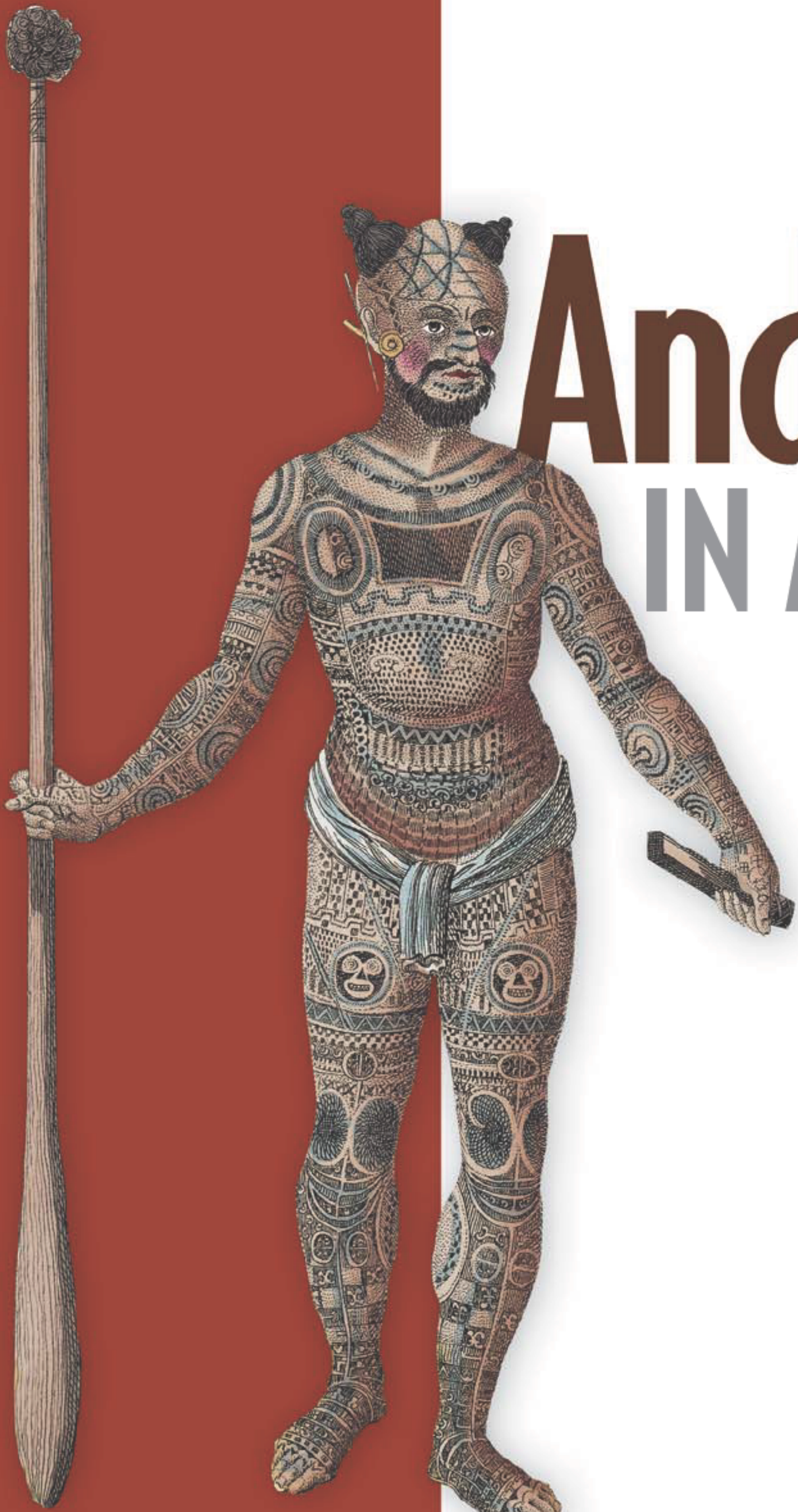
Covers Android 2



Android IN ACTION

SECOND EDITION

W. Frank Ableson
Robi Sen
Chris King

MANNING

Android in Action

SECOND EDITION

W. FRANK ABLESON
ROBI SEN
CHRIS KING

Revised Edition of *Unlocking Android*

MANNING
Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
180 Broad St.
Suite 1323
Stamford, CT 06901
Email: orders@manning.com

∞ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine

Manning Publications Co.
180 Broad St.
Suite 1323
Stamford, CT 06901

Development editor: Troy Mott
Copyeditors: Joan Celmer, Liz Welch
Typesetter: Dottie Marsico
Cover designer: Marija Tudor

ISBN 978-1-935182-72-6
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 – MAL – 16 15 14 13 12 11

brief contents

PART 1 WHAT IS ANDROID?—THE BIG PICTURE 1

PART 2 EXERCISING THE ANDROID SDK 61

PART 3 ANDROID APPLICATIONS 291

contents

6 Networking and web services 159

7 Telephony 187

8 Notifications and alarms 205

18 *Localization 491*

19 *Android Native Development Kit 506*

preface

When we set out to write the first version of this book, many friends and family wondered just what this Android thing was all about. Now, two years after the publication of the first edition, Android is nearly a household term.

The first edition of the book, *Unlocking Android*, enjoyed enough success that we were privileged to have the opportunity to write this second edition, renamed as *Android in Action*. The first thirteen chapters of the book have been refreshed and/or rewritten to bring the content up to date with Android 2.2+. Six chapters were added, bringing in more topics of interest that stray from the simplistic but are still within the realm of instructional and informational. The new content extends beyond the basics of Android development, including some topics that I've envisioned for a long time but lacked the proper platform to bring them to fruition. We could have written many more chapters, but we had to draw the line somewhere!

The second edition of this book was written by Frank Ableson, Robi Sen, and Chris King. Chris updated chapters 4, 5, 7, and 11. Some excellent content originally written by Charlie Collins remains in this second edition. Early on in the project Chris and I were discussing the need to bring social networking into the book. Chris examined the available social networks and came back with a clever mechanism to integrate the Android contacts database with the popular business networking service LinkedIn. His work is shown in chapter 15, "Integration." The application from chapter 15 is available as a free download in the Android Market.

Robi updated his chapters on notifications, graphics, and media, while I focused on some new content areas of interest, including Bluetooth communications, sensors, localization, AppWidgets, native development in C, and web development for Android.

In addition to the LinkedIn application from chapter 15, two more applications from this book are available in the Market as free downloads. The first is SenseBot—an application that allows you to drive a LEGO Mindstorms-powered robot by tilting your phone. The application demonstrates both the sensor subsystem of Android, as well as communicating with Bluetooth. The other application available in the Market is called FindEdges. FindEdges demonstrates the Android Native Development Kit as it exercises an image processing algorithm written in the C language.

All in all, writing a book for Android is both exciting and challenging. Android continues to mature and promises to be a major player for years to come. Many thanks are owed to readers of the first edition, for without you, there wouldn't be a second edition!

FRANK ABLESON

preface to the first edition

The first mobile applications I had the opportunity to work with were inventory control programs used in retail and manufacturing settings. The "terminals," as we called them at the time, were heavy and expensive. They had big antennas, lots of clunky keys, grayscale LCD displays, and they looked like they came straight from the set of a science fiction movie.

From that austere beginning, my mobile horizons expanded when the Palm Pilot became the craze in the mid to late 1990s. My first significant PalmOS project was to develop an IrDA communications library for an application that printed calendars, contacts, and task-lists. Back then, the hip printers had an IrDA port and it was cool to beam your business card to someone. Ironically, I always enjoyed designing and writing the software more than using the devices themselves.

Fast forward ten years, and I have had the privilege of working on some very challenging and engaging mobile software projects for numerous clients along the way. Much of my career to date can be traced back to relationships stemming from my early mobile development experiences—and what a blessing it has been for me. I just love the question, "would it be possible to…?" And more often than not, the answer has been "Yes!" What I particularly enjoy is helping change the way a business operates or the way problems are solved through the application of mobile software. Mobile technology can and will continue to change the way we live, work, and play…and this brings me to Android and this book.

In the fall of 2007, I was speaking with my friend Troy Mott, who happens to be an editor for Manning, the publisher of this book. Troy and I were discussing the mobile marketplace, something we've been doing for years. We started kicking around the idea of writing a book on Android. The challenge was that Android didn't really exist. Yet. We knew from some of the preliminary information that the platform promised to be open, capable, and popular. We felt that those ingredients could make for an interesting and valuable topic, so we began thinking about what that book might look like, taking it on faith that the platform would actually come to fruition.

Before long, we convinced ourselves (and Manning) that this was a good idea and the work began in early 2008. Beyond the usual challenges of putting a book together, we had the additional obstacle that our subject matter has been in a steady, though unpredictable, state of change over the past year. In essence, we've written this book twice because the SDK has been changed multiple times and Android-equipped phones have become available, accelerating the interest and demand for the platform. Every time a significant change occurred, we went back and revisited portions of the book, sometimes rewriting entire chapters to accommodate the latest developments in the Android platform.

I say "we" because in the process of writing this book, Troy and I decided to share the fun and brought in two experienced authors to contribute their expertise and enthusiasm for this platform. It has been a pleasure getting to know and working with both Charlie Collins and Robi Sen. While I focused on the first and third parts of the book in the first edition, Charlie and Robi wrote part 2, which covers the important fundamentals of writing Android applications. Thanks to their contributions, I enjoyed the freedom to express my vision of what Android means to the mobile space in the first part of the book, and then to work on a couple of more advanced applications at the end of the book.

We hope that you enjoy reading this book and that it proves to be a valuable resource for years to come as together we contribute to the future of the Android platform.

FRANK ABLESON

acknowledgments

Perhaps the only thing more challenging than writing a technical book is writing the second edition. There is a lot of excitement when writing the proposed table of contents for the updated edition but at some point the work must commence. The size and scope of this project meant working together as a team from the start. I had the privilege of working again with Robi Sen from the first edition and also with experienced developer and writer Chris King. Along with the help of the talented team at Manning, we are pleased to present *Android in Action*, the update to *Unlocking Android*.

In particular, we'd like to acknowledge and thank those at Manning who helped bring this book about. First, thanks to Troy Mott, our acquisition and development editor, who has been involved in every aspect of both the first and second editions. Troy was there from the beginning, from the "what if" stages, through helping push us over the goal line—twice! Karen Tegtmeyer did all the big and little things to bring the project together; Mary Piergies skillfully piloted the team through the harrowing production process; and Marjan Bace, our publisher, showed an attention to detail at once challenging, beneficial, and appreciated.

Once the writing was done, the next round of work began and special thanks need to go to: Benjamin Berg who performed the pre-production editing pass, Joan Celmer and Liz Welch, our copyeditors, who made our content readable in cases where it went either "too geek" or where the geek in us tried to be "too literary;" Elizabeth Martin, our proofreader, who added common sense to the project, as well as a terrific sense of humor and encouraging attitude; Janet Vail who jumped in at the last minute to help us bring the final pieces of the project together; and finally Dottie Marsico who handles the actual layout of the pages. It is sometimes hard to envision the final product when looking at edits upon edits in MS Word, but Dottie's magic makes the product you hold in your hands. Thanks to each of you for your special contribution to this project. Next, we would like to thank Candace Gillhooley for her efforts in getting the word out about the book.

And special thanks to the other reviewers who read our revised manuscript at different times during its development: Michael Martin, Orhan Alkan, Eric Raymond, Jason Jung, Frank Wang, Robert O'Connor, Paul Grebenc, Sean Owen, Loïc Simon, Greg Donald, Nikolaos Kaintantzis, Matthew Johnson, and Patrick Steger; and to Michael Galpin and Jérôme Bâton for their careful tech review of the final manuscript during production.

Lastly, we want to thank the thoughtful and encouraging MEAP subscribers who provided feedback along the way; the book is better thanks to your contributions.

