bproductinfo.setOnClickListener(new Button.OnClickListener()
{
    public void onClick(View v)
    {
        try
        {
            Intent productInfoIntent = new
Intent("android.intent.action.VIEW",android.net.Uri.parse(je.get_producturl
()));
            startActivity(productInfoIntent);
        }
        catch (Exception ee)
        {
        }
    }
});
Button bclose = (Button) findViewById(R.id.closejob);
if (je.get_status().equals("CLOSED"))                9
{
    bclose.setText("Job is Closed. View Signature");
}
bclose.setOnClickListener(new Button.OnClickListener()
{

```

```

        public void onClick(View v)
        {
            if (je.get_status().equals("CLOSED"))           10
            {
                Intent signatureIntent = new
                Intent("android.intent.action.VIEW",android.net.Uri.parse(myprefs.getServer
                () + "sigs/" + je.get_jobid() + ".jpg"));

                startActivity(signatureIntent);

            }
            else
            {
                Intent closeJobIntent = new
                Intent>ShowJob.this,CloseJob.class);           11
                Bundle b = je.toBundle();
                closeJobIntent.putExtras(b);
                startActivityForResult(closeJobIntent,CLOSEJOBTASK);

            }
        }
    });
    Log.d("CH12","Job status is :" + je.get_status());
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
data)
{
    switch (requestCode)
    {
        case CLOSEJOBTASK:
            if (resultCode == 1)
            {
                this.setResult(1, "", data.getExtras());    12
                finish();
            }
            break;
    }
}
}
}

```

1. Get a reference to the Prefs as this will be needed if when we need to fetch a signature from the server.
2. Get a reference to the Intent which launched this Activity.
3. Extract the Bundle from the Intent. If present, this contains data elements which represent the JobEntry of interest.
4. In the event that the Bundle was not found, present something useful to the user and return.
5. Obtain a reference to the TextView needed to display Job details.
6. Wire up the Button's click handler. The address data is pulled from the JobEntry instance.

7. A "geo" query is built into an Intent and then subsequently launched.
8. Each JobEntry contains a URL for product information. Using this URL it is easy to take the user to a web page showing the product detail.
9. Using the current Job's status, this Activity switches both the textual description and the functionality of the third Button.
10. If the status of the Job is closed, the Button takes the user to view the captured electronic signature.
11. If the status is not yet closed, the CloseJob Activity is launched as a sub Activity.
12. Upon completion of the CloseJob Activity, the onActivityResult callback is invoked. When this situation occurs, this method receives a Bundle containing the data elements for the recently closed JobEntry. If you recall, the ShowJob Activity was launched "for result". The requirement is to propagate this JobEntry data back up to the calling Activity, ManageJobs. Calling setResult() and passing the Bundle (obtained with getExtras()) fulfills this requirement.

Despite the simple appearance of some text and a few easy-to-hit buttons, the ShowJob Activity provides a significant amount of functionality to the user. All that remains is to capture the signature to close out the Job. To do this requires an examination of the CloseJob Activity.

12.4.4 CloseJob

Our faithful mobile technician has just completed the maintenance operation on the part and is ready to head off to lunch before stopping for another job on the way home, but first – he must close out this job with a signature from the customer. To accomplish this, the Field Service Application presents a blank screen and the customer uses a stylus (or a mouse in the case of the Android Emulator) to "sign" the device, acknowledging that the work has been completed. Once the signature has been captured, the data is submitted to the Server. The proof of Job completion has been captured and the Job may now be billed. Figure 12.13 demonstrates this sequence of events.

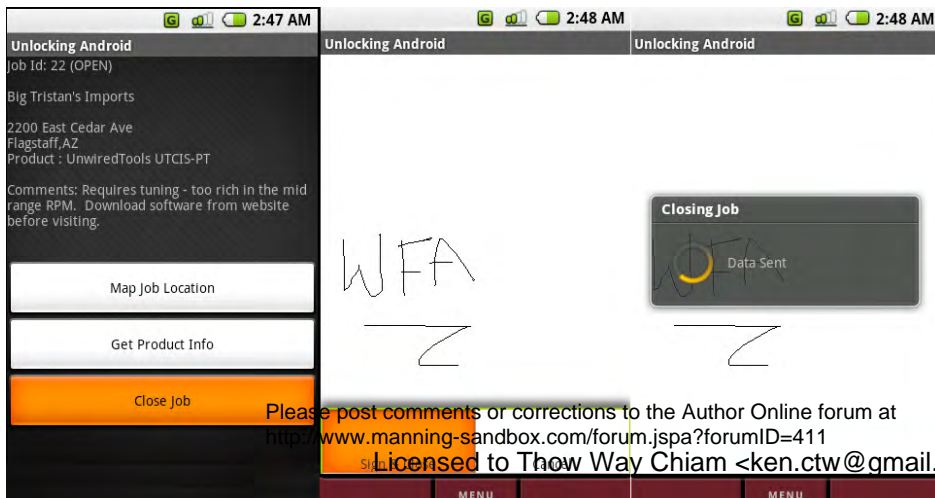


Figure 12.13. The CloseJob Activity capturing a signature and sending data to the Server.

This Activity can be broken down into two basic functions. The first is the capture of a signature. The second is transmittal of Job data to the server. Of interest is that this Activity's user interface has no layout resource. All of the user interface elements in this Activity are generated dynamically as seen in code Listing 12.19. Additionally, the ProgressDialog introduced in the RefreshJobs Activity is brought back for an encore, to let our mobile technician know that the captured signature is being sent when the "Sign & Close" menu option is selected. If the user selects "Cancel", the ShowJob Activity resumes control. Note that the signature should be made prior to selecting the menu option.

Local Queuing

One element not found in this sample application is the local queuing of the signature. Ideally this would be done in the event that data coverage is not available. The storage of the image is actually quite simple; the perhaps more challenging piece is the logic on when to attempt to send the data again. Considering all of the development of this sample application is done on the Android Emulator with near perfect connectivity, it is of little concern here. However, in the interest of best preparing you to write real-world applications it is worth the reminder of local queuing in the event of communications trouble in the field.

Listing 12.19 CloseJob.java

```
package com.msi.manning.UnlockingAndroid;

// multiple imports omitted for brevity, see full source

public class CloseJob extends Activity
{
    ProgressDialog myprogress;
    Handler progresshandler;
    Message msg;
    JobEntry je = null;
    private closejobView sc = null;
    @Override
    public void onCreate(Bundle icle)
    {
        super.onCreate(icle);
        Intent startingIntent = getIntent();
        if (startingIntent != null)
        {
            Bundle b = startingIntent.getExtras()
```

1


```

        msg.what = 1;
        progresshandler.sendMessage(msg);
    }
    return true;
case 1:
    finish();
    return true;
}
return false;
}
class DoCloseJob implements Runnable
{
    Prefs _myprefs;
    DoCloseJob(Prefs p)
    {
        _myprefs = p;
    }
    public void run()
    {
        try
        {
FileOutputStream os = getApplication().openFileOutput("sig.jpg", 0); 7
            os.Save(os);
            os.flush();
            os.close();
            // reopen to so we can send this data to server

File f = new
File(getApplication().getFileStreamPath("sig.jpg").toString());
long flength = f.length();
FileInputStream is = getApplication().openFileInput("sig.jpg");
    byte data[] = new byte[(int) flength];
        int count = is.read(data);
        if (count != (int) flength)
        {
            // bad read?
        }
        msg = new Message();
        msg.what = 0;
        msg.obj = (Object)("Connecting to Server");
        progresshandler.sendMessage(msg);
        URL url = new URL(_myprefs.getServer() +
"/closejob.php?jobid=" + je.get_jobid());
        URLConnection conn = url.openConnection();
        conn.setDoOutput(true);
        BufferedOutputStream wr = new
BufferedOutputStream(conn.getOutputStream());
        wr.write(data);
        wr.flush();
        wr.close();
        msg = new Message();
        msg.what = 0;
        msg.obj = (Object)("Data Sent");
        progresshandler.sendMessage(msg);
        BufferedReader rd = new BufferedReader(new
InputStreamReader(conn.getInputStream()));
        String line = "";
        Boolean bSuccess = false;

```

```

        while ((line = rd.readLine()) != null)
        {
            if (line.indexOf("SUCCESS") != -1)                11
            {
                bSuccess = true;
            }
        }
        wr.close();
        rd.close();
        if (bSuccess)
        {
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object)("Job Closed Successfully");
            progresshandler.sendMessage(msg);
            je.set_status("CLOSED");                            12
            CloseJob.this.setResult(1,"",je.toBundle());        13
        }
        else
        {
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object)("Failed to Close Job");
            progresshandler.sendMessage(msg);
            CloseJob.this.setResult(0);
        }
    }
    catch (Exception e)
    {
        Log.d("CH12","Failed to submit job close
signature: " + e.getMessage());
    }
    msg = new Message();
    msg.what = 1;
    progresshandler.sendMessage(msg);
}

public class closejobView extends View                14
{
    Bitmap _bitmap;                                    15
    Canvas _canvas;                                    15
    final Paint _paint;                                15
    int lastX;
    int lastY;

    public closejobView(Context c)
    {
        super(c);
        _paint = new Paint();                            16
        _paint.setColor(Color.BLACK);
        lastX = -1;
    }

    public boolean Save(OutputStream os)                17
    {
        try
        {

```



```

        _canvas.drawText("Unlocking Android", 10, 10, _paint);          18
        _canvas.drawText("http://manning.com/ablesen", 10, 25, _paint);
        _canvas.drawText("http://android12.msi-wireless.com", 10, 40,
_paint);
        _bitmap.compress(Bitmap.CompressFormat.JPEG, 100, os);          19
            invalidate();
            return true;
        }
        catch (Exception e)
        {
            return false;
        }
    }
    @Override
    protected void onSizeChanged(int w, int h, int oldw, int oldh)
    {
        Bitmap img = Bitmap.createBitmap(w, h, Bitmap.Config.ARGB_8888); 20
        Canvas canvas = new Canvas();
        canvas.setBitmap(img);
        if (_bitmap != null)
        {
            canvas.drawBitmap(img, 0, 0, null);
        }
        _bitmap = img;
        _canvas = canvas;
        _canvas.drawColor(Color.WHITE);
    }
    @Override
    protected void onDraw(Canvas canvas)
    {
        if (_bitmap != null)
        {
            canvas.drawBitmap(_bitmap, 0, 0, null);          21
        }
    }
    @Override
    public boolean onTouchEvent(MotionEvent event)              22
    {
        int action = event.getAction();
        int X = (int)event.getX();
        int Y = (int)event.getY();
        switch (action )
        {
            case MotionEvent.ACTION_UP:
                // reset location
                lastX = -1;
                break;
            case MotionEvent.ACTION_DOWN:
                if (lastX != -1)
                {
                    if ((int) event.getX() != lastX)
                    {
                        _canvas.drawLine(lastX, lastY, X, Y, _paint);
                    }
                }
                lastX = (int)event.getX();
                lastY = (int)event.getY();
            }
        }
    }

```