Frank Ableson

I would like to thank Robi Sen, Chris King, and Troy Mott for their contributions, collaboration, and endurance on this project! And of course, my wife Nikki and my children deserve special recognition for the seemingly endless hours of wondering when I would emerge from the "lab" and what mood I would be in—either elation when the robot worked, or near depression when the AppWidgets wouldn't go away. Thank you for getting neither too excited nor too concerned! My staff at navitend also deserve a big thank you for carrying the water while I finished my work on this project. Finally, a big thank you to Miriam Raffay from Madridiam.com, who provided the much-needed Spanish translations for chapter 18. Gracias!

Chris King

I am deeply grateful to Troy Mott and Frank Ableson for bringing me into this project and providing support and inspiration throughout. Troy has been welcoming and enthusiastic, showing great flexibility as we discussed what projects to undertake. Frank has a keen eye for quality, and provided great guidance from start to finish on how to craft the best book possible. I also appreciate all the work done by the reviewers and editors from Manning, whose contributions have improved the text's accuracy and style. Working on this book has been a joy, and I've greatly enjoyed the opportunities to contribute more and more to its progress.

Thanks also to the crew at Gravity Mobile, especially Noah Hurwitz, Chris Lyon, Young Yoon, and Sam Trychin. You guys keep my life fun and challenging, and have made mobile development an even better place to work. Finally, my love to my family: Charles, Karen, Patrick, Kathryn, and Andrew. You've made everything possible for me.

Robi Sen

I would like to thank Troy Mott and the team—and everyone at Manning Publications— for their hard work making this book something worth reading. I would like to thank my coauthors, Frank and Chris, who were great to work with and very understanding when I was the one holding things up. I would also like to thank Jesse Dailey for his help with OpenGL as well as David Cartier with the Contacts API. Finally, I would like to thank my family who, more often than I liked, had to do without me while I worked on my chapters, worked multiple jobs, and finished grad school.

about this book

Android in Action, Second Edition is a revision and update of *Unlocking Android*, published in April 2009. This book doesn't fit nicely into the camp of "introductory text," nor is it a highly detailed reference manual. The text has something to offer both the beginner and the experienced developer who is looking to sell his or her application in the Android Market. This book covers important beginner topics such as "What is Android" and installing and using the development environment. We then advance to practical working examples of core programming topics any developer will be happy to have at the ready on the reference shelf. The remaining chapters present very detailed example applications covering advanced topics, including a complete field service application, localization, and material on Android web applications, Bluetooth, sensors, AppWidgets, and integration adapters. We even include two chapters on writing applications in C—one for the native side of Android and one using the more generally accepted method of employing the Android Native Development Kit.

Although you can read the book from start to finish, you can also consider it a couple of books in one. If you're new to Android, focus first on chapter 1, appendix A, and then chapter 2. With that foundation, you can then work your way through chapters 3 through 12. Chapter 13 and on are more in-depth in nature and can be read independently of the others.

The audience

We wrote this book for professional programmers and hobbyists alike. Many of the concepts can be absorbed without specific Java language knowledge, though the most value will be found by readers with Java programming skills because Android application programming requires them. A reader with C, C++, or C# programming knowledge will be able to follow the examples.

Prior Eclipse experience is helpful, but not required. A number of good resources are available on Java and Eclipse to augment the content of this book.

Roadmap

This book is divided into four parts. Part 1 contains introductory material about the platform and development environment. Part 2 takes a close look at the fundamental skills required for building Android applications. Part 3 presents a larger scope application and a Native C Android application. Part 4 explores features added to the Android platform, providing examples of leveraging the capable Android platform to create innovative mobile applicatoins.

PART 1: THE ESSENTIALS

Part 1 introduces the Android platform, including its architecture and setting up the development environment.

Chapter 1 delves into the background and positioning of the Android platform, including comparisons to other popular platforms such as BlackBerry, iPhone, and Windows Mobile. After an introduction to the platform, the balance of the first chapter introduces the high-level architecture of Android applications and the operating system environment.

Chapter 2 takes you on a step-by-step development exercise, teaching you the ropes of using the Android development environment, including the key tools and concepts for building an application. If you've never used Eclipse or have never written an Android application, this chapter will prepare you for the next part of the book.

PART 2: THE PROGRAMMING ENVIRONMENT

Part 2 includes an extensive survey of fundamental programming topics in the Android environment.

Chapter 3 covers the fundamental Android UI components, including `View` and `Layout`. We also review the `Activity` in more detail. These are the basic building blocks of screens and applications on the Android platform. Along the way, we also touch on other basic concepts such as handling external resources, dealing with events, and the lifecycle of an Android application.

Chapter 4 expands on the concepts you learned in chapter 3. We delve into the Android `Intent` to demonstrate interaction between screens, activities, and entire applications. We also introduce and use the `Service`, which brings background processes into the fold.

Chapter 5 incorporates methods and strategies for storing and retrieving data locally. The chapter examines use of the filesystem, databases, the SD card, and Android-specific entities such as the `SharedPreferences` and `ContentProvider` classes. At this point, we begin combining fundamental concepts with more real-world details, such as handling application state, using a database for persistent storage, and working with SQLite.

Chapter 6 deals with storing and retrieving data over the network. Here we include a networking primer before delving into using raw networking concepts such as sockets on Android. From there, we progress to using HTTP, and even exploring web services (such as REST and SOAP).

Chapter 7 covers telephony on the Android platform. We touch on basics such as originating and receiving phone calls, as well as more involved topics such as working with SMS. We also cover telephony properties and helper classes.

Chapter 8 looks at how to work with notifications and alarms. In this chapter, we look at how to notify users of various events such as receiving a SMS message, as well as how to manage and set alarms.

Chapter 9 deals with the basics of Android's Graphics API and more advanced concepts such as working with the OpenGL ES library for creating sophisticated 2D and 3D graphics. We also touch on animation.

Chapter 10 looks at Android's support for multimedia; we cover both playing multimedia as well as using the camera and microphone to record your own multimedia files.

Chapter 11 introduces location-based services as we look at an example that combines many of the concepts from the earlier parts of the book in a mapping application. You'll learn about using the mapping APIs on Android, including different location providers and properties that are available, how to build and manipulate map-related screens, and how to work with location-related concepts within the emulator.

PART 3: BRINGING IT ALL TOGETHER

Part 3 contains two chapters, both of which build on knowledge you gained earlier in the text, with a focus on bringing a larger application to fruition.

Chapter 12 demonstrates an end-to-end field service application. The application includes server communications, persistent storage, multiple Activity navigation menus, and signature capture.

Chapter 13 explores the world of native C language applications. The Android SDK is limited to the Java language, although native applications can be written for Android. This chapter walks you through examples of building C language applications for Android, including the use of built-in libraries and TCP socket communications as a Java application connects to your C application. This chapter is useful for developers targeting solutions beyond carrier-subsidized, locked down cell phones.

PART 4: THE MATURING PLATFORM

Part 4 contains six new chapters, each of which represents a more advanced development topic.

Chapter 14 demonstrates the use of both Bluetooth communication and processing sensor data. The sample application accompanying the chapter, SenseBot, permits the user to drive a LEGO Mindstorms robot with their Android phone.

Chapter 15 explores the Android contact database and demonstrates integrating with an external data source. In particular, this application brings Android into the social networking scene by integrating with the popular LinkedIn professional networking service.

Chapter 16 explores the world of web development. Android's browser is based on the open source WebKit engine and brings desktop-like capability to this mobile browser. This chapter equips you to bring attractive and capable web applications to Android.

Chapter 17 brings the "home screen" of your Android application to life by showing you how to build an application that presents its user interface as an AppWidget. In addition to AppWidgets, this chapter demonstrates `BroadcastReceiver`, `Service`, and `Alarms`.

Chapter 18 takes a real-world look at localizing an existing application. Chapter 12's Field Service application is modified to support multiple languages. Chapter 18's version of the Field Service application contains support for both English and Spanish.

Chapter 19 reaches into Android's open source foundation by using a popular edge detection image processing algorithm. The Sobel Edge Detection algorithm is written in C and compiled into a native library. The sample application snaps a picture with the Android camera and then uses this C algorithm to find the edges in the photo.

THE APPENDICES

The appendices contain additional information that didn't fit with the flow of the main text. Appendix A is a step-by-step guide to installing the development environment. This appendix, along with chapter 2, provides all the information you need to build an Android application. Appendix B demonstrates how to prepare and submit an application for the Android Market—an important topic for anyone looking to sell an application commercially.

Code conventions and downloads

All source code in the book is in a `fixed-width font like this`, which sets it off from the surrounding text. For most listings, the code is annotated to point out the key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code. We have tried to format the code so that it fits within the available page space in the book by adding line breaks and using indentation carefully. Sometimes, however, very long lines will include line-continuation markers.

Source code for all the working examples is available from www.manning.com/AndroidinActionSecondEdition or http://www.manning.com/ableson2. A readme.txt file is provided in the root folder and also in each chapter folder; the files provide details on how to install and run the code. Code examples appear throughout this book. Longer listings will appear under clear listing headers while shorter listings will appear between lines of text.

Software requirements

Developing applications for Android may be done from the Windows XP/Vista/7 environment, a Mac OS X (Intel only) environment or a Linux environment. Appendix A includes a detailed description of setting up the Eclipse environment along with the Android Developer Tools plug-in for Eclipse.

A note about the graphics

Many of the original graphics from the first edition, *Unlocking Android,* have been reused in this version of the book. While the title of the revised edition was changed to *Android in Action, Second Edition* during development, we kept the original book title in our graphics and sample applications.

Author Online

Purchase of *Android in Action, Second Edition* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/AndroidinActionSecondEdition or www.manning.com/ableson2. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the cover illustration

The illustration on the cover of *Android in Action, Second Edition* is taken from a French book of dress customs, *Encyclopédie des Voyages* by J. G. St. Saveur, published in 1796. Travel for pleasure was a relatively new phenomenon at the time and illustrated guides such as this one were popular, introducing both the tourist as well as the armchair traveler to the inhabitants of other regions of the world, as well as to the regional costumes and uniforms of France.

The diversity of the drawings in the *Encyclopédie des Voyages* speaks vividly of the uniqueness and individuality of the world's countries and regions just 200 years ago. This was a time when the dress codes of two regions separated by a few dozen miles identified people uniquely as belonging to one or the other, and when members of a social class or a trade or a tribe could be easily distinguished by what they were wearing.

This was also a time when people were fascinated by foreign lands and faraway places, even though they could not travel to these exotic destinations themselves. Dress codes have changed since then and the diversity by region and tribe, so rich at the time, has faded away. It is now often hard to tell the inhabitant of one continent from another. Perhaps, trying to view it optimistically, we have traded a world of cultural and visual diversity for a more varied personal life. Or a more varied and interesting intellectual and technical life.

We at Manning celebrate the inventiveness, the initiative, and the fun of the computer business with book covers based on native and tribal costumes from two centuries ago brought back to life by the pictures from this travel guide.

Part 1

What is Android?— The Big Picture

Android promises to be a market-moving technology platform—not just because of the functionality available in the platform but because of how the platform has come to market. Part 1 of this book brings you into the picture as a developer of the open source Android platform. We begin with a look at the Android platform and the impact it has on each of the major "stakeholders" in the mobile marketplace (chapter 1). We then bring you on board to developing applications for Android with a hands-on tour of the Android development environment (chapter 2).

1

Introducing Android

This chapter covers

- Exploring Android, the open source mobile platform
- Android Intents, the way things work
- Sample application

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

Android is a software platform that's revolutionizing the global cell phone market. It's the first open source mobile application platform that's moved the needle in major mobile markets around the globe. When you're examining Android, there are a number of technical and market-related dimensions to consider. This first section introduces the platform and provides context to help you better understand Android and where it fits in the global cell phone scene.

Android is primarily a Google effort, in collaboration with the Open Handset Alliance. Open Handset Alliance is an alliance of nearly 50 organizations committed to bringing a "better" and more "open" mobile phone to market. Considered a novelty at first by some, Android has grown to become a market-changing player in a few short years, earning both respect and derision alike from peers in the industry.