```

        break;
    case MotionEvent.ACTION_MOVE:
        if (lastX != -1)
        {
            _canvas.drawLine(lastX, lastY, X, Y, _paint);
        }
        lastX = (int)event.getX();
        lastY = (int)event.getY();
        break;
    }
    invalidate();
    return true;
}
}
}

```

1. Just as was done in the ShowJob Activity's onCreate method, this activity also needs to access the passed-in JobEntry, which is stored in a Bundle, accessed via the "extras" of the Intent. Remember we are not passing an actual JobEntry object, but rather a Bundle which contains data representing the data members of a JobEntry object.
2. Create an instance of a closejobView, which is defined below and referenced as #14. This is an extension of a View object and is set as the ContentView for this Activity. In the other Activities of this application, the argument to setContentView has been a layout reference from the "R" class.
3. The onCreateOptionsMenu method is an override of the base View's method, allowing a convenient way to add menus to this screen. Note that two menus are added, one for "Sign & Close" and one for "Cancel".
4. The onOptionsItemSelected method is invoked when the user selects a menu item. A ProgressDialog and accompanying Handler are instantiated.
5. A new Thread is created and started to process the steps required to "Close the Job". Note that an instance of Prefs is passed in as an argument to the constructor.
6. The constructor of the DoCloseJob class makes a reference to the Prefs instance.
7. At this point, we have a signature on the screen and need to capture it. A new FileOutputStream is obtained for a file on the local file system. A method of the closejobView named Save() is invoked, which is responsible for converting the signature to a JPG image. This step is covered in # 17. Once captured, the file contents are read into a byte array via an instance of a FileInputStream.
8. Using the Prefs instance to get specific configuration information, a URL is created to POST data to the server. The Query String of the URL contains the jobid and the POST data contains the image itself. In the next section we will demonstrate how this data is handled on the Server.
9. An BufferedOutputStream is used to send the POST data, which is the captured signature in JPEG format.

10. After sending the data, the method looks for a return value from the Server to indicate that the data was successfully transmitted.
11. Look for a particular string, namely "SUCCESS".
12. Upon success, the `JobEntry` status member is marked as "CLOSED".
13. This `JobEntry` is converted to a `Bundle` and the information communicated to the caller by invoking the `setResult()` method. Once the `Handler` receives the "I'm done" Message and the Activity finishes, this data is propagated back to the `ShowJob` and subsequently, to the `ManageJob` Activity.
14. The class `closejobView` extends the base class, `View`. This is the class which is responsible for allowing the signature to be written on the screen.
15. A few drawing classes are necessary to help capture the signature, namely the `Bitmap`, `Canvas`, and `Paint` classes.
16. In the constructor, the `Paint` object is instantiated, setting the color to `Black`, along with some other initialization of the drawing code.
17. The `Save` method of the `closejobView` is responsible for converting the contents of the image to a form usable for submission to the server.
18. Note also that additional text is drawn on the signature. In this case, it is little more than a shameless plug for this book's webpage, however this could also be Location based data. Why is this important? Imagine someone forging a signature. Could happen, but it would be more challenging and of less value to a rogue mobile technician if the GPS/Location data were actually stamped on the Job, along with the date and time.
19. The input argument to this method is an `OutputStream`, which is subsequently passed to the `compress` method of the `Bitmap` object.
20. The `Bitmap` and `Canvas` classes work together to form the drawing surface for this Activity. Note the call to the `Canvas.drawColor` method which sets the background color to `WHITE`.
21. When the `onDraw()` method is invoked, the canvas draws its associated `Bitmap` with a call to `drawBitmap()`.
22. The logic for where to draw relies on the `onTouchEvent` method, which receives an instance of the `MotionEvent` class. The `MotionEvent` class tells what happened and where. `ACTION_UP`, `ACTION_DOWN` and `ACTION_MOVE` are the events captured with some logic to guide when and where to draw.

That wraps up the source code review for the Android side of things. Now it is time to take quick look at the server application.

12.5 Server Code

A mobile application often relies on server side resources, and our Field Service Application is no exception. Since this is not a book on server side development techniques, server related code and discussion things will be presented briefly and matter-of-factly.

12.5.1 Dispatcher User Interface

Before jumping into any server code specific items, it is important to understand how the application is organized. All Jobs entered by a dispatcher are assigned to a particular mobile technician. That identifier is interpreted as an email address, as seen in the Android example where the User id was used throughout the application. Once the User Id is specified, all of the records revolve around that data element. For example, Figure 12.14 demonstrates this by showing the Jobs assigned to the author, fablesn@msiservices.com.

Unlocking Android, Chapter 12 Sample Application

For assistance with this application, please contact [Frank Ableson](#) of MSI Services, Inc.

Job List for [fablesn@msiservices.com].

Job Id#	Customer	Address	City	State	Zip	Product	Product URL	Comments	Status
18	Path of Growth, LLC	123 Main Street	Chester	NJ	07930	Wireless Router	http://cisco.com	SID broadcast not working	CLOSED
19	Indy Products	49 Route 206	Standhope	NJ	07874	Water Cooler	http://warblercool.com	Water is not cold enough!	CLOSED
21	Slm's Boats, Inc	1 Orchard Lane	Chester	NJ	07930	Cigarette Boat	http://fishcraft.com/	needs a light	CLOSED
22	Big Tristan	2200 East Cedar Ave	Flagstaff	AZ	86004	UnwiredTools	http://unwiredtools.com	Requires tuning - too rich in the mid range RPM. Download software from website before visiting.	CLOSED
23	JF's Ices	17 Route 206	Standhope	NJ	07874	Gelato Machine	http://gel.com	Ice pops	CLOSED
24	Matyas Grocer	144 Whitehall Road	Andover	NU	07821	Rotokiller	http://rotokiller.com	Required firmware upgrade.	CLOSED
27	Google	123 Main Street	Somewhere	CA	12345	Android	http://google.com	test	CLOSED

[Export Your Job List](#)

[Add a Job](#)

[Home](#)

MSI Wireless is a division of [MSI Services](#).
Check out [Unlocking Android](#)

Figure 12.14. The server-side dispatcher screen.

NOTE

This application is available for testing the sample application yourself. It is located at <http://android12.msi-wireless.com>. Simply sign on and add Jobs for your email address.

Let's now turn our attention to the underlying data structure, which contains the list of Jobs.

12.5.2 Database

As mentioned earlier in the architecture section, the database in use in this application is MySQL, with a single database table called tbl_jobs. The sql to create this table is provided in Listing 12.20.

Listing 12.20. Data Definition for tbl_jobs

```
CREATE TABLE IF NOT EXISTS `tbl_jobs` (  
  `jobid` int(11) NOT NULL auto_increment,           1  
  `status` varchar(10) NOT NULL default 'OPEN',  
  `identifier` varchar(50) NOT NULL,                 2  
  `address` varchar(50) NOT NULL,  
  `city` varchar(30) NOT NULL,  
  `state` varchar(2) NOT NULL,  
  `zip` varchar(10) NOT NULL,  
  `customer` varchar(50) NOT NULL,  
  `product` varchar(50) NOT NULL,  
  `producturl` varchar(100) NOT NULL,               3  
  `comments` varchar(100) NOT NULL,  
  UNIQUE KEY `jobid` (`jobid`)  
) ENGINE=MyISAM DEFAULT CHARSET=ascii AUTO_INCREMENT=25 ;
```

1. The field jobid is an auto increment integer field which is also the unique identifier for each row.
2. The identifier field corresponds to the “User Id/Email” of the assigned mobile technician.
3. The producturl is designed to be a specific url to assist the mobile technician in the field to quickly gain access to helpful information to assist in completing the job.

The next section provides a road map to the server code.

12.5.3 PHP Dispatcher Code

The server side dispatcher system is written in PHP and contains a number of files working together to create the application. Table 12.3 presents a brief synopsis of each source file to help you navigate the application should you choose to host a version of this application yourself.

Table 12.3. Server side source code.

Source File	Description
addjob.php	Form for entering new job information
closejob.php	Used by Android application to submit signature
db.php	Database connection info
export.php	Used to export list of jobs to a csv file
footer.php	Used to create a consistent look and feel for the footer of each page
getjoblist.php	Used by Android application to request job XML stream
header.php	Used to create a consistent look and feel for the header of each page
index.php	Home page, including search form

manage.php	Used to delete jobs on the web application
savejob.php	Used to save a new job (called from addjob.php)
showjob.php	Used to display a job details and load into a form for updating
showjobs.php	Displays all jobs for a particular user
updatejob.php	Used to save updates to a job
utils.php	Contains various routines for interacting with the database

Of all of these files, there are only two that actually concern the Android application. These are discussed in the next section.

12.5.4 PHP Mobile Integration Code

When the Android application runs the RefreshJobs Activity, the server side generates an XML stream. Without going into excessive detail on the server side code, the getjoblist.php file is explained in Listing 12.21.

Listing 12.21 getjoblist.php

```
<?
require('db.php');                                1
require('utils.php');                              2
$theid = $_GET['identifier'];                       3
print (getJobsXML($theid));                         4
?>
```

1. db.php contains the linkage to the MySQL database
2. utils.php contains routines for interacting with the database. For example, the routine getJobsXML is implemented in the utils.php file.
3. The variable \$theid is taken from the QUERY STRING from the GET request.
4. getJobsXML retrieves data from the database and formats each row into an XML representation. It wraps the list of XML-wrapped job records in the <joblist> tags along with the <?xml ...> header declaration to generate the expected XML structure used by the Android application. Remember, this is the data ultimately parsed by the SAX based JobListHandler class.

The other transaction which is important to our Android Field Service Application is the closejob.php file, examined in Listing 12.22.

Listing 12.22 closejob.php

```
<?
require('db.php');
require('utils.php');
$data = file_get_contents('php://input');          1
$jobid = $_GET['jobid'];                           2
```

```

    $f = fopen("~/pathtofiles/sigs/".$jobid.".jpg","w");      3
    fwrite($f,$data);                                         3
    fclose($f);
    print(closeJob($_GET['jobid']));                           4
?>

```

1. The POST-ed image data is read via the `file_get_contents()` function. The secret is the special identifier of `'php://input'`. This is the equivalent of a "binary read". This data is read into a variable named `$data`.
2. The `jobid` is extracted from the QUERY STRING.
3. The image file is written out to a directory which contains signatures as JPEG files, keyed by the `jobid` as part of the filename. When a job has been closed and the signature is requested by the Android application, it is this file which is requested in the Android browser.
4. The `closeJob` function (defined in `utils.php`) updates the database to mark the selected job as "CLOSED".

That wraps up the review of the source code for this chapter's sample application.

12.6 Summary

This chapter certainly was not short, but hopefully worth the read. The intent of this chapter's sample application was to tie a lot of things learned in the previous chapters together into a composite application that has some real world applicability to the kind of uses an Android device is capable of bringing to fruition. Is this sample application production ready? Of course not, but almost! That is as they say, and exercise for the reader.

Starting with a simple splash screen, this application demonstrated the use of handlers and displaying images stored in the resources section of an Android project. Moving along to the main screen, a simple user interface led to different Activities useful for launching various aspects of the realistic application.

Communications with the server downloaded XML data, while showing the user a `ProgressDialog` along the way. Once the data stream commenced, the data was parsed by the SAX XML parser, using a custom Handler to navigate the XML document.

Managing jobs in a `ListView` was demonstrated to be as easy as tapping on the desired job in the list. The next screen, the `ShowJobs` Activity, allowed even more functionality with the ability to jump to a Map showing the location of the job and even a specific product information page customized to this job. Both of those functions were as simple as preparing an Intent and a call to `startActivity()`.

Once the mobile technician has completed the Job in the field, the `CloseJob` Activity brought the touch-screen elements into play by allowing the user to capture a signature from his customer. That digital signature was then stamped with additional, contextual information, and then transmitted over the Internet to prove the job was done! Jumping back to what was learned in Chapter 12, it would be straight forward to add Location based data to further authenticate the captured signature.

The chapter wrapped up with a quick survey of the server side components to demonstrate some of the steps necessary to tie the mobile and the server side together.

The sample application is hosted on the Internet and is free to test out with your own Android application and of course the full source code is provided for the Android and server applications discussed in this chapter.

After seeing what can be accomplished when exercising a broad range of the Android SDK, the next chapter takes a decidedly different turn as we explore the underpinnings of Android a little deeper and look at building native C applications for the Android platform.

This book has presented a cross section of development topics in an effort to “unlock” the potential of the Android platform for the purpose of delivering useful, and perhaps even fun, mobile applications. Last chapter we built a more comprehensive application, building upon what was introduced in the prior chapters. As we embark on this final chapter, we are leaving behind the comforts of working strictly in the Android SDK, Java and Eclipse.

The Android SDK is quite comprehensive and capable, as this book has attempted to convey, but there may be times when your application requires something more. This chapter explores the steps required to build applications that run in the Linux foundation layer of Android. To accomplish this, we are going to use the “C” programming language. In this chapter we will use the term Android/Linux to refer to the Linux underpinnings of the Android platform. We also use the term Android/Java to refer to a Java application built using the Android SDK and Eclipse.

The steps of building an Android/Linux application are demonstrated commencing with a description of the environment and the required tool chain. After an obligatory “Hello World” caliber application, a more sophisticated application is constructed which implements a Daytime Server. Ultimately any application built for the Android/Linux needs to bring value to the user in some form. In an effort to meet this objective, it is desirable if Android/Java can interact in a meaningful manner with our Android/Linux application. To that end we will build a traditional Android application using Java in Eclipse to interact with the Android/Linux server application.

Let’s get started with an examination of the requirements of building our first “C” application for Android.

13.1 The Android/Linux Junction

Applications for Android/Linux are markedly different from the applications constructed with the Android SDK. Applications built with Eclipse, and the context sensitive Java syntax tools, made for a comfortable learning environment. In line with the spirit of Linux development, from here on out, all development takes place with command line tools and nothing more sophisticated than a text editor. While the Eclipse environment could certainly be leveraged for non-Java development, the focus of this chapter is on core C language coding for Android/Linux. The first place to start is with the cross-compiling tool chain required to build Android/Linux applications.

13.1.1 Tool Chain

Building applications for Android/Linux requires the use of a cross-compiler tool chain from CodeSourcery. The specific version required is the Sourcery G++ Lite Edition for ARM, found at http://www.codesourcery.com/gnu_toolchains/arm/portal/package2548?@template=release. Once installed, the Sourcery G++ tool-chain contributes a number of useful tools to assist in the creation of applications targeting Linux on ARM, which is the architecture of the Android platform. The CodeSourcery installation comes with a fairly comprehensive set of pdf documents describing the main components of the tool-chain including the C compiler, the assembler, the linker and many more tools. A full discussion of these versatile tools is well beyond the scope of this chapter; however three tools in particular are demonstrated in the construction of this chapter’s sample applications. We will be using these tools right away, so they are briefly introduced in this section.

The first and most important tool introduced is `gcc`. This tool is the compiler responsible for turning C source files into object files and optionally initiating the `link` process to build an executable suitable for the Android/Linux target platform. The full name of the `gcc` compiler for our cross-compilation environment is `arm-none-linux-gnueabi-gcc`. This tool is invoked from the command line of the development machine. The tool takes command line arguments of one or more source files along with zero or more of the numerous available switches.

The linker, `arm-none-linux-gnueabi-ld`, is responsible for producing an executable application for our target platform. When performing the link step, object code, along with routines from one or more library files, are combined into a re-locatable, executable binary file, compatible with the Android Emulator's Linux environment. While a simple application may be compiled and linked directly with `gcc`, the linker is used when creating applications with more than one source file and/or more complex application requirements.

If the linker is responsible for constructing applications from more than one contributing component, the object dump utility is useful for dissecting, or disassembling an application. We introduce the `objdump`, or `arm-none-linux-gnueabi-objdump`, tool presently however its usefulness becomes more apparent later in the chapter. This utility examines an executable application, i.e. a binary file, and turns the machine instructions found there into an assembly language listing file, suitable for analysis.

NOTE:

All of the examples in this chapter take place on a Windows XP workstation. It is also possible to use this tool-chain on a Linux development machine.

With this brief introduction behind us, let's build the obligatory "Hello Android" application to run in the Linux foundation of the Android Emulator.

13.1.2 Building an Application

The first thing we want to accomplish with our journey into Android/Linux development is to simply print something to the screen of the emulator to demonstrate that we are running something on the platform, outside of the Android SDK and its Java application environment. There is no better way to accomplish this feat than by writing a variant of the Hello World application. At this point, there will be little talk of Android Activities, Views or resource layouts. Most code samples in this chapter are in the C language. Listing 13.1 shows the code listing for our first "Hello Android" application.

Listing 13.1 Hello.c

```
#include <stdio.h>                                1

int main(int argc, char * argv[])                  2
{
    printf("Hello, Android!\n");                    3
    return 0;
}
```

1. Virtually all C language applications require a "#include" of a header file containing function definitions, commonly referred to as prototypes. In this case, the application includes the header file for the standard input and output routines, `stdio.h`.
2. The standard C language entry point for user code is the function named `main`. The function returns an integer return code (a value of zero is returned in this simple example) and takes two arguments. The first argument, `argc`, is an integer indicating the number of command line arguments passed in to the program when invoked. The second argument, `argv`, is an array of pointers to null-terminated strings representing each of the command line arguments. The first argument, `argv[0]`, is always the name of the program executing.
3. This application has but a single useful instruction, `printf`, which is to write to standard output (ie, the screen) a textual string. The `printf` function is declared in the header file, `stdio.h`.

To build this application, we employ the `gcc` tool:

```
arm-none-linux-gnueabi-gcc hello.c -static -o hellostatic
```

There are a few items to note about this command line instruction:

1. The compiler is invoked with the full name `arm-none-linux-gnueabi-gcc`.
2. The source file is named `hello.c`.
3. The `-static` command line switch is used to instruct `gcc` to fully link all required routines and data into the resulting binary application file. In essence the application is fully stand-alone and ready to be run on the target Android Emulator without any additional components. An application statically linked tends to be rather large because so much code and data is included into the executable file. For example, this statically linked application with basically a single line of code weighs in at 568,231 bytes. Ouch! If this static switch is omitted, the application is built without any extra routines linked in. In this case the application will be much smaller, however it will rely on finding compatible routines on the

target system in order to run. For now, we are keeping things simple and building our sample application in such a manner that all support routines are linked statically.

4. The output switch, `-o`, is used to request the name of the executable application to be `hellostatic`. If this switch is not provided, the default application name is `a.out`.

Now that the application is built, it is time to try it out on the Android Emulator. In order to do this we will rely on the `adb` tool introduced in Chapter 2.

13.1.3 Installing & Running the Application

In preparation to install and run the “Hello Android” application, let’s take a quick tour of our build and testing environment. There are four distinct environments/tools that need to be identified and clearly understood when building applications for Android/Linux. The first environment to grasp is the bigger picture architecture of the Android Emulator running essentially “on top of” Linux, as shown in Figure 13.1.



Fig 13.1

As presented in the early chapters of this book, there is a Linux Kernel running underneath the pretty graphical face of Android. There exist device drivers, process lists, and memory management, among other elements of a sophisticated operating system.

As shown in the previous section, we need an environment in which to compile our C code. This is most likely to be a command prompt window on a Windows machine, or a shell window on a Linux desktop machine, exercising the CodeSourcery tool chain. This is the second environment to be comfortable operating within.

NOTE

The CodeSourcery tool chain is not designed to run on the Android/Linux environment itself, so the development work being done here is considered to be “cross compiling”.

The next requirement is to copy our newly constructed binary executable application to the Android Emulator. This can be done with a call to the `adb` utility, or by using the DDMS view in Eclipse. Both of these tools were demonstrated in Chapter 2. Here is the syntax for copying the executable file to the Android Emulator:

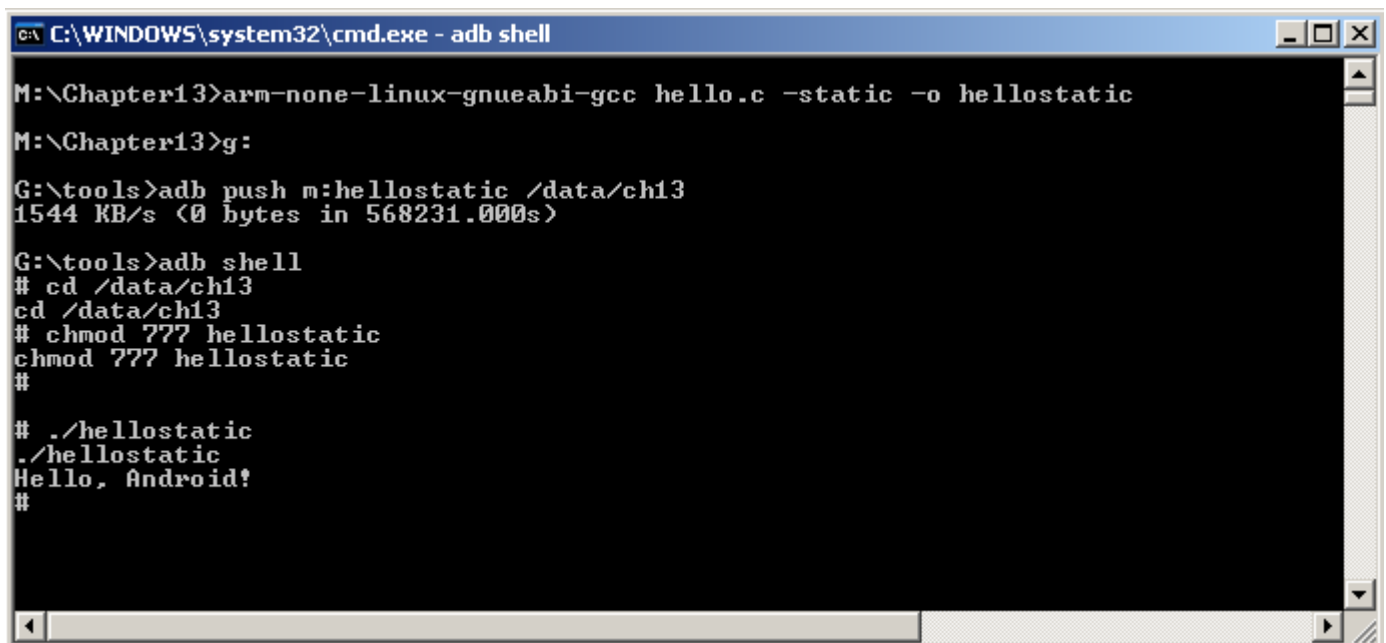
```
adb push hellostatic /data/ch13
```

Here are a few items to note about this command.

1. The command name is `adb`. This command takes a number of arguments which guide its behavior. In this case, the sub command is “push”, which means to copy a file to the Android Emulator. There is also a “pull” option for moving files from the Android Emulator file system to the local development machine’s hard drive.
2. After the `push` option, the next argument, `hellostatic` in this case, represents the local file, stored on the development machine’s hard drive.
3. The last argument is the destination directory (and/or filename) for the transferred file. In this sample, we are copying the `hellostatic` file from the current working directory to the directory named `/data/ch13` on the Android Emulator.

Be sure that the desired target directory exists first! This can be accomplished with a `mkdir` command on the `adb` shell, described next.

The final tool to become familiar with is the `shell` option of the `adb` shell. Using this command, we can interact directly on the Android Emulator’s file system with a limited shell environment. To enter this environment (and assuming the Android Emulator is already running), execute `adb shell` from the command line. When invoked, the shell displays the “#” prompt, just as if you had made a secure shell (`ssh`) or `telnet` connection to a remote unix based machine. Figure 13.2 shows these steps in action.



Looking at Figure 13.2, note the sequence shown. First the application is built with a call to gcc. Next we push the file over to the Android Emulator. We then connect to the Android emulator via the adb shell command, which gives us the # prompt, indicating that we are now on the shell. Next we change directory (cd) to /data/ch13. Remember that this is Linux, so the application by default may not be executable. A call to chmod sets the file's attributes, turning on the executable bits, allowing the application to be invoked. Lastly, we invoke the application with a call to ./helloworld. The search path for executable applications does not by default include the current directory on a Linux system, so we must provide a more properly qualified path, hence the ./ prefix. Of course, we can see that our application has run successfully because we see the "Hello, Android!" text displayed on the screen.

Congratulations, we have a successful, albeit simple, Android/Linux application running on the Android Emulator. In the next section, we take a quick look at streamlining this build process.

13.1.4 Build Script

In the last section we reviewed each of the steps in building and preparing to test our application. Due to the rather tedious nature of executing each of these steps, we have a strong desire to utilize command line tools when building C applications as it greatly speeds up the edit, compile, copy, debug cycle. This example with only a single C source file is rather simplistic; however when there are multiple source files that must be linked together, the thought of having a build script is very appealing. The need for a build script is particularly evident where there are numerous source files to compile and link, as we will encounter later in this chapter.

Listing 13.2 shows the build script for our Hello Android application.

Listing 13.2 Build script for Hello Android, buildhello.bat

```
arm-none-linux-gnueabi-gcc hello.c -static -o helloworld      1
g:\tools\adb push helloworld /data/ch13                      2
g:\tools\adb shell "chmod 777 /data/ch13/helloworld"         3
```

1. A call to arm-non-linux-gnueabi-gcc compiles the source file, hello.c. The file is statically linked against the standard C libraries and the resulting binary executable file is written out as helloworld.
2. The file helloworld is copied to the Android Emulator and placed in the directory /data/ch13.
3. The permissions for this file are changed, permitting execution. Note the use of the adb shell with a quote-delimited command. Once this command executes the adb application exits and returns to the Windows command prompt.

This example can be extended to perform other build steps or clean-up procedures such as removing temporary test data files on the Android Emulator or any similarly helpful tasks. As you progress, it will become clear what commands to put into your build script to make the testing process more efficient.

Now that the pressure is off – we have successfully written, built and executed an application in the Android/Linux environment it is time to deal with the problematic issue of a simple application requiring a file size of half a megabyte.

13.2 A Better Way

That was fun, but who wants a 500+ KB file that only displays something to the screen? Recall that the -static flag links in the essentials for running the application, including the input/output routines required for actually printing a message to the screen. If you are thinking there must be a better way, you are correct - we need to link our application to existing system libraries rather than including all of that code in our application's executable file.

13.2.1 The static Flag, Revisited

When an application is built with the -static flag, it is entirely self contained – meaning that all of the routines it requires are linked directly into the application. That is not new information, we have discussed that already. However, it has another important implication beyond just the size of the code. It also means that using Android resident code libraries is a bigger challenge. Let's dig a little deeper to understand why. In order to do this, we have to have a quick look at the file system of Android/Linux.

System libraries in Android/Linux are stored in the directory /system/lib. This directory contains some important functionality, such as OpenGL, SQLite, C standard routines, Android Runtime, user interface routines, and much more. Figure 13.3 shows a

listing of the available libraries in the Android Emulator. In short, everything that is specific to the Android platform is found in /system/lib, so if we are going to build an application that has any significant functionality, we cannot rely on the libraries that ship with CodeSourcery alone. We have to write an application which can interact with the Android system libraries. This calls for a quick side trip to discuss the functionality of the linker application.