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

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

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

1.1 The Android platform

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

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

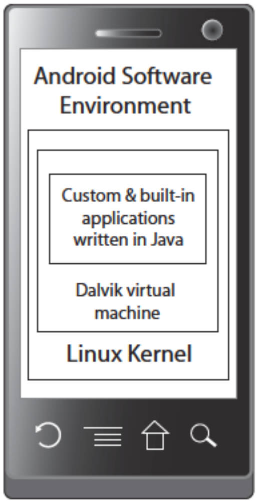

Android Software Environment
Custom & built-in applications written in Java
Dalvik virtual machine
Linux Kernel

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

> **PLATFORM VS. DEVICE** Throughout this book, wherever code must be tested or exercised on a device, a software-based emulator is typically employed. An exception is in chapter 14 where Bluetooth and Sensors are exercised. See chapter 2 for information on how to set up and use the Android emulator.
>
> The term *platform* refers to Android itself—the software—including all the binaries, code libraries, and tool chains. This book focuses on the Android platform; the Android emulators available in the SDK are simply components of the Android platform.

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

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

1.2 Understanding the Android market

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

1.2.1 Mobile operators

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

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

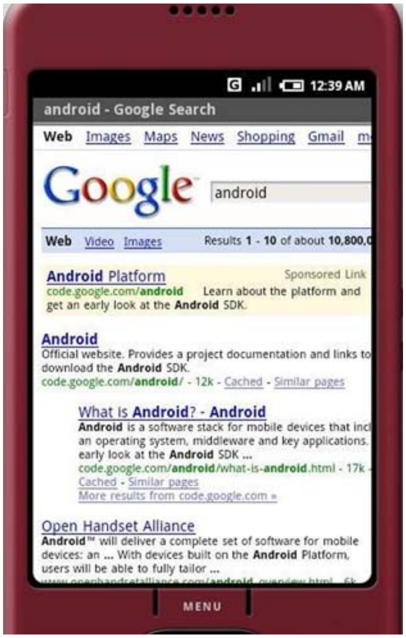
Other mobile operators feel threatened by Google and the potential of "free wireless," driven by advertising revenues and an upheaval of the market. Another challenge for mobile operators is that they want the final say on what services are enabled across their networks. Historically, handset manufacturers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

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

1.2.2 Android vs. the feature phones

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

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



Results **1** - **10** of about **10,800,0**

Android Platform — Sponsored Link
code.google.com/**android** Learn about the platform and get an early look at the **Android** SDK.

Android
Official website. Provides a project documentation and links to download the **Android** SDK.
code.google.com/**android**/ - 12k - Cached - Similar pages

What Is **Android**? - **Android**
Android is a software stack for mobile devices that inc an operating system, middleware and key applications. early look at the **Android** SDK ...
code.google.com/**android**/what-is-**android**.html - 17k
Cached - Similar pages
More results from code.google.com »

Open Handset Alliance
Android™ will deliver a complete set of software for mobile devices: an ... With devices built on the **Android** Platform, users will be able to fully tailor ...

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

[1] Only 12% of phones sold in the fourth quarter of 2008 were smartphones: http://www.gartner.com/it/page.jsp?id=910112.

penetrating this end of the market. Chapter 16 takes a close look at using web development skills for creating Android applications.

> **WEBKIT** The WebKit (http://www.webkit.org) browser engine is an open source project that powers the browser found in Macs (Safari) and is the engine behind Mobile Safari, which is the browser on the iPhone. It's 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 the lower end of the market generally falls into one of two camps:

- *Qualcomm's BREW environment*—BREW stands for Binary Runtime Environment for Wireless. For a high-volume example of BREW technology, consider Verizon's Get It Now-capable devices, which run on this platform. The challenge for software developers who want to gain access to this market is that the bar to get an application on this platform is high, because everything is managed by the mobile operator, with expensive testing and revenue-sharing fee structures. The upside to this platform is that the mobile operator collects the money and disburses it to the developer after the sale, and often these sales recur monthly. Just about everything else is a challenge to the software developer. Android's open application environment is more accessible than BREW.
- *Java ME*, or *Java Platform, Micro Edition*—A popular platform for this class of device. The barrier to entry is much lower for software developers. Java ME developers will find a same-but-different environment in Android. Android isn't strictly a Java ME-compatible platform, but the Java programming environment found in Android is a plus for Java ME developers. There are some projects underway to create a bridge environment, with the aim of enabling Java ME applications to be compiled and run for Android. Gaming, a better browser, and anything to do with texting or social applications present fertile territory for Android at this end of the market.

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

1.2.3 Android vs. the smartphones

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

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

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

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

1.2.4 Android vs. itself

Android will likely always be an open source project, but to succeed in the mobile market, it must sell millions of units and stay fresh. Even though Google briefly entered the device fray with its Nexus One phone, it's not a hardware company. From necessity, Android is sold by others such as HTC and Motorola, to name the big players. These manufacturers start with the Android Open Source Platform (AOSP), but extend it to meet their need to differentiate their offerings. Android isn't the first open source phone, but it's the first from a player with the market-moving weight of Google leading the charge. This market leadership position has already translated to impressive unit sales across multiple manufacturers. So, now that there are a respectable number of devices on the market, can Android keep it together and avoid fragmentation?

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

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

1.2.5 Licensing Android

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

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

Selling applications

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

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

1.3 The layers of Android

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

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

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

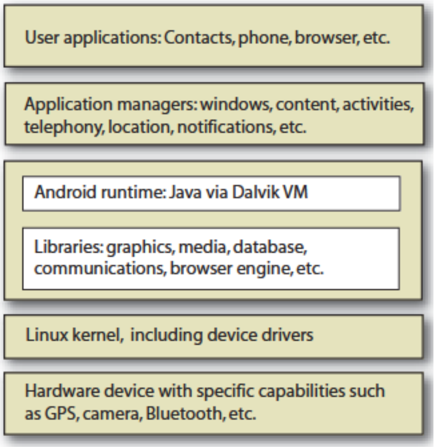

User applications: Contacts, phone, browser, etc.
Application managers: windows, content, activities, telephony, location, notifications, etc.
Android runtime: Java via Dalvik VM
Libraries: graphics, media, database, communications, browser engine, etc.
Linux kernel, including device drivers
Hardware device with specific capabilities such as GPS, camera, Bluetooth, etc.

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

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

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

1.3.1 Building on the Linux kernel

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

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

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

1.3.2 Running in the Dalvik VM

The Dalvik VM is an example of the need for efficiency, the desire for a rich programming environment, and even some intellectual property constraints, colliding, with innovation as the result. Android's Java environment provides a rich application platform and is accessible because of the popularity of Java itself. Also, application performance, particularly in a low-memory setting such as you find in a mobile phone, is paramount for the mobile market. But this isn't the only issue at hand.

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

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

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

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

1.4 The Intent of Android development

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

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

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

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

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

1.4.1 Empowering intuitive UIs

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

Beyond being an effective way to get access to data, why is this URI topic important, and what does it have to do with `Intents`? The answer is nontechnical but crucial: The way a mobile user navigates on the platform is crucial to its commercial success. Platforms that replicate the desktop experience on a mobile device are acceptable to only a small percentage of hardcore power users. Deep menus and multiple taps and clicks are generally not well received in the mobile market. The mobile application, more than in any other market, demands intuitive ease of use. A consumer might buy a device based on cool features that were enumerated in the marketing materials, but that same consumer is unlikely to even touch the instruction manual. A UI's usability is highly correlated with its market penetration. UIs are also a reflection of the platform's data access model, so if the navigation and data models are clean and intuitive, the UI will follow suit.

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

1.4.2 Intents and how they work

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

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

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

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

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

Table 1.1 Commonly employed URIs in Android

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

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

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

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

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

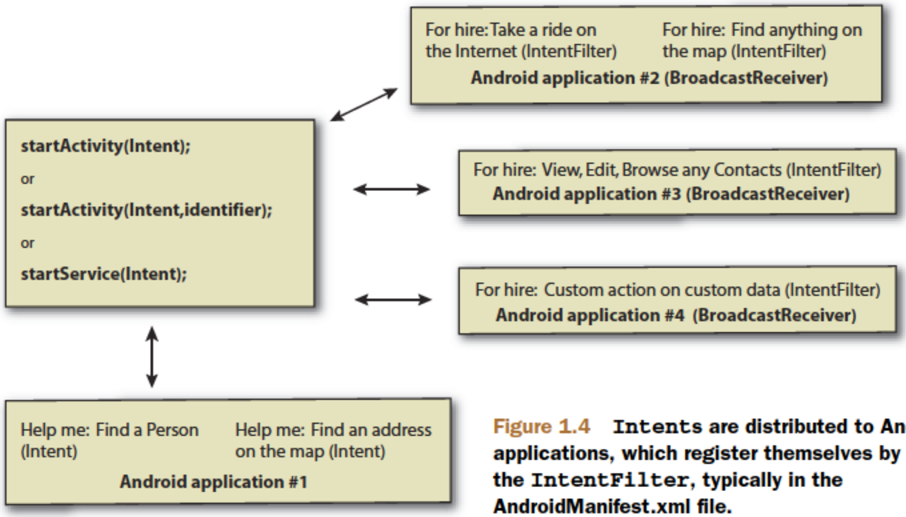

For hire: Take a ride on the Internet (IntentFilter)
For hire: Find anything on the map (IntentFilter)
Android application #2 (BroadcastReceiver)
startActivity(Intent);
or
startActivity(Intent,identifier);
or
startService(Intent);
For hire: View, Edit, Browse any Contacts (IntentFilter)
Android application #3 (BroadcastReceiver)
For hire: Custom action on custom data (IntentFilter)
Android application #4 (BroadcastReceiver)
Help me: Find a Person (Intent)
Help me: Find an address on the map (Intent)
Android application #1

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

identifier. To carry out these kinds of tasks, you create an `Intent` with the action set to `ACTION_VIEW` and a URI that represents the specific person of interest.

Here are some examples:

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

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

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

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

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

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

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

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

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

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

The power and the complexity of Intents

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

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

1.5 Four kinds of Android components

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

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

1.5.1 Activity

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

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

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

Listing 1.1 A basic Activity in an Android application

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

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

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

The Activity represents a visible application component within Android. With assistance from the View class, which we'll cover in chapter 3, the Activity is the most

You say Intent; I say Intent

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

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

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

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

commonly employed Android application component. The next topic of interest is the Service, which runs in the background and doesn't generally present a direct UI.

1.5.2 Service

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

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

Listing 1.2 A simple example of an Android Service

```
package com.msi.manning.chapter1;
import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
public class Service1 extends Service implements Runnable {         1 Extend Service class
public static final String tag = "service1";
    private int counter = 0;
    @Override
    protected void onCreate()  {                                2 Initialization
          super.onCreate();
   Thread aThread = new Thread (this);
          aThread.start();
    }
    public void run() {
             while (true) {
               try {
               Log.i(tag,"service1 firing : # " + counter++);
               Thread.sleep(10000);
               } catch(Exception ee) {
                Log.e(tag,ee.getMessage());
               }
             }
    }
@Override                                            ❸ Handle
public IBinder onBind(Intent intent)    {          ⊲  Binding request
return null;
}

}
```

This example requires that the package android.app.Service be imported. This package contains the Service class. This example also demonstrates Android's logging mechanism android.util.Log, which is useful for debugging purposes. (Many examples in this book include using the logging facility. We'll discuss logging in more depth in chapter 2.) The Service1 class ❶ extends the Service class. This class implements the Runnable interface to perform its main task on a separate thread. The onCreate method ❷ of the Service class permits the application to perform initialization-type tasks. We're going to talk about the onBind() method ❸ in further detail in chapter 4, when we'll explore the topic of interprocess communication in general.

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

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

1.5.3 BroadcastReceiver

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

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

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

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

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

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

Listing 1.3 A sample BroadcastReceiver

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

❶ Tag used in logging

❷ Check Intent's action

We need to discuss a few items in this listing. The class MySMSMailBox extends the BroadcastReceiver class. This subclass approach is the most straightforward way to employ a BroadcastReceiver. (Note the class name MySMSMailBox; it'll be used in the AndroidManifest.xml file, shown in listing 1.4.) The tag variable ❶ is used in conjunction with the logging mechanism to assist in labeling messages sent to the console log on the emulator. Using a tag in the log enables us to filter and organize log messages in the console. (We discuss the log mechanism in more detail in chapter 2.) The onReceive method is where all the work takes place in a BroadcastReceiver; you must implement this method. A given BroadcastReceiver can register multiple IntentFilters. A BroadcastReceiver can be instantiated for an arbitrary number of Intents.