```
# ls /system/lib
ls /system/lib
security
libdl.so
libthread_db.so
libc.so
libm.so
libstdc++.so
libz.so
libcutils.so
libexpat.so
libcrypto.so
libicudata.so
libvorbisidec.so
libsonivox.so
libdbus.so
librpc.so
libadsp.so
libaes.so
libevent.so
libctest.so
libGLES_CM.so
libssl.so
libutils.so
libicuuc.so
libdrm1.so
libreference-ril.so
libicui18n.so
libcorecg.so
libmedia.so
libpim.so
libril.so
libdrm1_jni.so
libhardware.so
libsqlite.so
libpixelflinger.so
libaudioflinger.so
libsgl.so
libnativehelper.so
libUAPI_jni.so
libagl.so
libui.so
libdvm.so
libsurfaceflinger.so
libandroid_runtime.so
libsystem_server.so
libFFIEm.so
libpv.so
libpvdownloadreg.so
libmedia_jni.so
libportsreg.so
libpnet_support.so
libpvdownload.so
libportsp.so
libwebcore.so
#
```

When building an application which requires the use of the linker, a few things change. First, the gcc command is no longer responsible for invoking the linker. Instead, the `-c` option is used informing the tool to simply compile the application and leave the link step to someone else. Here is an example:

```
arm-none-linux-gnueabi-gcc -c hello.c -o hello.o
```

This command tells the compiler to compile the file `hello.c` and place the resulting object code into the file named `hello.o`.

This process is repeated for as many source files as necessary for a particular application. For our sample application, we only have this single source file. However, in order to get an executable application, we must employ the services of the linker.

Another important change in the build environment is that we need to get a copy of the Android/Linux libraries. We are compiling on the Windows platform (or Linux if you prefer), so we need to get access to the Android Emulator's /system/lib contents in order to properly link against the library files. Just how do we go about this? We use the adb utility, of course! Listing 13.3 shows a Windows batch file used to extract the system libraries from a running instance of the Android Emulator. A few of the libraries are pointed out.

Listing 13.3 pullandroid.bat

```
adb pull /system/lib/libdl.so      m:\android\system\lib      1
adb pull /system/lib/libthread_db.so m:\android\system\lib
adb pull /system/lib/libc.so       m:\android\system\lib      2
adb pull /system/lib/libm.so       m:\android\system\lib      3
adb pull /system/lib/libGLES_CM.so m:\android\system\lib
4
adb pull /system/lib/libssl.so     m:\android\system\lib
...
adb pull /system/lib/libhardware.so m:\android\system\lib
adb pull /system/lib/libsqlite.so  m:\android\system\lib
5
many entries omitted for brevity
```

1. libdl.so is the library used for dynamic loading of libraries explicitly.
2. libc.so contains C language runtime routines, such as printf.
3. libm.so is the math library.
4. libGLES_CM is the Open GL library.
5. libsqlite.so is the SQLite database library.

Figure 13.4 shows these files now copied over to the development machine.

```
M:\android\system\lib>dir
Volume in drive M has no label.
Volume Serial Number is 48F6-A2D6

Directory of M:\android\system\lib

07/29/2008  12:22 AM    <DIR>          .
07/29/2008  12:22 AM    <DIR>          ..
07/29/2008  12:04 AM      13,836 libadsp.so
07/29/2008  12:04 AM      34,100 libaes.so
07/29/2008  12:05 AM      96,032 libagl.so
07/29/2008  12:05 AM     366,756 libandroid_runtime.so
07/29/2008  12:05 AM     100,332 libaudioflinger.so
07/29/2008  12:04 AM     241,868 libc.so
07/29/2008  12:05 AM      48,324 libcorecg.so
07/29/2008  12:04 AM     893,856 libcrypto.so
07/29/2008  12:05 AM       3,916 libctest.so
07/29/2008  12:04 AM     57,584 libcutils.so
07/29/2008  12:04 AM     321,560 libdbus.so
07/29/2008  12:04 AM       7,036 libdl.so
07/29/2008  12:05 AM      46,280 libdrm1.so
07/29/2008  12:05 AM      11,236 libdrm1_jni.so
07/29/2008  12:05 AM     451,900 libdvm.so
07/29/2008  12:05 AM      19,992 libevent.so
07/29/2008  12:04 AM     122,480 libexpat.so
07/29/2008  12:05 AM     402,404 libFFIEm.so
07/29/2008  12:05 AM      30,004 libGLES_CM.so
07/29/2008  12:05 AM      22,328 libhardware.so
07/29/2008  12:04 AM     1,041,236 libicudata.so
07/29/2008  12:05 AM     754,916 libicui18n.so
07/29/2008  12:05 AM     806,380 libicuuc.so
07/29/2008  12:04 AM     133,192 libm.so
07/29/2008  12:05 AM     84,344 libmedia.so
07/29/2008  12:05 AM      16,944 libmedia_jni.so
07/29/2008  12:05 AM     317,528 libnativehelper.so
07/29/2008  12:05 AM       6,452 libpin.so
07/29/2008  12:05 AM     114,400 libpixelflinger.so
07/29/2008  12:05 AM      4,737,012 libpv.so
07/29/2008  12:05 AM      139,296 libpvdnload.so
07/29/2008  12:05 AM       13,068 libpvdnloadreg.so
07/29/2008  12:05 AM     589,180 libpvnnet_support.so
07/29/2008  12:05 AM     705,712 libportsp.so
07/29/2008  12:05 AM       13,188 libportspreg.so
07/29/2008  12:05 AM     18,844 libreference-ril.so
07/29/2008  12:05 AM      31,100 libril.so
07/29/2008  12:04 AM      20,752 librpc.so
07/29/2008  12:05 AM     1,078,252 libsgl.so
07/29/2008  12:04 AM     280,844 libsonivox.so
07/29/2008  12:05 AM     444,324 libsqlite.so
07/29/2008  12:05 AM     158,608 libssl.so
07/29/2008  12:04 AM       4,152 libstdc++.so
07/29/2008  12:05 AM     147,716 libsurfaceflinger.so
07/29/2008  12:05 AM       5,936 libsystem_server.so
07/29/2008  12:04 AM       9,952 libthread_db.so
07/29/2008  12:05 AM      721,672 libUIP1_jni.so
07/29/2008  12:05 AM     187,504 libui.so
07/29/2008  12:05 AM      379,744 libutils.so
```

Figure 13.4 Android libraries pulled to the development machine.

Once these files are available on the development machine, we can proceed with the build step using the linker.

13.2.2 Linking

The name for the linker is `arm-none-linux-gnueabi-ld`. In most Linux environments the linker is named simply, `ld`. When using the linker, there are many command line options available for controlling the output. There are so many options that an entire book could be written covering no other topic. Our interest in this chapter is writing applications and we are taking as stream-lined an approach as possible. So while there may be other options available to get the job done, the aim here is to learn how to build an application which enables us as much flexibility as possible to employ the Android system libraries. To that end, Listing 13.4 shows the build script for building a dynamic version of Hello Android.

Listing 13.4

```
arm-none-linux-gnueabi-gcc -c hello.c -o hello.o 1

arm-none-linux-gnueabi-ld -entry=main -dynamic-linker /system/bin/linker -nostdlib -rpath /system/lib -
rpath-link /android/system/lib -L /android/system/lib -l android_runtime -l c -o helldynamic hello.o
2

g:\tools\adb push helldynamic /data/ch13 3
g:\tools\adb shell "chmod 777 /data/ch13/helldynamic" 3
```

1. This build script passes the `-c` compiler option when compiling the source file, `hello.c`. This way `gcc` does not attempt to link the application.
2. The link command, `arm-none-linux-gnueabi-ld` has a number of options. They are broken out in Table 13.1.
3. As in the previous example, `adb` is used to push the executable over to the Android Emulator. The permissions are also modified to mark the application as executable.

Table 13.1 Linker Options

Linker Option	Description
<code>-entry=main</code>	Indicate the entry point for the application. In this case, the function named <code>main</code>
<code>-dynamic-linker /system/bin/linker</code>	Tell the application where the dynamic linker application may be found at runtime. The <code>/system/bin/linker</code> path is found on the Android Emulator, not the development environment
<code>-nostdlib</code>	Tells linker to not include standard C libraries when attempting to resolve code during the link process
<code>-rpath /system/lib</code>	Tell the executable where libraries can be found at runtime. This works in a manner similar to the environment variable <code>LD_LIBRARY_PATH</code>
<code>-rpath-link /android/system/lib</code>	Tell the linker where libraries can be found when linking
<code>-L /android/system/lib</code>	Tell the linker where libraries can be found. This is the linker import directory.
<code>-l android_runtime</code>	Tell the linker that this application requires routines found in the library file <code>libandroid_runtime.so</code> .
<code>-l c</code>	Tell the linker that this application requires routines found in the library file <code>libc.so</code> .

-o hellodynamic	Request an output file name of hellodynamic
hello.o	Include hello.o as an input to the link process.

If our application required routines from the Open GL or SQLite libraries, the link command would have additional parameters of `-l GLES_CM` and `-l sqlite`, respectively. Leaving those library options off of the link command prevents the application from linking properly as certain symbols (functions, data) cannot be found.

So, did it work? The hellodynamic binary is now only 2504 bytes. That's a great improvement. Figure 13.5 shows a listing of the two "Hello Android" files for a remarkable comparison. Each program is run, first the static version and then the dynamic version.

```

C:\WINDOWS\system32\cmd.exe - adb shell

# ls -l hello*
ls -l hello*
-rwxrwxrwx root    root      2504 2008-07-29 04:49 hellodynamic
-rwxrwxrwx root    root    568231 2008-07-29 04:21 hellostatic
#
# ./hellostatic
./hellostatic
Hello, Android!
#
#
# ./hellodynamic
./hellodynamic
Hello, Android!
[1]  Killed                  ./hellodynamic
#

```

Figure 13.5. Hello Android. Static and Dynamically Linked

This looks great, except for one little problem. Note in Figure 13.5 the last line which says "Killed". There must be a problem with our dynamic version? We are almost there. Let's look closer.

13.2.3 Exit, not Return

While our application has successfully linked with the Android system libraries of `libc.so` and `libandroid_runtime.so` and can actually run, there are some missing pieces that cause the application to not properly execute. When we build an application in this manner, without letting the linker do all of its magic of knitting the entire application together, we have to do some housekeeping ourselves. Looks like there was something to that 500K application after all!

For one thing, if our application's entry point is the main function and the main function executes a return statement, just where is it return-ing to? Let's replace the `return` statement with an `exit()` call as shown in Listing 13.5.

Listing 13.5 Add an `exit()` call

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```
#include <stdio.h>

int main(int argc, char * argv[])
{
    printf("Hello, Android!\n");

    exit(0);
    //return 0;
}
```

2 1

1. Add a call to `exit()`. This should return execution back to the operating system.
2. Get rid of the call to `return()`. This must cause a stack underflow as there is nowhere within this application to return to!

This fixed the problem, no more “Killed” messages! Look at Figure 13.6 where we see that the dynamic version of Hello Android now runs just fine.

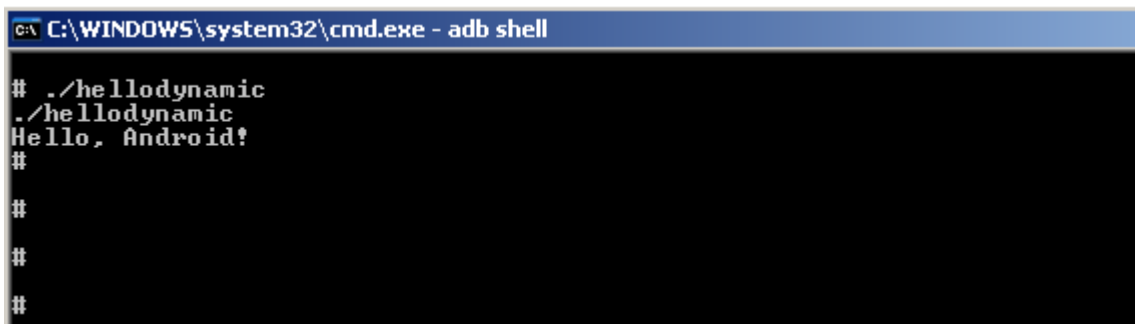


Figure 13.6 A better behaving dynamic version of Hello Android.