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

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

Listing 1.4 AndroidManifest.xml

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

Required permission ❶

❷ **Receiver tag; note dot prefix**

Certain tasks within the Android platform require the application to have a designated privilege. To give an application the required permissions, use the `<uses-permission>` tag ❶. (We'll discuss this tag in detail in section 1.6.) The `<receiver>` tag contains the class name of the class implementing the `BroadcastReceiver`. In this example, the class name is `MySMSMailBox`, from the package `com.msi.manning.unlockingandroid`. Be sure to note the dot that precedes the name ❷. This dot is required. If your application isn't behaving as expected, one of the first places to check is your Android.xml file, and look for the dot before the class name! The `IntentFilter` is defined in the `<intent-filter>` tag. The desired action in this

Testing SMS

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

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

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

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

We'll discuss these tools in more detail in chapter 2.

example is android.provider.Telephony.SMS_RECEIVED. The Android SDK contains the available actions for the standard Intents. Also, remember that user applications can define their own Intents, as well as listen for them.

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

1.5.4 ContentProvider

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

A ContentProvider can use any form of data storage mechanism available on the Android platform, including files, SQLite databases, or even a memory-based hash map if data persistence isn't required. The ContentProvider is a data layer that provides data abstraction for its clients and centralizing storage and retrieval routines in a single place.

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

Data stored in a ContentProvider can be traditional data types, such as integers and strings. Content providers can also manage binary data, such as image data. When binary data is retrieved, the suggested best practice is to return a string representing the filename that contains the binary data. If a filename is returned as part of a ContentProvider query, the application shouldn't access the file directly; you should use the helper class, ContentResolver's openInputStream method, to access the binary data. This approach navigates the Linux process and security hurdles, as well as keeps all data access normalized through the ContentProvider. Figure 1.5 outlines the relationship among ContentProviders, data stores, and their clients.

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

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

From this point, accessing a ContentProvider is similar to using Structured Query Language (SQL) in other platforms, though a complete SQL statement isn't employed. A query is submitted to the ContentProvider, including the columns

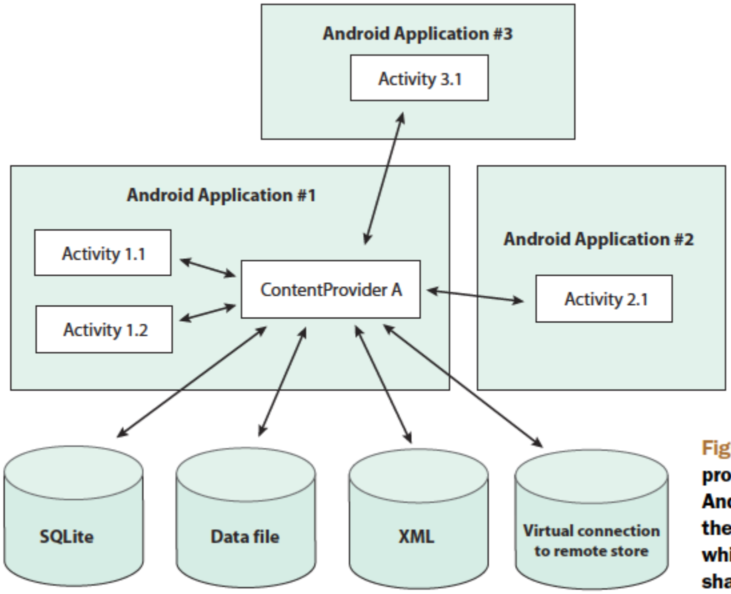


Figure 1.5 The content provider is the data tier for Android applications and is the prescribed manner in which data is accessed and shared on the device.

desired and optional `Where` and `Order By` clauses. Similar to parameterized queries in traditional SQL, parameter substitution is also supported when working with the `ContentProvider` class. Where do the results from the query go? In a `Cursor` class, naturally. We'll provide a detailed `ContentProvider` example in chapter 5.

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

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

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

1.6 Understanding the AndroidManifest.xml file

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

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

Listing 1.5 AndroidManifest.xml file for a basic Android application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.unlockingandroid">
    <application android:icon="@drawable/icon">
       <activity android:name=".Activity1" android:label="@string/app_name">
           <intent-filter>
               <action android:name="android.intent.action.MAIN" />
               <category android:name="android.intent.category.LAUNCHER" />
           </intent-filter>
       </activity>
    </application>
</manifest>
```

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

In addition to the elements used in the sample manifest file shown in listing 1.5, other common tags are:

- The <service> tag represents a Service. The attributes of the <service> tag include its class and label. A Service might also include the <intent-filter> tag.

- The `<receiver>` tag represents a `BroadcastReceiver`, which might have an explicit `<intent-filter>` tag.
- The `<uses-permission>` tag tells Android that this application requires certain security privileges. For example, if an application requires access to the contacts on a device, it requires the following tag in its AndroidManifest.xml file:

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

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

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

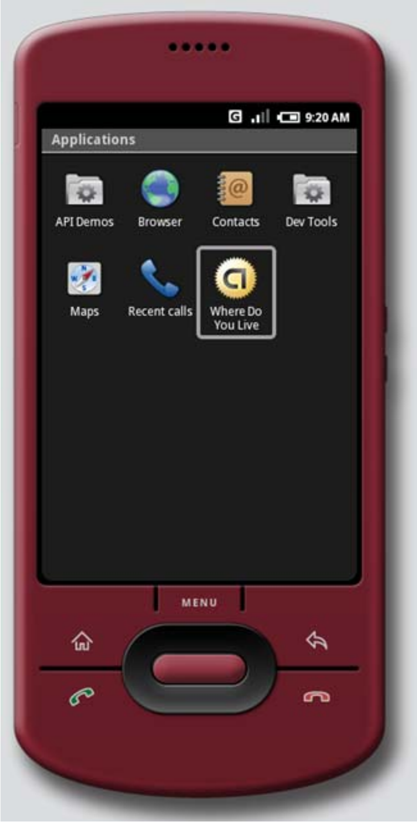


Figure 1.6 **Applications are listed in the launcher based on their `IntentFilter`. In this example, the application Where Do You Live is available in the `LAUNCHER` category.**

1.7 Mapping applications to processes

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

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

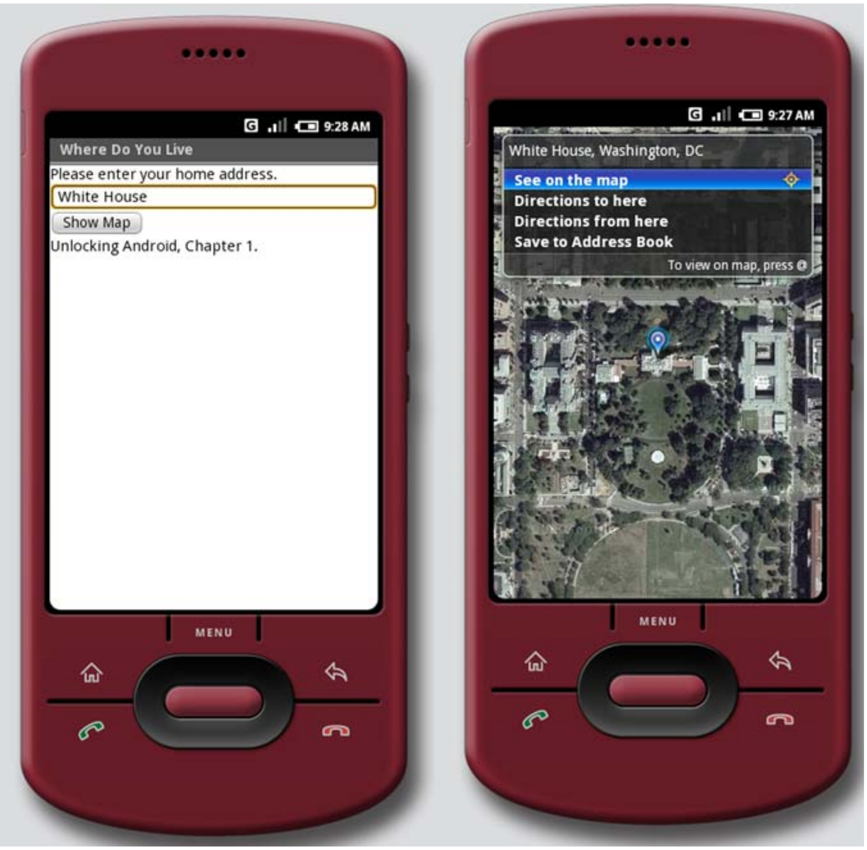
Let's apply some of what you've learned by building your first Android application.

ps -a

The Linux environment is complete, including process management. You can launch and kill applications directly from the shell on the Android platform, but this is a developer's debugging task, not something the average Android handset user is likely to carry out. It's nice to have this option for troubleshooting application issues. It's a relatively recent phenomenon to be able to touch the metal of a mobile phone in this way. For more in-depth exploration of the Linux foundations of Android, see chapter 13.

1.8 Creating an Android application

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



Where Do You Live

Please enter your home address.

White House

Unlocking Android, Chapter 1.

White House, Washington, DC

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

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.unlockingandroid">
    <application android:icon="@drawable/icon">
        <activity android:name=".AWhereDoYouLive"
android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category
 android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
<uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

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

Listing 1.7 Implementing the Android `Activity` in AWhereDoYouLive.java

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

                }
            }
        });
    }
}
```

❶ Get address

❷ Prepare Intent

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

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

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

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

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

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

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */
package com.msi.manning.unlockingandroid;
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int icon=0x7f020000;
    }
    public static final class id {
        public static final int address=0x7f050000;
        public static final int launchmap=0x7f050001;
    }
    public static final class layout {
        public static final int main=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
    }
}
```

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

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

Listing 1.9 Main.xml defines the UI elements for our sample application

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Please enter your home address."
    />
<EditText
    android:id="@+id/address"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
android:autoText="true"
/>
<Button
    android:id="@+id/launchmap"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Show Map"
    />
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Unlocking Android, Chapter 1."
    />
</LinearLayout>
```

❶ **ID assignment for EditText**

❷ **ID assignment for Button**

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

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

Listing 1.10 strings.xml

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

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

1.9 Summary

This chapter introduced the Android platform and briefly touched on market positioning, including what Android is up against in the rapidly changing mobile marketplace. Inside two short years, the Android SDK has been announced, released, and updated no fewer than four times. And that's just the software. Major device manufacturers have now signed on to the Android platform and have brought capable devices to market, including a privately labeled device from Google itself. Android's future continues to brighten.

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

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

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

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

2 Android's development environment

This chapter covers

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

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

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

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

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

2.1 Introducing the Android SDK

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

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

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

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

2.1.1 Core Android packages

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

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

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

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

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

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

2.1.2 Optional packages

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

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

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

2.2 Exploring the development environment

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

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

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

Eclipse supports the concept of *perspectives*, where the layout of the screen has a set of related windows and tools. The windows and tools included in an Eclipse perspective

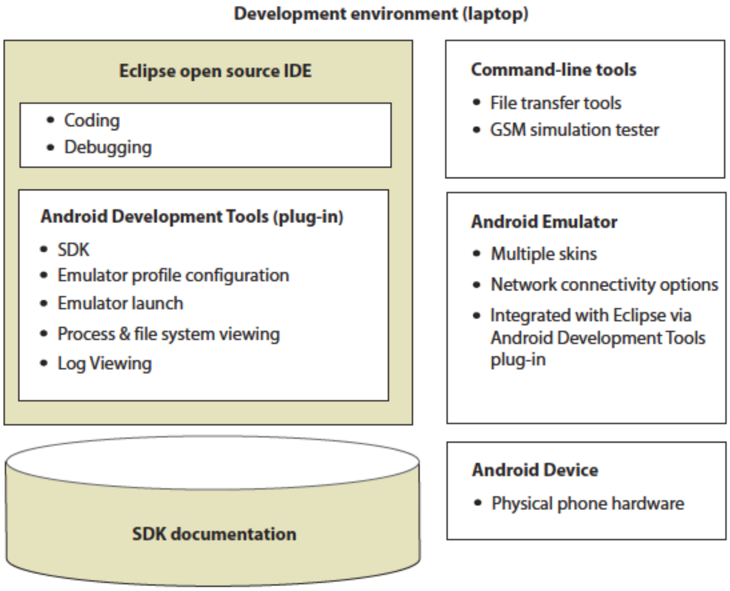

Development environment (laptop)
Eclipse open source IDE
• Coding
• Debugging
Android Development Tools (plug-in)
• SDK
• Emulator profile configuration
• Emulator launch
• Process & file system viewing
• Log Viewing
SDK documentation
Command-line tools
• File transfer tools
• GSM simulation tester
Android Emulator
• Multiple skins
• Network connectivity options
• Integrated with Eclipse via Android Development Tools plug-in
Android Device
• Physical phone hardware

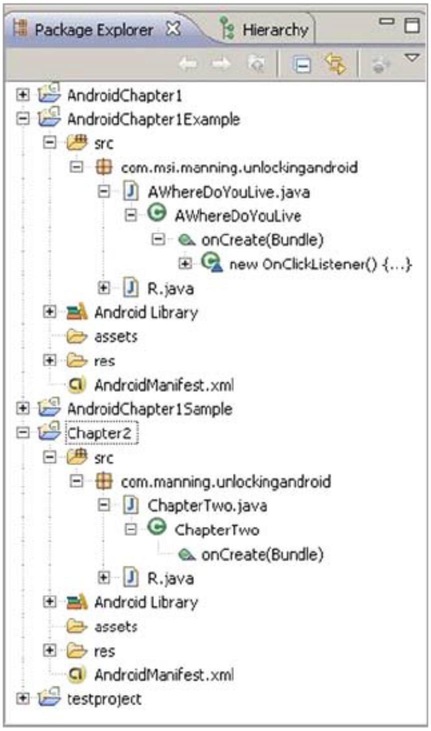
Figure 2.1 The development environment for building Android applications, including the popular open source Eclipse IDE

are known as *views*. When developing Android applications, there are two Eclipse perspectives of primary interest to us: the Java perspective and the DDMS perspective. Beyond those two, the Debug perspective is also available and useful when you're debugging an Android application; we'll talk about the Debug perspective in section 2.5. To switch between the available perspectives in Eclipse, use the Open Perspective menu, under the Window menu in the Eclipse IDE.

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

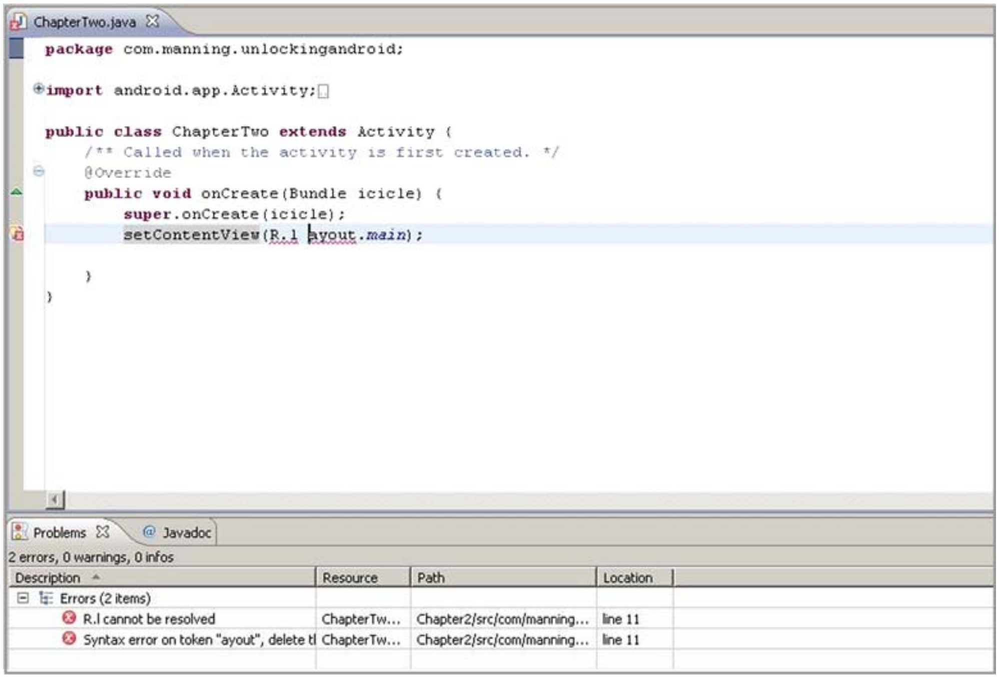
2.2.1 The Java perspective

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



```
AndroidChapter1
AndroidChapter1Example
    src
        com.msi.manning.unlockingandroid
            AWhereDoYouLive.java
                AWhereDoYouLive
                    onCreate(Bundle)
                        new OnClickListener() {...}
            R.java
    Android Library
    assets
    res
    AndroidManifest.xml
AndroidChapter1Sample
Chapter2
    src
        com.manning.unlockingandroid
            ChapterTwo.java
                ChapterTwo
                    onCreate(Bundle)
            R.java
    Android Library
    assets
    res
    AndroidManifest.xml
testproject
```

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



```
package com.manning.unlockingandroid;

import android.app.Activity;

public class ChapterTwo extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.l ayout.main);

    }
}
```

2 errors, 0 warnings, 0 infos

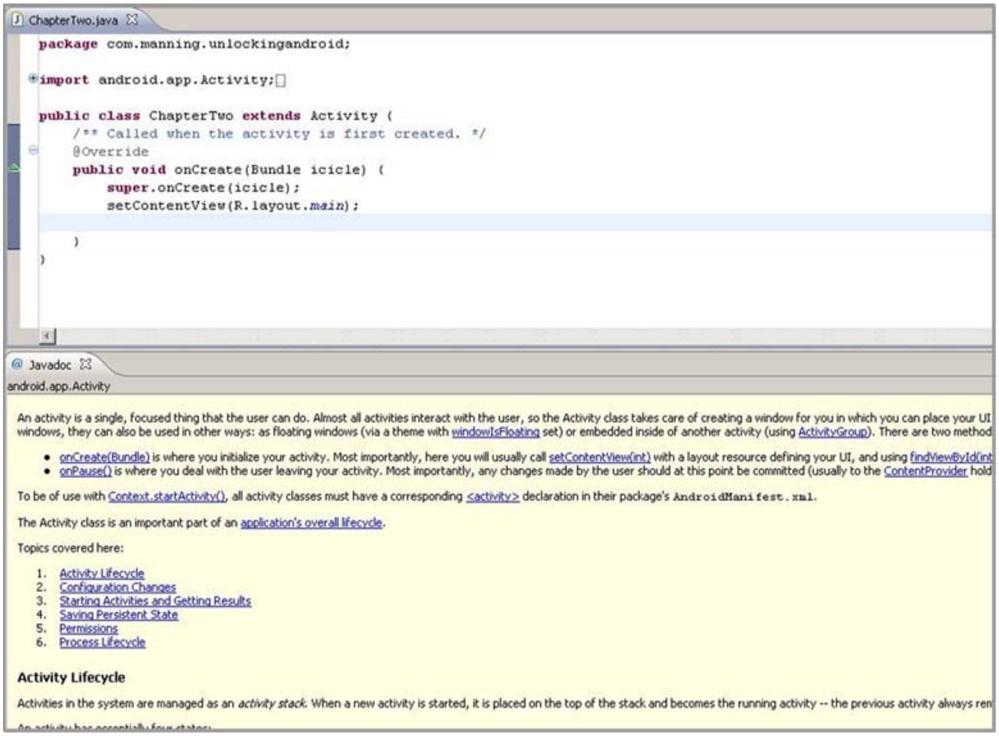
Description	Resource	Path	Location
Errors (2 items)			
R.l cannot be resolved	ChapterTw...	Chapter2/src/com/manning...	line 11
Syntax error on token "ayout", delete tl	ChapterTw...	Chapter2/src/com/manning...	line 11

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

The Java perspective is where you'll edit your Java source code. Every time you save your source file, it's automatically compiled by Eclipse's Java development tools (JDT) in the background. You don't need to worry about the specifics of the JDT; the important thing to know is that it's functioning in the background to make your Java experience as seamless and painless as possible. If there's an error in your source code, the details will show up in the Problems view of the Java perspective. Figure 2.3 has an intentional error in the source code to demonstrate the Problems view. You can also put your mouse over the red *x* to the left of the line containing the error for a tool-tip explanation of the problem.

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

> **TIPS** This chapter scratches the surface in introducing the powerful Eclipse environment. To learn more about Eclipse, you might consider reading *Eclipse in Action: A Guide for Java Developers,* by David Gallardo, Ed Burnette, and Robert McGovern, published by Manning and available online at http://www.manning.com/.
>
> It's easy to get the views in the current perspective into a layout that isn't what you really want. If this occurs, you have a couple of choices to restore



```
package com.manning.unlockingandroid;

import android.app.Activity;

public class ChapterTwo extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);

    }
}
```

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

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

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

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

Topics covered here:

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

Activity Lifecycle

Activities in the system are managed as an *activity stack*. When a new activity is started, it is placed on the top of the stack and becomes the running activity -- the previous activity always ren

An activity has essentially four states:

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

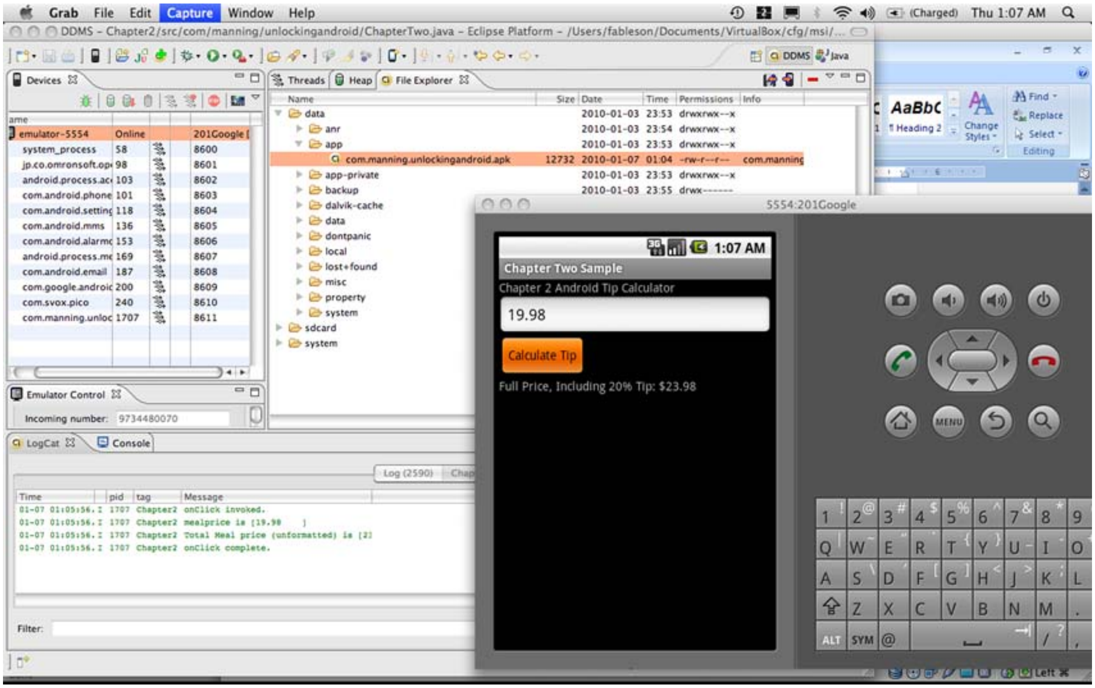
the perspective to a more useful state. You can use the Show View menu under the Window menu to display a specific view or you can select the Reset Perspective menu to restore the perspective to its default settings.