We’re done, right? Well, unfortunately, no. It turns out that our application does not properly interact with other libraries, nor does it properly handle the `argc` and `argv[]` arguments to the `main` function. The C library (remember, we are linking against `libc.so`) has certain expectations for application structure and stack location. We are closer, but still not quite ready for prime time.

What our application requires is a “start” routine which is called by the operating system when our application is invoked. This function in turn calls our application’s `main` function. This start routine must setup the necessary structures to allow the application to properly interact with the operating system and the core C libraries.

13.2.4 Startup Code

We have surmised that our application is missing the proper startup code, but just what does startup code for an Android/Linux application on ARM look like? Where do we turn to get this kind of information? Let’s look deeper into the bag of CodeSourcery tricks for a clue.

There are a number of executable applications that ship with Android. Let’s pull one of those over to the desktop and see what we can learn. Perhaps we can extract some information from that file that can assist in solving this puzzle?

The tool we are going to use to assist us in this effort is the object dump command, `arm-none-linux-gnueabi-objdump`. This utility has a number of options for tearing apart an ELF file for examination. An ELF file stands for Executable and Linkable Format – this is the kind of file structure used by applications in the Android/Linux environment. Using the “-d” option to the `objdump` command results in a disassembly of the executable file, showing the assembly language equivalent of the code in each executable section. Our interest is in the first `.text` section of the disassembly, as this ought to be the entry point of the application. Listing 13.6 shows a listing of the `.text` section from the `ping` program taken from the Android Emulator (via `adb pull`).

Listing 13.6 Disassembly of ping

```
000096d0 <dlopen-0x60>:
96d0: e1a0000d
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

96d4:	e3a01000	mov	r1, #0 ; 0x0	2
96d8:	e28f2004	add	r2, pc, #4 ; 0x4	3
96dc:	e28f3004	add	r3, pc, #4 ; 0x4	3
96e0:	eaffff8b	b	9514 <dlopen-0x21c>	4
96e4:	ea000e03	b	cef8 <dlclose+0x37bc>	5
96e8:	0000e408	andeq	lr, r0, r8, lsl #8	6
96ec:	0000e410	andeq	lr, r0, r0, lsl r4	6
96f0:	0000e418	andeq	lr, r0, r8, lsl r4	6
96f4:	0000e420	andeq	lr, r0, r0, lsr #8	6
96f8:	e1a00000	nop	(mov r0,r0)	7
96fc:	e1a00000	nop	(mov r0,r0)	7

1. sp represents the stack pointer. This instruction assigns the value of the stack pointer (sp) to register 0 (r0)
2. This instruction mov-es the literal value of zero to register r1
3. add 4 to the program counter (pc), leaving the result in register r2
4. The b instruction tells the code to branch to the address which is 0x21c bytes prior to the address of the dlopen function, or 9514 decimal.
5. The next branch is to an address which is 0x37bc bytes beyond the dlclose label.
6. Conditional and operations.
7. No – ops. Do nothing

OK, so that looks a little different from the rest of the code in this chapter, and just what does it do? Unfortunately, other than some basic interpretation of the op codes used, there is little to tell us why those instructions are there. After doing some research on the Internet, we find a better example of this code, shown in Listing 13.7.

Listing 13.7 crt.S

.text	1
.global _start	2
_start:	2
mov r0, sp	3
mov r1, #0	3
add r2, pc, #4	3
add r3, pc, #4	3
b __libc_init	4
b main	5
.word __preinit_array_start	6
.word __init_array_start	6
.word __fini_array_start	6
.word __ctors_start	6
.word 0	7
.word 0	7
.section .preinit_array	
__preinit_array_start:	8
.word 0xffffffff	
.word 0x00000000	
.section .init_array	
__init_array_start:	8
.word 0xffffffff	
.word 0x00000000	
.section .fini_array	
__fini_array_start:	8
.word 0xffffffff	
.word 0x00000000	
.section .ctors	
__ctors_start:	8
.word 0xffffffff	
.word 0x00000000	

1. .text directive indicates that this code should be placed in the .text section of the resulting executable
2. global _start and _start label makes this routine visible to the rest of the application and the linker.
3. mov and add instructions, just as seen in the extracted code from the ping program.

4. A branch instruction to call the `__libc_init` routine. This routine is found in the library `libc.so`. When this routine is complete, execution returns to the next instruction, another branch.
5. A branch instruction to `main`. This is the `main()` routine provided in our C application!
6. The next instructions presumably setup sections required by an executable C application.
7. A pair of `nop` instructions, i.e. do nothing
8. The sections `preinit_array`, `init_array`, `fini_array`, and `.ctors` are defined. Note that it appears that these sections are required and that the values provided are an allowable address range for these sections. The linker takes care of putting these sections into the resulting executable file. Attempting to run the application without these sections results in crashing code. I know – I tried!

NOTE

All credit for this `crt.S` file belongs to the author of a blog found at <http://honeypod.blogspot.com/2007/12/initialize-libc-for-android.html>. Additional reference material for low level Android information can be found at <http://benno.id.au>

Now that we have found an adequate startup routine, let's take a quick look at how to add this routine to our application. The assembly file is handled just like a c language file by the compiler:

```
arm-none-linux-gnueabi-gcc -c -o crt0.o crt.S
```

The resulting object file, `crt0.o` is passed to the linker as an input file, just as any other object file would be. Also, the entry switch to the linker must now specify `_start`, rather than `main`:

```
arm-none-linux-gnueabi-ld --entry=_start --dynamic-linker /system/bin/linker -nostdlib -rpath /system/lib -rpath-link \android\system\lib -L \android\system\lib -l c -l android_runtime -l sqlite -o executablefile csourcefile.o crt0.o
```

At this point, we are comfortable that we can build applications for Android/Linux, so it is time to attempt to build something useful. The next section walks through the construction of a Daytime Server.

13.3 What Time is It?

Though not talked about much today, Linux systems (and more generically, Unix systems) have a service running which provides the server's current date and time. This application, known as a Daytime Server, typically runs as a daemon, meaning in the background, not connected to a particular shell. For our purposes, we will implement a basic Daytime Server for Android/Linux, but will not worry about turning it into a background service.

This application helps exercise our interest in developing Android/Linux application. First and most importantly, this is an application of some significance beyond a simple `printf` statement. Secondly, once this application is built we write an Android/Java application to interact with the Daytime Server.

13.3.1 Daytime Server

The Daytime Server has a very basic function. The application listens on a TCP port for incoming socket connections. When a connection is made, the application writes a short textual string representation of the date and time via the socket, closes the socket, and then returns to listening for a new connection.

In addition to the TCP socket interactions, our application also logs requests to a SQLite database. Why? Because we can! The purpose of this application is to demonstrate non-trivial activities in the Android/Linux environment, including the use of the SQLite system library. Let's get started with examining the Daytime Server application.

13.3.2 *daytime.c*

The Daytime Server application can be broken into two basic functional parts. The first is the TCP socket server. Our Daytime application binds to TCP port number 1024 when looking for new connections. Ordinarily a Daytime service binds to TCP port number 13, however Linux has a security feature where only trusted users can bind to any port below 1023. The second feature is the insertion of data into a SQLite database. Listing 13.8 shows the code for the Daytime Server application.

Listing 13.8 daytime.c

```

/*
    daytime.c
*/
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <resolv.h>
#include "sqlite3.h"

int PORTNUMBER = 1024;

// define this as a macro as we cannot seem to link it successfully for Android
#define htons(a) ( ((a & 0x00ff) << 8) | ((a & 0xff00) >> 8))

void RecordHit(char * when)
{
    int rc;
    sqlite3 *db;
    char *zErrMsg = 0;
    char sql[200];

    rc = sqlite3_open("daytime_db.db",&db);
    if( rc )
    {
        printf( "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return;
    }

    bzero(sql,sizeof(sql));
    sprintf(sql,"insert into hits values (DATETIME('NOW'),'s');",when);
    rc = sqlite3_exec(db, sql, NULL, 0, &zErrMsg);
    if( rc!=SQLITE_OK )
    {
        printf( "SQL error: %s\n", zErrMsg);
    }

    sqlite3_close(db);
}

int main(int argc, char **argv)
{
    int listenfd, connfd;
    struct sockaddr_in servaddr;
    char buf[100];
    time_t ticks;
    int done = 0;
    int rc;
    fd_set readset;
    int result;
    struct timeval tv;

    printf("Daytime Server\n");

    listenfd = socket(AF_INET,SOCK_STREAM,0);
    bzero(&servaddr,sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = INADDR_ANY;
    servaddr.sin_port = htons(PORTNUMBER);

    rc = bind(listenfd, (struct sockaddr *) &servaddr,sizeof(servaddr));
    if (rc != 0)
    {
        printf("after bind,rc = [%d]\n",rc);
        return rc;
    }

    listen(listenfd,5);
    while (!done)
    {
        printf("Waiting for connection\n");
        while (1)
        {
            bzero(&tv,sizeof(tv));

```

```

        tv.tv_sec = 2;
        FD_ZERO(&readset);
        FD_SET(listenfd, &readset);
        result = select(listenfd + 1, &readset, &readset, NULL, &tv);
        if (result >= 1)
        {
            printf("Incoming connection!\n");
            break;
        }
        else if (result == 0)
        {
            printf("Timeout.\n");
            continue;
        }
        else
        {
            printf("Error, leave.\n");
            return result;
        }
    }

    printf("Calling accept:\n");
    connfd = accept(listenfd, (struct sockaddr *) NULL, NULL);
    printf("Connecting\n");
    ticks = time(NULL);
    sprintf(buf, "%.24s", ctime(&ticks));
    printf("sending [%s]\n", buf);
    write(connfd, buf, strlen(buf));
    close(connfd);
    RecordHit(buf);
}
return 0;
}

```

1. `<time.h>` contains references for the function to get the current system time, as required by this application.
2. Multiple system header files are required for interacting with tcp sockets.
3. The `sqlite3.h` header file is not provided in the CodeSourcery tool-chain. This file was acquired from a `sqlite3` distribution and the file copied into the local directory along with `daytime.c`. This is why the include file is delimited with quotation marks rather than `<>`, which is used for finding include files in the system or compiler path.
4. Define the tcp port we will listen on. We are using port # 1024 because our application cannot bind to any port 1023 or below. Only system processes may bind to ports below 1024.
5. The `htons` function is typically implemented in the library named `socket` (`libsocket.so`). Android does not provide this library, nor could I find the function in any of the system libraries. Therefore `htons` is defined here as a macro. This macro is required to get the network byte ordering correct. When the application is running, this port can be verified by running `netstat -tcp` on the command line in adb shell.
6. The `RecordHit()` function is responsible for inserting a record into the SQLite database created for this application.
7. A "handle" to the database of type `sqlite3 *` is defined.
8. The function `sqlite3_open()` opens the database.
9. A record is inserted into the database with a call to `sqlite3_exec()`, passing a handle to the database and a formatted SQL statement.
10. The database is closed.
11. A `socket` is created. This `socket` is used for listening for incoming connections.
12. The `socket` address is setup to bind to any available IP addresses and a specific port number. Note the use of the `htons` macro.
13. The `bind` function associates a `socket` with an address.
14. The `listen` function tells a `socket` to look for new connections on the address to which it is bound.
15. It is not good form for an application to block while waiting for an incoming connection. This code initializes a `timeval` structure which is used in conjunction with the `select` function to provide a well-behaving thread of execution.
16. We setup the `select` function's parameters to trigger when the listening `socket` is ready to be read. A read signal on a `socket` that is listening means that a connection attempt has been detected and that an `accept` function invocation will succeed immediately.

17. The `select` function returns within an interval that is defined as the sooner of two events: one of the sockets of interest is signaled (read, write, error) or a timeout period elapses.
18. If an incoming connection is detected, we continue on to process it with a call to `accept`.
19. If no connection has occurred within the timeout interval, display a timeout message and then return to listening. By implementing a timeout in this manner, the application can look for other work to do in between looking for new connections. This technique also allows the user interface to be responsive, when desired. Using this approach to program execution is an alternative to multi-threading.
20. The `accept()` function associates a new socket with the incoming connection request.
21. The `time()` function gets the current system time, stored in a data type of `time_t`, which is essentially a long integer.
22. The `ctime()` function converts a `time_t` value to a textual representation of the date and time and stores it in a char buffer as a null terminated string.
23. The formatted date and time is written over the socket connection using the `write()` function and the socket identifier returned by `accept()`.
24. The connection is closed with a call to `close()`.
25. The `RecordHit()` function is invoked, which initiates the storage of this hit into a SQLite database.

That is all the code necessary to implement the Android/Linux Daytime Server application. Let's look next at the sqlite3 database interaction in more detail.

13.3.3 The SQLite Database

This application employs a simple database structure created with the sqlite3 application. We interact with sqlite3 from the adb shell environment as shown in Figure 13.7. The purpose of this database is to record some data each Daytime Server processes an incoming request. From a data perspective this sample is perhaps a bit boring as it simply records the system time plus the text returned to the client, which is a `ctime` formatted time string. Though somewhat redundant from a data perspective, the purpose is to demonstrate the use of SQLite from our C application, utilizing the Android/Linux resident sqlite3 library, `libsqlite.so`.