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

2.2.2 The DDMS perspective

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

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


Grab File Edit Capture Window Help
(Charged) Thu 1:07 AM
DDMS - Chapter2/src/com/manning/unlockingandroid/ChapterTwo.java - Eclipse Platform - /Users/fableson/Documents/VirtualBox/cfg/msi/...
DDMS
Java
Devices
Threads
Heap
File Explorer
Emulator Control
LogCat
Console
5554:201Google
MENU
ALT
SYM
AaBbC
Heading 2
Change Styles
Find
Replace
Select
Editing

ame				
emulator-5554	Online			201Google [
system_process	58			8600
jp.co.omronsoft.op	98			8601
android.process.ac	103			8602
com.android.phone	101			8603
com.android.setting	118			8604
com.android.mms	136			8605
com.android.alarmc	153			8606
android.process.me	169			8607
com.android.email	187			8608
com.google.androic	200			8609
com.svox.pico	240			8610
com.manning.unloc	1707			8611

```
Name                                  Size   Date        Time   Permissions  Info
data                                         2010-01-03  23:53  drwxrwx--x
  anr                                        2010-01-03  23:54  drwxrwx--x
  app                                        2010-01-03  23:53  drwxrwx--x
    com.manning.unlockingandroid.apk  12732  2010-01-07  01:04  -rw-r--r--   com.manning
  app-private                                2010-01-03  23:53  drwxrwx--x
  backup                                     2010-01-03  23:55  drwx------
  dalvik-cache
  data
  dontpanic
  local
  lost+found
  misc
  property
  system
sdcard
system
```

1:07 AM

Chapter Two Sample

Chapter 2 Android Tip Calculator

19.98

Calculate Tip

Full Price, Including 20% Tip: $23.98

Incoming number: 9734480070

Log (2590)

```
Time               pid   tag       Message
01-07 01:05:56.I   1707  Chapter2  onClick invoked.
01-07 01:05:56.I   1707  Chapter2  mealprice is [19.98     ]
01-07 01:05:56.I   1707  Chapter2  Total Meal price (unformatted) is [2]
01-07 01:05:56.I   1707  Chapter2  onClick complete.
```

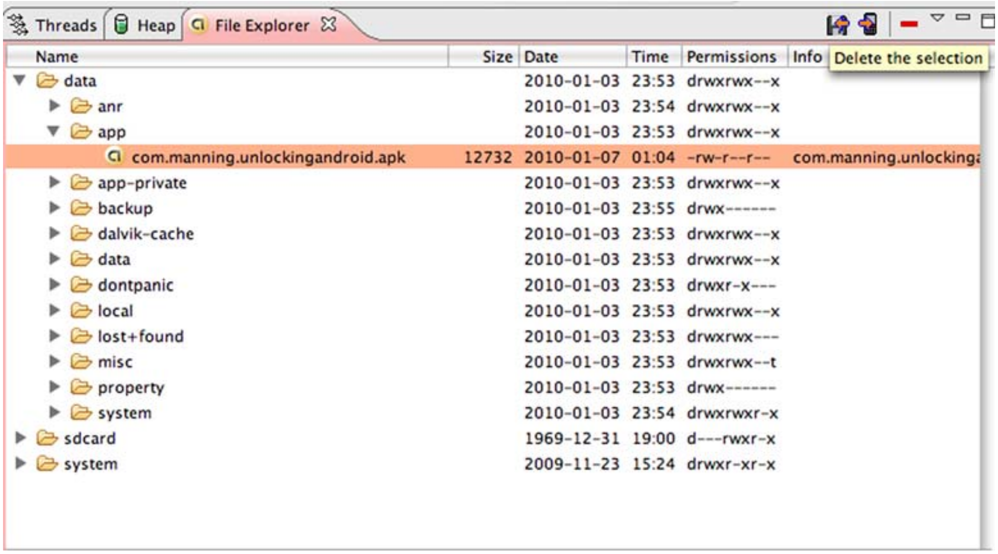
Filter:

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

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

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

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



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

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

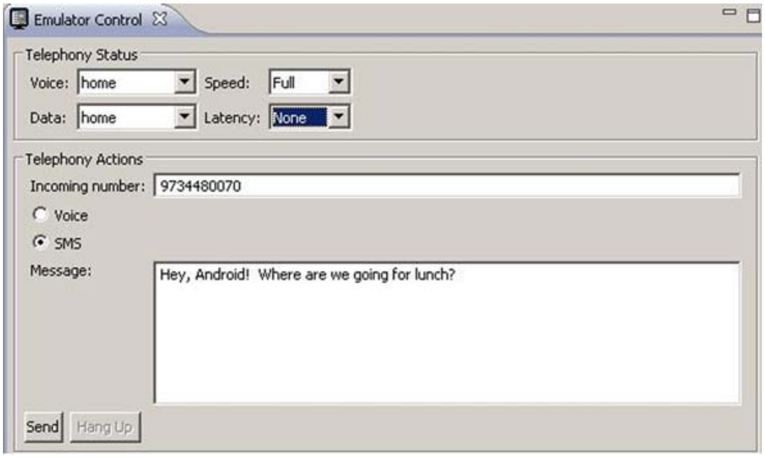
application contributing the log entry. As expected, the PID for our log entries is 616, matching our running application instance in the emulator.

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

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

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

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



Emulator Control

Telephony Status

Voice: home Speed: Full

Data: home Latency: None

Telephony Actions

Incoming number: 9734480070

○ Voice

● SMS

Message: Hey, Android! Where are we going for lunch?

Figure 2.7 Sending a test SMS to the Android emulator

2.2.3 Command-line tools

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

> **TIP** It's a good idea to add the tools directory to your search path. For example, if your Android SDK is installed to c:\software\google\ androidsdk, you can add the Android SDK to your path by performing the following operation in a command window on your Windows computer:
>
> ```
> set path=%path%;c:\software\google\androidsdk\tools;
> ```
>
> Or use the following command for Mac OS X and Linux:
>
> ```
> export PATH=$PATH:/path_to_Android_SDK_directory/tools
> ```

ANDROID ASSET PACKAGING TOOL

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

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

ANDROID DEBUG BRIDGE

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

```
adb devices<return>
```

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

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

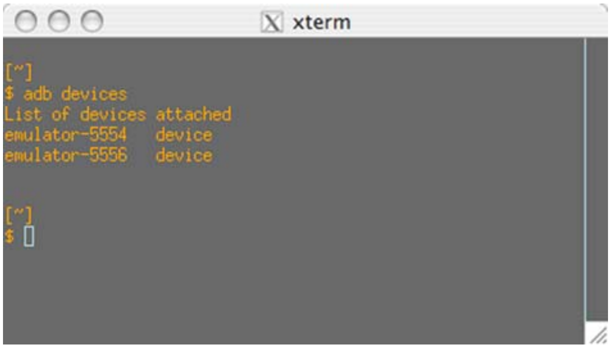
```
adb –s "serialnumber" shell
```

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

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

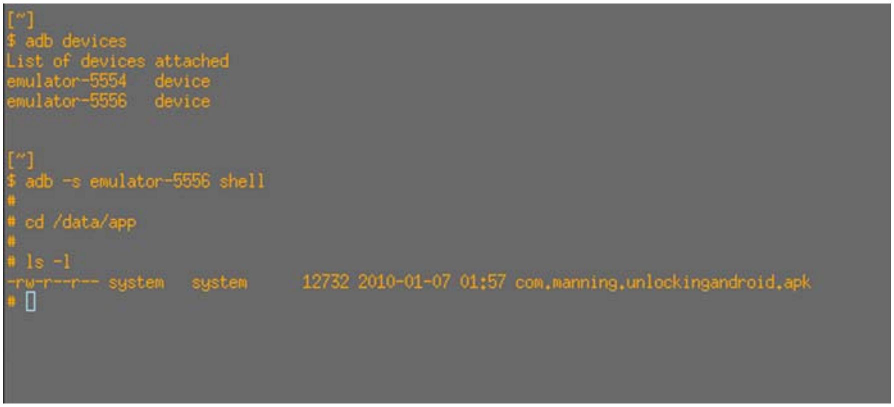
```
adb [-s serialnumber] shell install someapplication.apk
```

This command installs the application named `someapplication` to the Android emulator. The file is copied to the /data/app directory and is accessible from the Android application launcher. Similarly, if you want to remove an application, you can run `adb` to remove an application from the Android emulator. If you want to remove the



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

[~]
$ 
```

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



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


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

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

com.manning.unlockingandroid.apk sample application from a running emulator's filesystem, for example, you can execute the following command from a terminal or Windows command window:

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

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

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

TELNET

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

```
telnet localhost 5554
```

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

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

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

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

2.3 Building an Android application in Eclipse

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

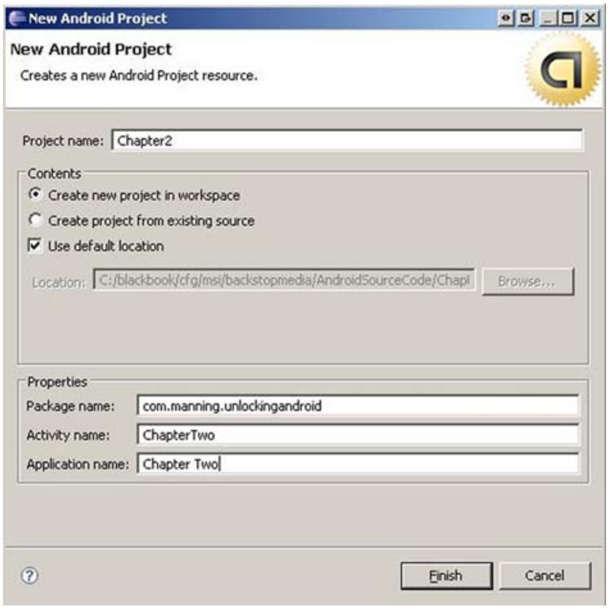
Building an Android application isn't much different from creating other types of Java applications in the Eclipse IDE. It all starts with choosing File > New and selecting an Android application as the build target.

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

2.3.1 The Android Project Wizard

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

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



New Android Project

New Android Project

Creates a new Android Project resource.

Project name: Chapter2

Contents

● Create new project in workspace

○ Create project from existing source

☑ Use default location

Location: C:/blackbook/cfg/msi/backstopmedia/AndroidSourceCode/Chapl

Properties

Package name: com.manning.unlockingandroid

Activity name: ChapterTwo

Application name: Chapter Two

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

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

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

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

2.3.2 Android sample application code

The Android 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, you can see all the elements of this application. Here's a quick description of the elements included in the sample application:

- The src folder contains two Java source files automatically created by the wizard.
- ChapterTwo.java contains the main `Activity` for the application. You'll modify this file to add the sample application's tip calculator functionality.
- R.java contains identifiers for each of the UI resource elements in the application. Never modify this file directly. It automatically regenerates every time a resource is modified; any manual changes you make will be lost the next time the application is built.

- Android.jar contains the Android runtime Java classes. This reference to the android.jar file found in the Android SDK ensures that the Android runtime classes are accessible to your application.
- The res folder contains all the Android resource folders, including:
  - Drawables contains image files such as bitmaps and icons. The wizard provides a default Android icon named icon.png.
  - Layout contains an XML file called main.xml. This file contains the UI elements for the primary view of your `Activity`. In this example, you'll modify this file but you won't make any significant or special changes—just enough to accomplish the meager UI goals for your tip calculator. We cover UI elements, including `Views`, in detail in chapter 3. It's not uncommon for an Android application to have multiple XML files in the Layout section of the resources.
  - Values contains the strings.xml file. This file is used for localizing string values, such as the application name and other strings used by your application.

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

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

Listing 2.1 main.xml contains UI elements

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Chapter 2 Android Tip Calculator"
    />
<EditText
    android:id="@+id/mealprice"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoText="true"
/>
<Button
android:id="@+id/calculate"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Calculate Tip"
    />
<TextView
    android:id="@+id/answer"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text=""
    />

</LinearLayout>
```