```

C:\WINDOWS\system32\cmd.exe - adb shell
#
#
# sqlite3 daytime_db.db
sqlite3 daytime_db.db
SQLite version 3.5.0
Enter ".help" for instructions
sqlite> .databases
.databases
seq  name                file
-----
0     main                  /data/ch13/daytime_db.db
sqlite> .tables
.tables
hits
sqlite> .schema hits
.schema hits
CREATE TABLE hits (hittime date, hittext text);
sqlite> .header on
.header on
sqlite> .mode column
.mode column
sqlite> select * from hits;
select * from hits;
hittime                hittext
-----
2008-07-29 07:31:35    Tue Jul 29 07:31:35 2008
2008-07-29 07:56:27    Tue Jul 29 07:56:27 2008
2008-07-29 07:56:28    Tue Jul 29 07:56:28 2008
2008-07-29 07:56:29    Tue Jul 29 07:56:28 2008
2008-07-29 07:56:29    Tue Jul 29 07:56:29 2008
2008-07-29 07:56:29    Tue Jul 29 07:56:29 2008
2008-07-29 07:56:29    Tue Jul 29 07:56:29 2008
2008-07-29 07:56:29    Tue Jul 29 07:56:29 2008
2008-07-29 07:56:29    Tue Jul 29 07:56:29 2008
2008-07-29 07:56:30    Tue Jul 29 07:56:30 2008
sqlite> .exit
.exit
#

```

The previous section outlined the code syntax for inserting a row to the database, this section shows how to interact with the database using the sqlite3 tool. The sequence shown in Figure 13.7 is broken out and explained in Listing 13.9.

Listing 13.9. Interacting with a sqlite database.

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=411>
 Licensed to Thow Way Chiam <ken.ctw@gmail.com>

```

# pwd
pwd
/data/ch13
# sqlite3 daytime_db.db
sqlite3 daytime_db.db
SQLite version 3.5.0
Enter ".help" for instructions
sqlite> .databases
.databases
seq  name          file
-----
0    main           /data/ch13/daytime_db.db
sqlite> .tables
.tables
hits
sqlite> .schema hits
.schema hits
CREATE TABLE hits (hittime date,hittext text);
sqlite> .header on
.header on
sqlite> .mode column
.mode column
sqlite> select * from hits;
select * from hits;
hittime          hittext
-----
2008-07-29 07:31:35 Tue Jul 29 07:31:35 2008
2008-07-29 07:56:27 Tue Jul 29 07:56:27 2008
2008-07-29 07:56:28 Tue Jul 29 07:56:28 2008
2008-07-29 07:56:29 Tue Jul 29 07:56:28 2008
2008-07-29 07:56:30 Tue Jul 29 07:56:30 2008
sqlite> .exit
.exit
#

```

1. Display the current working directory with `pwd`.
2. The `sqlite3` program, found in `/system/bin`, allows a command line interaction with our sample's database file, `daytime_db.db`.
3. `.databases` displays the currently available database(s).
4. The active database is called `main` and is found in the file `/data/ch13/daytime_db.db`. This database was originally created with a command line sequence of `sqlite3 daytime_db.db`, invoked from the `/data/ch13` directory.
5. `.tables` displays the tables in the current database
6. This database has a single table, namely `hits`.
7. The `.schema` command displays the structure of a database. In this case the display is in the form of a create table syntax.
8. The structure for the `hits` table shows that it consists of two columns. The first column, named `hittime` is a date data type. The second column, named `hittext` is of data type `text`.
9. The `.header` command allows us to turn the header text on or off for subsequent query results.
10. The `.mode` command allows us to specify the format of the query results. Available options are: `column`, `csv`, `html`, `insert`, `line`, `list`, `tabs`, `tcl`.
11. Query the database with a standard sql statement.
12. The contents of the table are displayed in column format.
13. The `.exit` command leaves the `sqlite3` application and returns to the shell.

The SQLite database engine is known for its simplicity. This section displayed a simple interaction and just how easy it is to use and employ. Additionally, the `sqlite3` database may be pulled from the Android Emulator and used on the development machine as

```

C:\WINDOWS\system32\cmd.exe

C:\software\sqlite-3.5.4>g:\tools\adb pull /data/ch13/daytime_db.db \temp\daytime_db.db
128 KB/s (0 bytes in 2048.000s)

C:\software\sqlite-3.5.4>sqlite3 \temp\daytime_db.db
SQLite version 3.5.4
Enter ".help" for instructions
sqlite> .databases
seq  name          file
-----
0    main           C:\temp\daytime_db.db
sqlite> .tables
.tables
hits
sqlite> select * from hits;

```


Figure 13.8. The SQLite database on the development machine

This feature makes Android a very compelling platform for mobile data collection applications because “synching” data can be as simple as copying a database file that is compatible across multiple platforms.

13.3.4 Building and Running Daytime Server

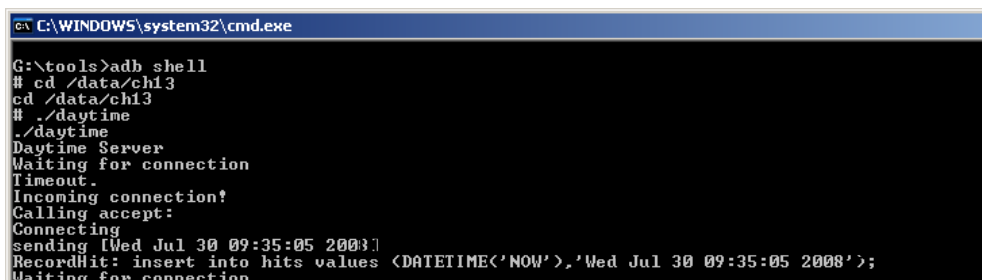
To build this application we need to combine the learning of the prior few sections. We know that our application requires a startup component and must also link against multiple libraries. Because the application interacts with the SQLite database, we must link against the sqlite library in addition to the c and android_runtime libraries. The full build script is shown in Listing 13.10.

Listing 13.10 daytime application build script

```
arm-none-linux-gnueabi-gcc -c daytime.c 1
arm-none-linux-gnueabi-gcc -c -o crt0.o crt.S 2
arm-none-linux-gnueabi-ld --entry=_start --dynamic-linker /system/bin/linker -nostdlib -rpath /system/lib
-rpath-link \android\system\lib -L \android\system\lib -l c -l android_runtime -l sqlite -o daytime
daytime.o crt0.o 3
C:\software\google\android-sdk_m5-rc15_windows\tools\adb push daytime /data/ch13 4
C:\software\google\android-sdk_m5-rc15_windows\tools\adb shell "chmod 777 /data/ch13/daytime" 5
```

1. Compile the daytime.c file, our main application code.
2. Compile the startup sequence found in crt.S. This code is assembly language.
3. Link the application, using the local copy of the Android /system/lib directory's libraries. Note the `-l sqlite` option which instruct the linker to link against the SQLite library. Note also the inclusion of both `daytime.o` and `crt0.o` object files as inputs to the linker. Both are required to properly construct the Daytime Server application.
4. Push the file to the Android Emulator
5. Change the permissions via this script so we don't have to do it on the Emulator, for convenience.

Running the Daytime Server application is the easy and fun part of this exercise. Figure 13.9 shows our Daytime Server running.



```
C:\WINDOWS\system32\cmd.exe
G:\tools>adb shell
# cd /data/ch13
cd /data/ch13
# ./daytime
./daytime
Daytime Server
Waiting for connection
Timeout.
Incoming connection!
Calling accept:
Connecting
sending [Wed Jul 30 09:35:05 2008]
RecordHit: insert into hits values (DATETIME('NOW'),'Wed Jul 30 09:35:05 2008');
Waiting for connection
```

Figure 13.9 Daytime Server running in the shell

Here is a quick run-down of the sequence shown in Figure 13.9:

- Start the shell by running `adb shell`.
- Change directories to `/data/ch13`, where our application resides, previously pushed there with an `adb push` command.
- Run `./daytime` application.
- The application binds to a port and begins listening for an incoming connection.
- A timeout occurs prior to a connection being made. Application displays timeout and then returns to look for connections again.
- A connection is detected and subsequently accepted.
- The time string is constructed and sent to the client.
- A record is inserted into the database with the shown sql statement.
- We kill the application and restart the shell. Note that this is because we did not build a clean way of killing the Daytime Server. A proper version of the application would be to convert it to a daemon, which is beyond the scope of our interest here.
- Run `sqlite3` to examine the contents of our application's database.
- Perform a select against the hits table where we see the recently inserted record.
-

We have built an Android/Linux application which implements a variant of the traditional Daytime Server application as well as interacts with a sql database. Not too shabby when you consider that this is a telephone platform! Let's move on to examine the Android/Java application used to exercise the Daytime Server, our Daytime Client.

13.4 *Daytime Client*

One of the stated objectives for this chapter is to connect the Java user interface to our Daytime Server application. This section demonstrates the construction of a Daytime Client application which communicates with our Daytime Server via TCP sockets.

13.4.1 *Activity*

The Daytime Client application has a single `Activity` which presents a single `Button` and a `TextView`, as shown in Figure 13.10.

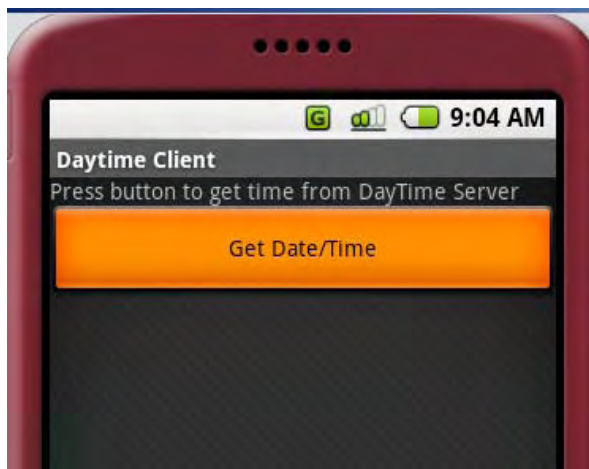


Figure 13.10 The Daytime Client

When the Button is clicked, the Activity initiates the Daytime Server query and then replaces the text of the TextView with the information received from the Daytime Server. Not much to it really, but that is fine, as all we are after in this sample is to demonstrate connectivity between the two applications. Listing 13.11 shows the onCreate method for this Activity.

Listing 13.11 User Interface elements of DaytimeClient.java

```

Handler h;
1
@Override
public void onCreate(Bundle icle)
{
    super.onCreate(icle);
    setContentView(R.layout.main);
2

    final TextView statuslabel = (TextView) findViewById(R.id.statuslabel);
3

    h = new Handler()
    {
        @Override
        public void handleMessage(Message msg)
        {
            switch (msg.what)
            {
                case 0:
5
                    Log.d("CH13","data [" + (String) msg.obj + "]");
                    statuslabel.setText((String) msg.obj);
                    break;
            }
            super.handleMessage(msg);
        }
    };

    Button test = (Button) findViewById(R.id.testit);
6
    test.setOnClickListener(new Button.OnClickListener()
    {
        public void onClick(View v)
        {
            try
            {
                Requester r = new Requester();
8
                r.start();
            }
            catch (Exception e)
            {
                Log.d("CH13 exception caught : ",e.getMessage());
            }
        }
    });

```

```

    }
    });
}

```

1. A Handler object is declared. This will be used for handling updates to the TextView as the communications elements of this Activity take place in a separate Thread, as shown in the next section.
2. Setup the user interface.
3. Obtain a reference to the TextView.
4. Implement the Handler as an anonymous class with the required handleMessage method.
5. The Handler has a single function, which is to update the user interface with textual data stored in the obj member of a Message object.
6. Obtain a reference to the Button.
7. Define the OnClickListener as an anonymous class.
8. When the Button is clicked, create a new instance of the Requester class, which is explained in the next section in the discussion of building a Socket Client.

While the user interface of this application is very simple, the more interesting side of this Activity is the interaction with the Daytime Server, which takes place in the Requester class, shown in the next section.

13.4.2 Socket Client

The Daytime Server application listens on a TCP port for incoming connections. In order to request the Date and Time, the Daytime Client must establish a client socket connection to the Daytime Server. It is hard to imagine a more simple TCP service than this –open a socket to the server, read data until the socket connection is closed. There is no additional requirement. Most of the networking examples in this book have focused on a higher level protocol, HTTP, where the request and response is clearly defined with headers and a specific protocol to observe. In this example, the communications involve a lower level socket connection, essentially raw if you will because there is no protocol associated with it beyond being a TCP Stream (as opposed to UDP). Listing 13.12 demonstrates this lower level socket communication.

Listing 13.12 Requester class implementation

```

public class Requester extends Thread           1
{
    Socket requestSocket;                        2
    String message;
    StringBuilder returnStringBuffer = new StringBuilder(); 3
    Message lmsg;                               4
    int ch;                                    5

    public void run()                           6
    {
        try
        {
            requestSocket = new Socket("localhost", 1024); 7
            InputStreamReader isr = new InputStreamReader(requestSocket.getInputStream(), "ISO-8859-
1"); 8

            while ((ch = isr.read()) != -1)
            {
                returnStringBuffer.append((char) ch); 9
            }
            message = returnStringBuffer.toString(); 10
            lmsg = new Message();                 11
            lmsg.obj = (Object) message;          11
            lmsg.what = 0;                        11
            h.sendMessage(lmsg);                  11
            requestSocket.close();                 12
        }
        catch (Exception ee)
        {
            Log.d("CH13", "failed to read data" + ee.getMessage()); 13
        }
    }
}

```

1. Our Requester class extends the Thread class.
2. Communications takes place via an instance of the Socket class, which is found in the java.net package.

3. A `StringBuilder` is used to accumulate data from the `Socket`.
4. A `Message` is used to communicate back to the user interface thread.
5. Data is read in from the `Socket` one character at a time.
6. Implementation of the `run` method for actually performing the work of this class.
7. Create a new `Socket` instance. Note the two parameters, one is the hostname and the other is the tcp port number. The port number is 1024, matching the port used by our Daytime Server.
8. Once the `Socket` is created, we get an `InputStreamReader`, using the `InputStream` from the socket and a preferred encoding description. This encoding tells the `InputStreamReader` how to interpret the data being read across the network. In this example, this really is not a concern because the data is plain text, however if this data contained accented characters or other special data, the encoding assists in building the proper representation of the received information.
9. This method reads on the socket (via the `InputStreamReader`) until the `Socket` is closed.
10. Convert the `StringBuilder` to a `String`.
11. Initialize a new `Message` object with a "what" value of 0 and assigning the `String` to the `obj` member of the `Message` object.
12. Send the `Message` to our `Handler`.
13. Close the `Socket`. This is ultimately more of a formality, as the connection will have already been closed by the Daytime Server.

With the Daytime Client now coded, it is time to test the application. In order for the Daytime Client to access a TCP socket, a special permission entry is required in the `AndroidManifest.xml` file: `<uses-permission android:name="android.permission.INTERNET"></uses-permission>`.

13.4.3 Testing the Daytime Client

The first step in testing the Daytime Client is to ensure that the Daytime Server application is running, as described in section 13.3.4. Once you know the Daytime Server is running, run the Daytime Client.

NOTE

If you are unclear on how to build and run the Daytime Client, refer to Chapter Two for properly setting up the Android development environment in Eclipse.

Figure 13.11 demonstrates the Daytime Client running, along side a view of the Daytime Server. Note how the `TextView` of the Android application is updated to reflect the Date and Time sent by the Daytime Server.

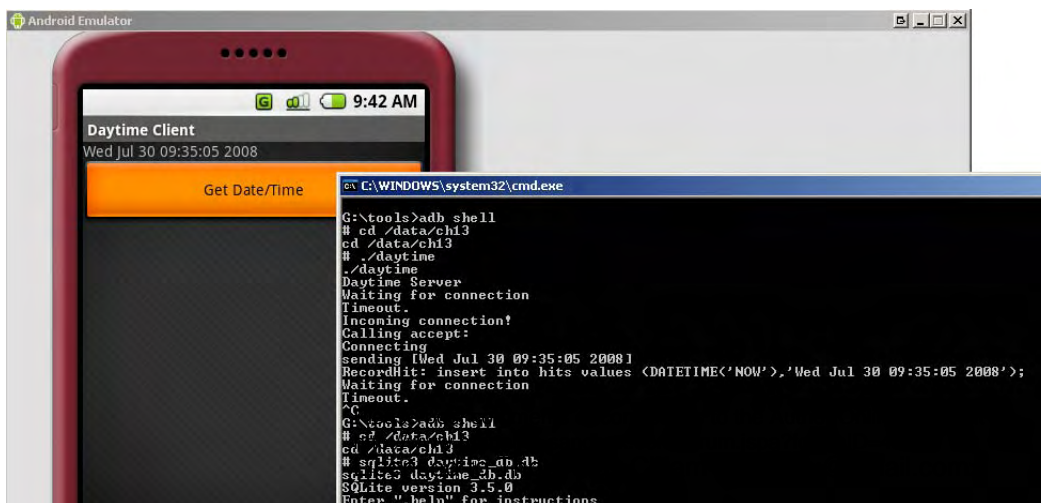


Figure 13.11 Testing the Daytime Client

The Daytime Server is exercising both tcp socket functionality and SQLite database record insertions, all running in the Android Emulator. A production ready Android/Linux application would need to be converted to run as a daemon, which is beyond our aim in this chapter.

13.5 Summary

This chapter wraps up this book with a topic that hopefully stretches your imagination for the kinds of applications possible with the versatile and open platform of Android. We had the goal of writing an application outside of the Android SDK and demonstrating how that kind of application may be leveraged by a “standard” Android Java application. To write for the Android/Linux layer, we turned to the C programming language.

Developing C language applications for Android/Linux is a cross-platform compilation exercise using the freely available Code Sourcery tool-chain. This chapter demonstrated using that tool set in conjunction with the adb utility provided in the Android SDK. The adb utility was vital as it enabled us to push our application to the Android Emulator for testing as well as it enabled us to extract the Android system libraries which were essential for linking our application with the Android resident libraries. Of course, the adb shell was used to interact directly with the Android Emulator to run our C application.

C language mastery on this platform is powerful because much of the C language development process involves porting existing, open source Linux code to the ARM processor. This has the potential benefit of speeding up development for future functionality delivery to Android by leveraging existing code bases. A logical extension to this topic would be the development of a Java Native Interface (JNI) to bring many capabilities residing in C language libraries directly into the Java environment of Android.

Our sample application exercised TCP socket communications. The TCP communications capability proved to be a ready interface mechanism between the Android/Java layer and the Android/Linux foundation of the environment in the Daytime Client and Server applications respectively. TCP socket communications may also take place from the Android/Linux environment to external, remote systems such as email servers or directory servers, opening up literally a world of possibilities.

The Daytime Server sample application also demonstrated the use of an Android resident library to manipulate a SQLite database used to store transaction data. The impact of this step should not be minimized as it satisfies three important development challenges. The first and most basic accomplishment of this functionality is that we have demonstrated linking against, and employing, an Android resident system library. This is significant because it shows how future applications may leverage Android functionality such as Open GL or media services. Secondly, using a device resident database which is also accessible from the Java layer means we have an additional (and persistent) interface mechanism between the Java and Linux environments on the platform. Lastly, Android is a mobile platform. Any time there is a mobile application, the topic of sharing and synching data bubbles up to view. We demonstrated in this chapter the ease with which an SQL capable database was shared between the Android Emulator and a personal computer – and all without complex synchronization programming. Synchronization is of course a larger topic than this, but the capability of moving a single file between platforms is a welcome feature. There are only a few comparable solutions in the marketplace for other mobile environments – and that is after literally years of market penetration by these other platforms. Android gets it right from the start.

I trust that this chapter and this book challenge you to dig deeper yourself and that you may enjoy Unlocking Android.

Appendix

Installing the Android SDK

In this appendix

- Development environment requirements
- Obtaining the latest Android SDK
- Configuring the Android Development Tools for Eclipse

This appendix walks through the installation of Eclipse, the Android SDK and Android Development Tools plug-in for Eclipse. This appendix is meant to be a reference resource to assist in setting up the development environment for Android application development. The topic of using the development tools is covered in Chapter 2 of this book.

Development Environment Requirements

In order to develop Android applications, your computing environment must satisfy the minimum requirements. Android development is a quick-paced topic, with changes coming about very rapidly, so it is a good idea to stay in tune with the latest developments from the Android development team at Google. The latest information regarding supported platforms and requirements for Android development tools may be found at <http://code.google.com/android/intro/installing.html#developmentrequirements>.

The development environment used for the sample applications in this book include:

- Windows XP/Vista or Mac OS X 10.4
- Eclipse 3.3 (Europa), including the Java Developer Tools (JDT), Web Developer Tools (WDT) which are included in the Eclipse installation package.
- JDK & JRE version 5
- Android Development Tools plug-in for Eclipse

Obtaining and Installing Eclipse

A requirement for running the Eclipse IDE is the Java JRE version 5 or later. For assistance in determining the best JRE for your development computer, you can look at this helpful page on the eclipse website: <http://www.eclipse.org/downloads/moreinfo/jre.php>. It is very likely that you already have an acceptable JRE installed on your computer. An easy way to determine what version (if any) you have on your computer is to run the following command from a command window or terminal session on your development computer:

```
java -version
```

This procedure checks to see if the JRE is installed and present in the search path on your computer. If the command comes back with an error stating an invalid or unrecognized command, it is likely that the JRE is not installed on your computer and/or that it is not properly configured. Figure A.x demonstrates using this command to check the version of the installed JRE.

```
C:\>java -version
java version "1.5.0_09"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_09-b03)
Java HotSpot(TM) Client VM (build 1.5.0_09-b03, mixed mode, sharing)