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

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

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

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

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

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

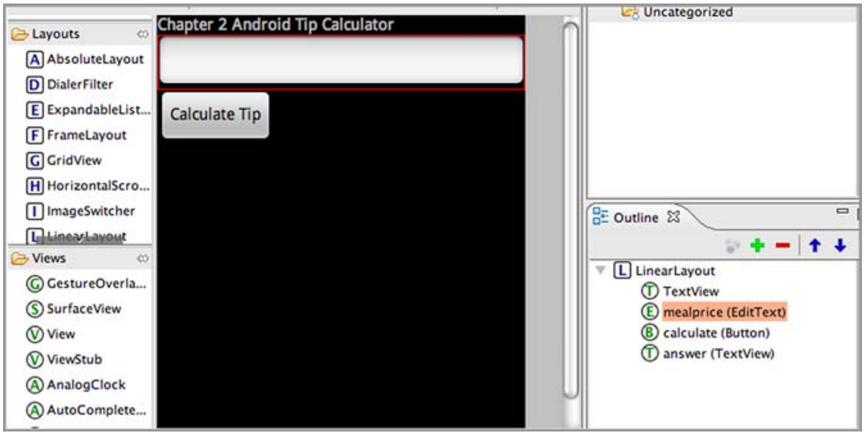


Figure 2.11 Using the GUI Layout tool provided in the ADT to define the user interface elements of your application

It's time to turn our attention to the file ChapterTwo.java to implement the tip calculator functionality. ChapterTwo.java is shown in the following listing. We've omitted some imports for brevity. You can download the complete source code from the Manning website at http://manning.com/ableson2.

Listing 2.2 ChapterTwo.java implements the tip calculator logic

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

1 Reference EditText for mealprice

2 Log entry

3 Get meal price

```
                   if (mealprice.indexOf("$") == -1) {
                      mealprice = "$" + mealprice;
                   }
                   float fmp = 0.0F;
                   // get currency formatter
                   NumberFormat nf =
                   java.text.NumberFormat.getCurrencyInstance();
                   // grab the input meal price
                   fmp = nf.parse(mealprice).floatValue();
                   // let's give a nice tip -> 20%
                   fmp *= 1.2;
                   Log.i(tag,"Total Meal Price (unformatted) is ["
+ fmp + "]");
                   // format our result
                   answer = "Full Price, Including 20% Tip: "
+ nf.format(fmp);
                   answerfield.setText(answer);                 4 Display full price, including tip
                   Log.i(tag,"onClick complete.");
                } catch (java.text.ParseException pe) {         5 Catch parse error
                  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());
                }
            }
        });
    }
}
```

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

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

Though a number of imports are necessary to resolve class names in use, most of the import statements have been omitted from listing 2.2 for the sake of brevity. One import that's shown contains the definition for the `java.text.NumberFormat` class, which is used to format and parse currency values.

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

Now let's go through the code in listing 2.2. We connect the UI element containing `mealprice` to a class-level variable of type `EditText` ❶ by calling the `findViewById` method and passing in the identifier for the `mealprice`, as defined by the automatically generated `R` class, found in R.java. With this reference, we can access the user's input and manipulate the meal price data as entered by the user. Similarly, we connect the UI element for displaying the calculated answer back to the user, again by calling the `findViewById` method.

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

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

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

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

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

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

> **TIP** Using the command-line tools found in the Android SDK, you can create batch builds of your applications without using the IDE. This approach is useful for software shops with a specific configuration-management function and a desire to conduct automated builds. In addition to the Android-specific build tools found under the tools subdirectory of your Android SDK

installation, you'll also need JDK version 5.0 or later to complete command-line application builds. Creating sophisticated automated builds of Android applications is beyond the scope of this book, but you can learn more about the topic of build scripts by reading *Ant in Action: Second Edition of Java Development with Ant,* by Steve Loughran and Erik Hatcher, found at http://www.manning.com/loughran/.

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

2.3.3 *Packaging the application*

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

Recall that despite the compile-time reliance on Java, Android applications don't run in a Java VM. Instead, the Android SDK employs the Dalvik VM. For this reason, Java bytecodes created by the Eclipse compiler must be converted to the .dex file format for use in the Android runtime. The Android SDK has tools to perform these steps, but thankfully the ADT takes care of all of this for you transparently.

The Android SDK contains tools that convert the project files into a file ready to run on the Android emulator. Figure 2.12 depicts the generalized flow of source files in the Android build process. If you recall from our earlier discussion of Android SDK tools, the tool used at design time is `aapt`. Application resource XML files are processed by `aapt`, with the R.java file created as a result—remember that you need to refer to the `R` class for UI identifiers when you connect your code to the UI. Java source

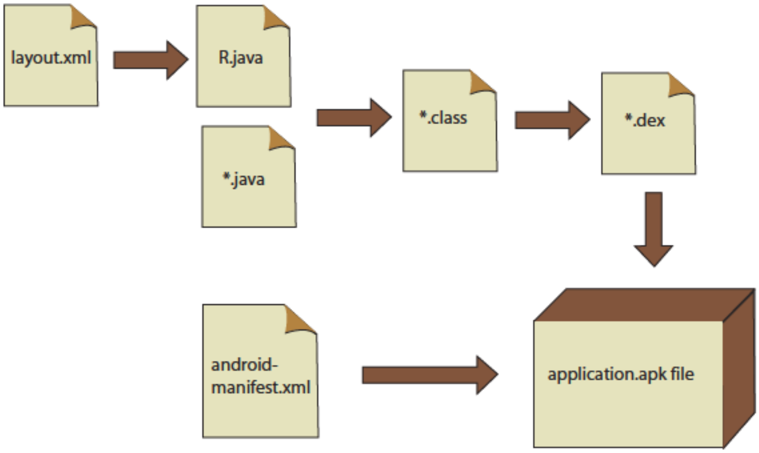
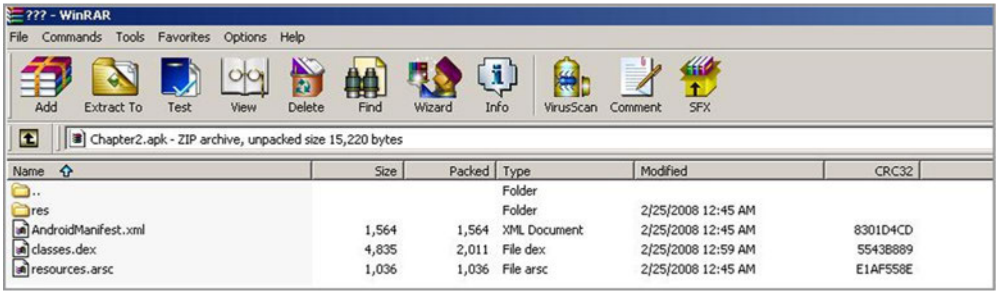


Figure 2.12 **The ADT employs tools from the Android SDK to convert source files to a package that's ready to run on an Android device or emulator.**



Chapter2.apk - ZIP archive, unpacked size 15,220 bytes

Name	Size	Packed	Type	Modified	CRC32
..			Folder		
res			Folder	2/25/2008 12:45 AM	
AndroidManifest.xml	1,564	1,564	XML Document	2/25/2008 12:45 AM	8301D4CD
classes.dex	4,835	2,011	File dex	2/25/2008 12:59 AM	5543B889
resources.arsc	1,036	1,036	File arsc	2/25/2008 12:45 AM	E1AF558E

Figure 2.13 The Android application file format is pzip compatible.

files are first compiled to class files by your Java environment, typically Eclipse and the JDT. After they're compiled, they're then converted to dex files to be ready for use with Android's Dalvik VM. Surprisingly, the project's XML files are converted to a binary representation, not to text as you might expect. But the files retain their .xml extension on the device.

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

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

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

2.4 Using the Android emulator

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

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

2.4.1 Setting up the emulated environment

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

Emulator vs. simulator

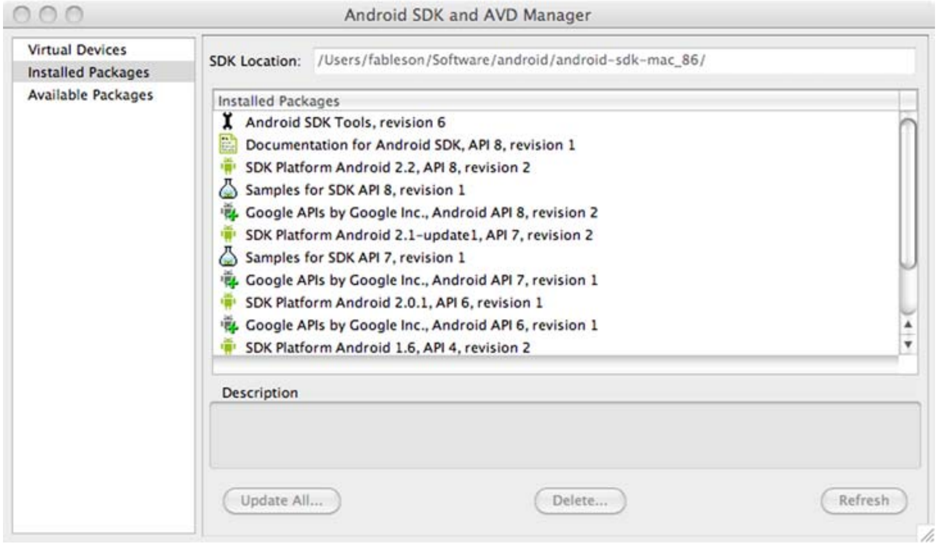
You might hear the words *emulator* and *simulator* thrown about interchangeably. Although they have a similar purpose—testing applications without the requirement of real hardware—those words should be used with care.

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

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

MANAGING AVDS

Starting with version 1.6 of the Android SDK, developers have a greater degree of control over the emulated Android environment than in previous releases. The SDK and AVD Manager permit developers to download the specific platforms of interest. For example, you might be targeting devices running version 1.5 and 2.2 of the Android platform, but you might want to add to that list as new versions become available. Figure 2.14 shows the SDK and AVD Manager with a few packages installed.