C:\>_
```

Figure A.1: The `java -version` command displays the version of Java installed on your computer.

Once your JRE is installed, you can proceed to install the Eclipse IDE. Download the latest stable release of the Eclipse IDE from <http://www.eclipse.org/downloads>. You will want to download the version for "Java Developers". This distribution is described at the Eclipse website: <http://www.eclipse.org/downloads/moreinfo/java.php>. The Eclipse download is a compressed file/folder. Once downloaded, extract the contents of the folder to a convenient place on your computer. Because this download is simply a compressed folder and not an "installer", it does not create any icons or shortcuts on your computer.

To start Eclipse, run `eclipse.exe` found in the eclipse installation directory. You may want to make your own menu or desktop shortcut to `eclipse.exe` for convenience. This will start the IDE. As seen in Figure A.X, Eclipse prompts for a "workspace" and suggests a default location such as `c:\documents and settings\username\workspace`. You may want to change that value to something Android specific to separate your Android work from other projects.

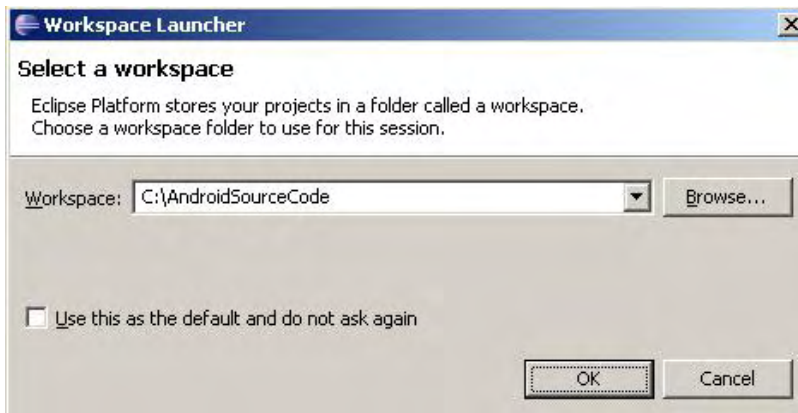
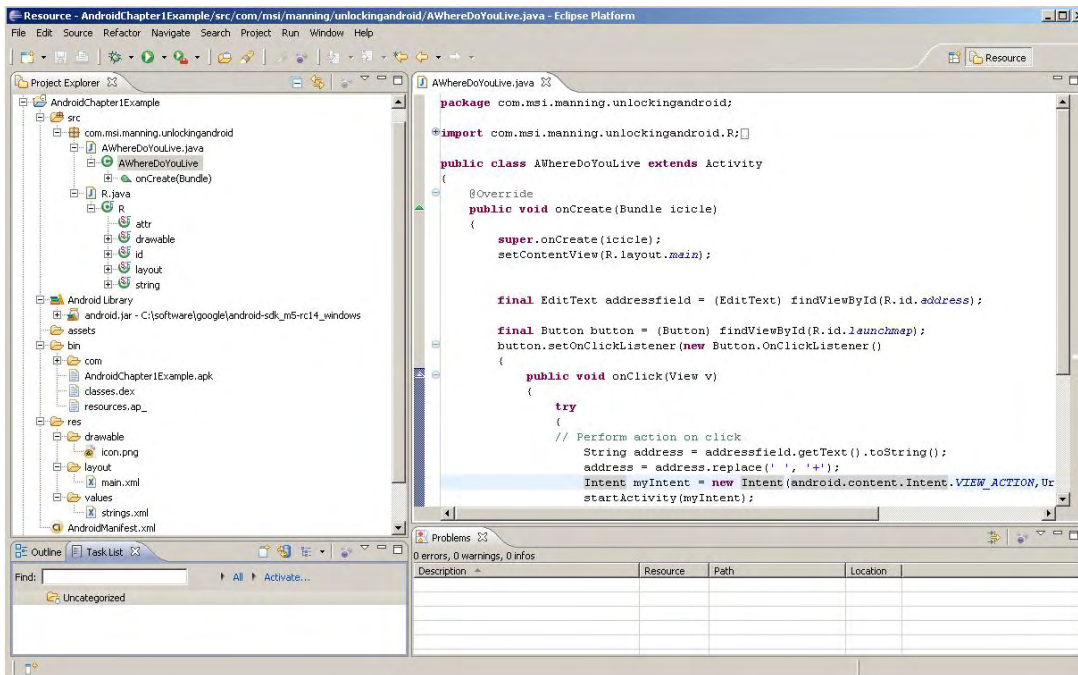


Figure A.2: Eclipse projects are stored in a workspace, which is a directory on your computer's hard drive.

Accept the suggested workspace location or specify an alternative workspace location, as desired. Once Eclipse is loaded, click on the "Workbench - Go to the workbench" icon on the main screen as seen in Figure A.X



Eclipse consists of many “perspectives”, the default being the Java perspective. It is from this perspective that Android Application development takes place. The Java perspective is shown in Figure A.x. Chapter 2 discusses in greater detail the use of the Eclipse IDE for Android application development.



For more information on getting familiar with the Eclipse environment, visit the eclipse.org website where you can find online tutorials for building Java applications with Eclipse.

Now that Eclipse is installed, it is time to focus on the Android SDK.

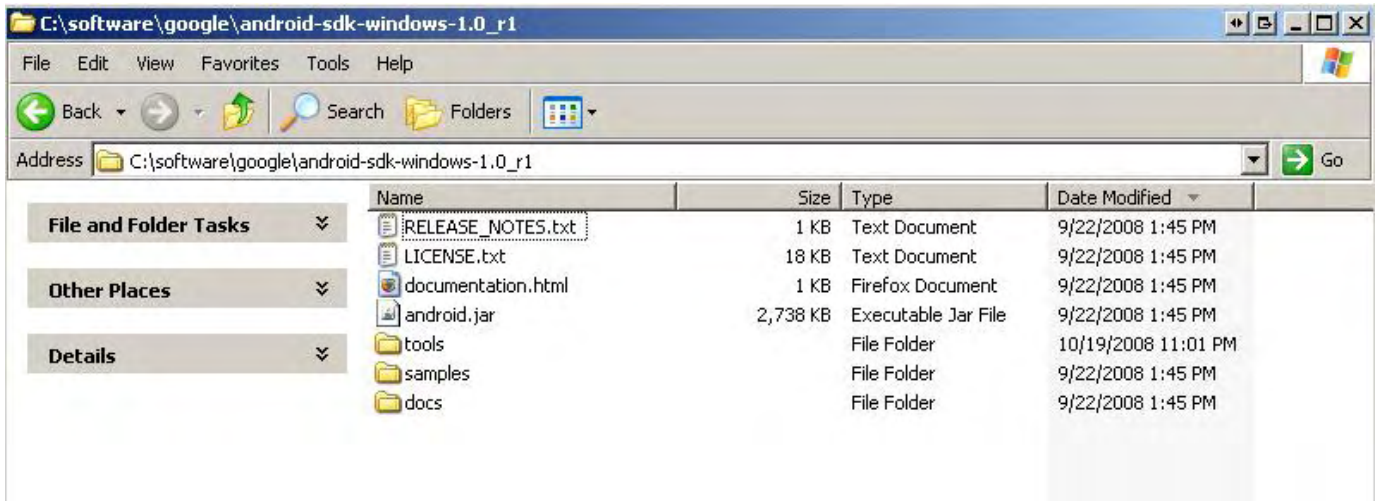
Obtaining and installing the Android SDK

The Android SDK is available as a free download from a link off of the Android home page, at <http://code.google.com/com/android/download.html>. There are SDK installation versions available for multiple platforms, including Windows, Mac OS X (Intel x86 only), and Linux (i386). Select the latest version of the SDK for the desired platform.

TIP

At the time of writing, the latest Android SDK version is marked: 1.0_r1. This will change from time to time as the Android team releases new versions. In the event that you need to upgrade from one version to another, there will be an upgrade document found on the Android website.

The Android SDK is a compressed folder download. Download and extract the contents of the compressed folder file to a convenient place on your computer. For example, you might install the SDK to `c:\software\google\android-sdk-windows-1.0_r1`, as seen in Figure A.5



As you can see from Figure A.5, the installation footprint is rather simple. Opening the file `documentation.html` in your browser launches the SDK's documentation, which is largely a collection of JavaDocs enumerating the packages and classes of the SDK; the complete documentation source is found in the `docs` folder. The file `android.jar` is the Android runtime Java archive. The `samples` folder contains a number of sample applications, each of which is mentioned in the documentation. The `tools` directory contains Android specific resource compilers and the very helpful `adb` tool. These tools are explained and demonstrated in Chapter 2 of this book.

Both Eclipse and the Android SDK are now installed. It is time to install the Android Development Tools (ADT) plug-in for Eclipse to take advantage of the ADT's powerful features, which will assist in bringing your Android applications to life.

Obtaining and installing the Eclipse plug-in

The following steps demonstrate the installation of the Android plug-in for Eclipse, known as the Android Developer Tools. Note that the most up to date installation directions are available from the Android website. The first steps are somewhat generic for any Eclipse plug-in installation.

Here are the basic steps to install the ADT:

- Run the "Find and Install" feature in Eclipse, found under the **Help > Software Updates** menu, as seen in Figure A.6



Figure A.6: The Eclipse environment supports an extensible plug-in architecture.

- Select the "Search for new features to install" option, as seen in Figure A.7.

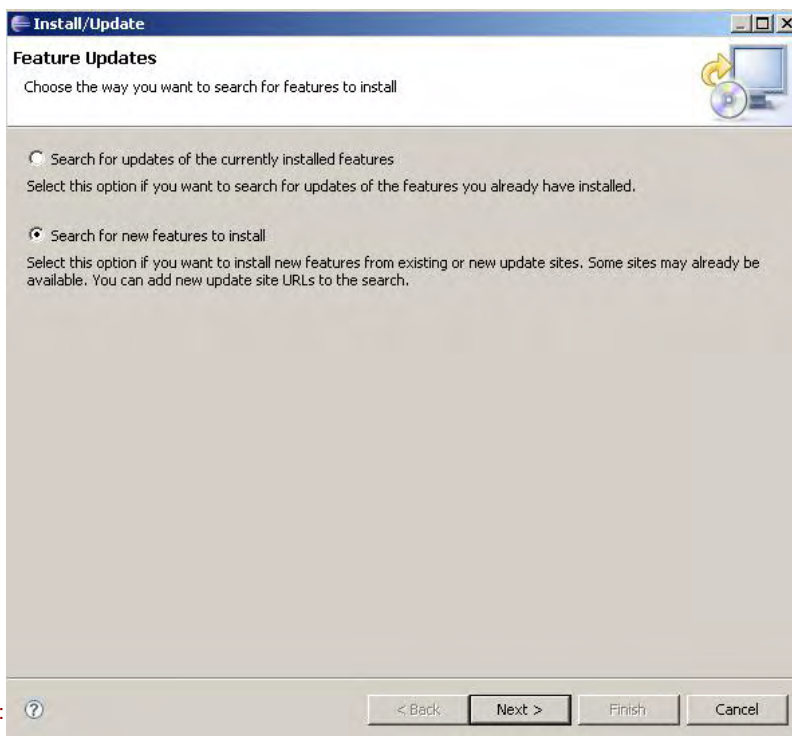


Figure A.7:

- Select "New Remote Site". Give this site a name, such as "Android Developer Tools" as shown in Figure A.8. Use the following URL in the dialog: <https://dl-ssl.google.com/android/eclipse>. Please note the https in the URL.

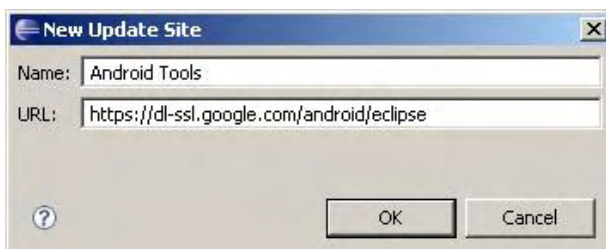


Figure A.8: Create a new update site to search for Android related tools

- A new entry is added to the list and is checked by default. Click on Finish. The search results display the Android Developer Tools.
- Select the Android Developer Tools and click on Next, as seen in Figure A.9

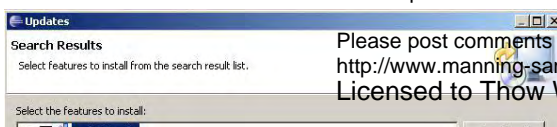


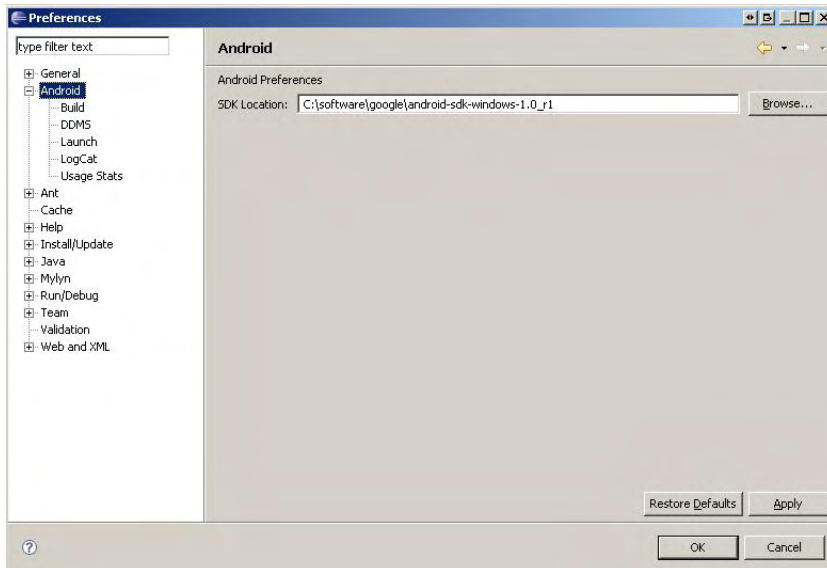
Figure A.9: The Android Tools must be selected for Eclipse to download and install.

- After reviewing and accepting the license agreement, click on Next.
- Review and accept the installation location. Click on Finish.
- The plug-in is now downloaded and installed.
- Eclipse must now be restarted to complete the installation.

Congratulations, the Android Development Tools Eclipse plug-in is installed, however it must now be configured.

Configuring the Eclipse plug-in

Once Eclipse is restarted, it is time to connect the plug-in to the Android SDK installation. Select Preferences under the Window menu in Eclipse. Click on the 'Android' item in the tree view to the left to expand the Android settings. In the right hand pane, specify the SDK installation location. For example, the value used for this appendix is `c:\software\google\android-sdk-windows-1.0_r1`, as shown in Figure A.10.



Once the SDK location is specified, there are five other sections which may be configured:

- The Build section has options for automatically rebuilding resources. Leave this checked. The Build option can change the level of verbosity. Normal is the default setting.

- DDMS - Dalvik Debug Monitor Service is used for peering into a running VM. These settings specify TCP/IP port numbers used for connecting to a running VM with the debugger and various logging levels and options. The default settings should be just fine.
- Launch – This section permits optional emulator switches to be sent to the emulator upon startup. An example of this might be the “wipe-data” option which cleans the persistent file system upon launch.
- LogCat - This is a log file created on the underlying Linux Kernel. The font is selectable in this dialog. Adjust this as desired.
- Usage Stats – this optional feature sends your usage stats to Google to help the Android tools team better understand which features of the plug-in are actually used in an effort to enhance the toolset over time.

Your Android development environment is complete!

Summary

This appendix has walked through the installation of the suggested tools for Android application development, including Eclipse, the Android SDK and the Android Development Tools. Chapter 2 of this book goes into detail on the use of the tools and more specifically, how to use the tools to build, test and debug Android applications. As Android continues to mature, be sure to monitor <http://code.google.com/android> for the most up to date Android development tools and <http://manning.com/android> for updates to this book.