Android SDK and AVD Manager

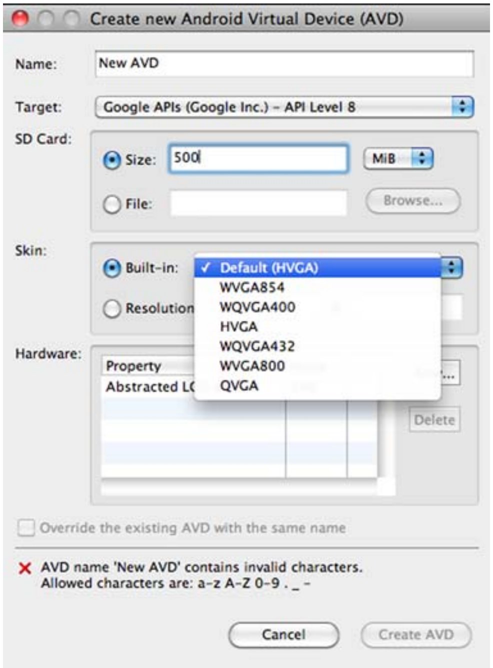
SDK Location: /Users/fableson/Software/android/android-sdk-mac_86/

Installed Packages

- Android SDK Tools, revision 6
- Documentation for Android SDK, API 8, revision 1
- SDK Platform Android 2.2, API 8, revision 2
- Samples for SDK API 8, revision 1
- Google APIs by Google Inc., Android API 8, revision 2
- SDK Platform Android 2.1-update1, API 7, revision 2
- Samples for SDK API 7, revision 1
- Google APIs by Google Inc., Android API 7, revision 1
- SDK Platform Android 2.0.1, API 6, revision 1
- Google APIs by Google Inc., Android API 6, revision 1
- SDK Platform Android 1.6, API 4, revision 2

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

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



Create new Android Virtual Device (AVD)

Name: New AVD

Target: Google APIs (Google Inc.) - API Level 8

SD Card: ● Size: 500 MiB ○ File:

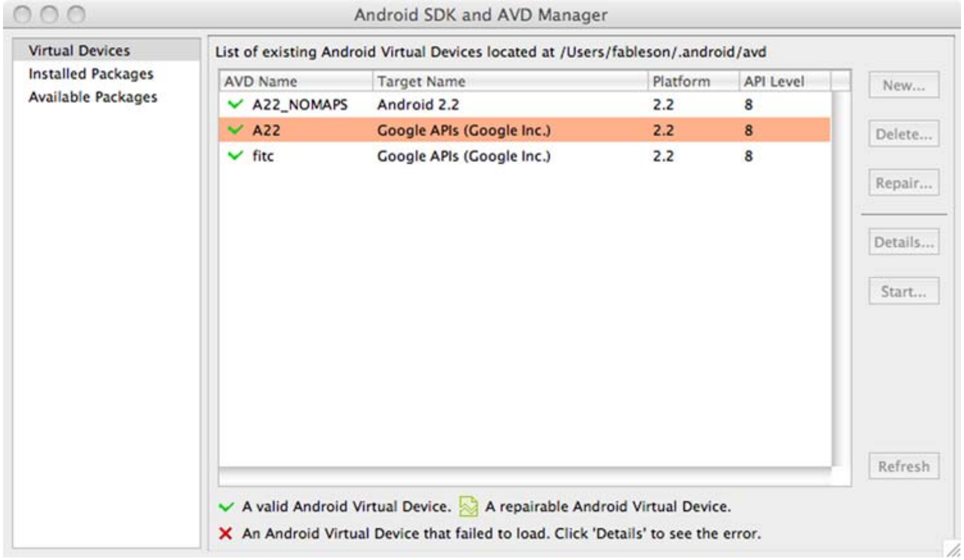
Skin: ● Built-in: ✓ Default (HVGA) ○ Resolution

Hardware: Property Abstracted LC

□ Override the existing AVD with the same name

✕ AVD name 'New AVD' contains invalid characters.
Allowed characters are: a-z A-Z 0-9 . _ -

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



List of existing Android Virtual Devices located at /Users/fableson/.android/avd

AVD Name	Target Name	Platform	API Level
✓ A22_NOMAPS	Android 2.2	2.2	8
✓ A22	Google APIs (Google Inc.)	2.2	8
✓ fitc	Google APIs (Google Inc.)	2.2	8

✓ A valid Android Virtual Device. A repairable Android Virtual Device.

✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

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

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

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

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

SETTING UP EMULATOR RUN CONFIGURATIONS

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

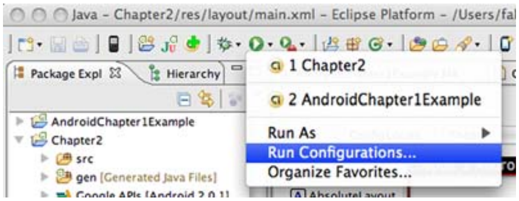
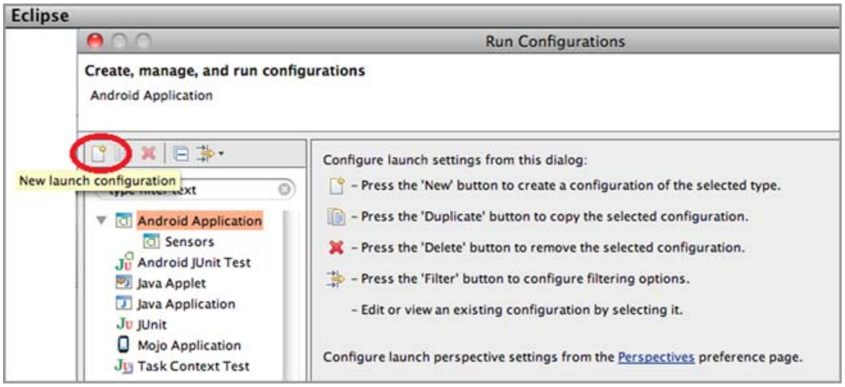


Figure 2.17 Creating a new launch configuration for testing your Android application



Create, manage, and run configurations

Android Application

Configure launch settings from this dialog:

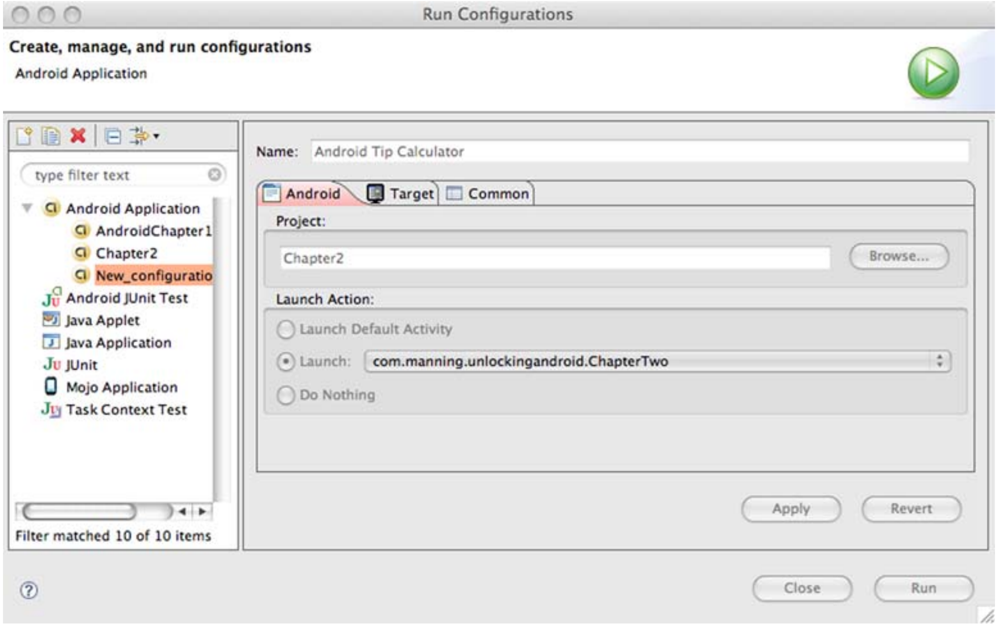
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

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

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

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



Create, manage, and run configurations

Android Application

Name: Android Tip Calculator

Project: Chapter2

Launch Action:

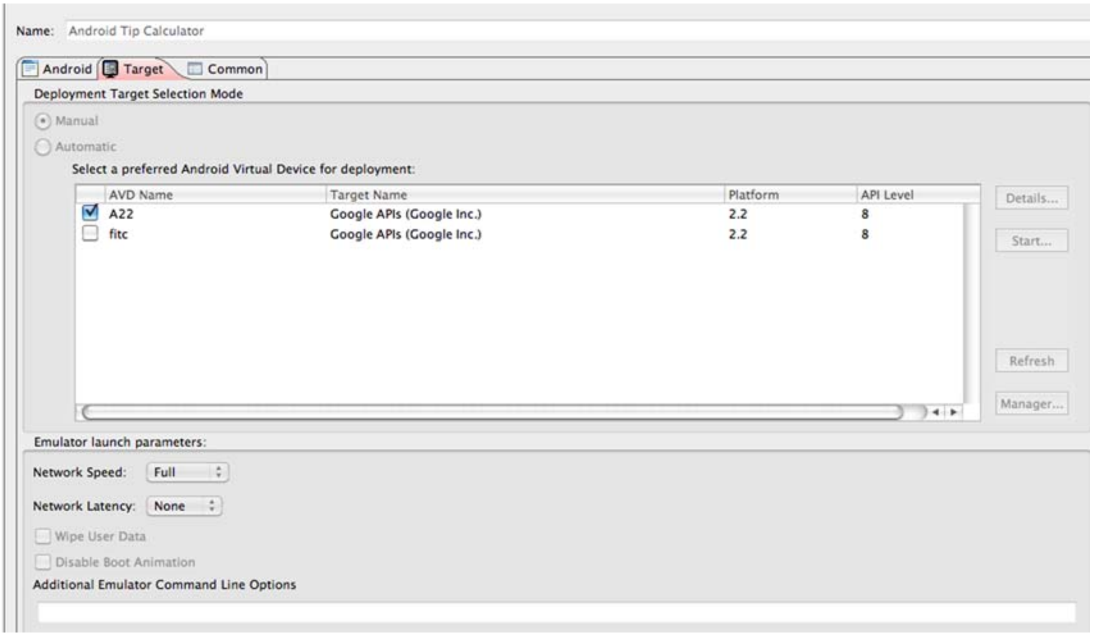
○ Launch Default Activity

● Launch: com.manning.unlockingandroid.ChapterTwo

○ Do Nothing

Filter matched 10 of 10 items

Figure 2.19 **Setting up the Android emulator launch configuration**



Name: Android Tip Calculator

Deployment Target Selection Mode

● Manual

○ Automatic

Select a preferred Android Virtual Device for deployment:

	AVD Name	Target Name	Platform	API Level
☑	A22	Google APIs (Google Inc.)	2.2	8
□	fitc	Google APIs (Google Inc.)	2.2	8

Emulator launch parameters:

Network Speed: Full

Network Latency: None

□ Wipe User Data

□ Disable Boot Animation

Additional Emulator Command Line Options

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

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

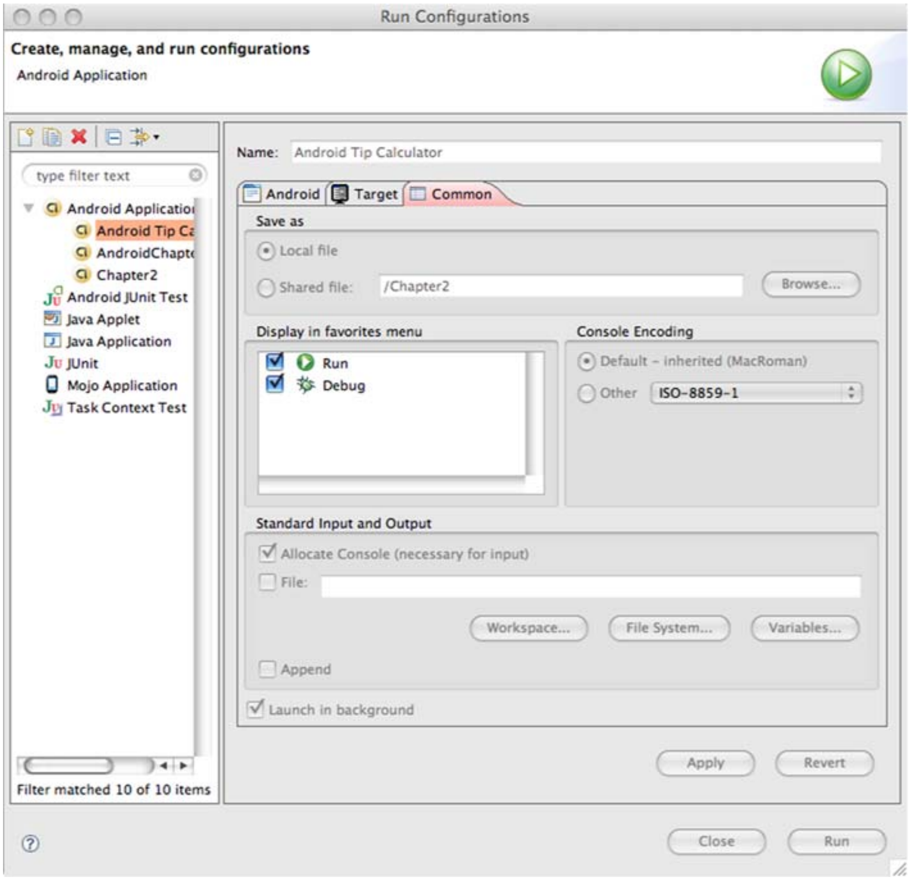
Use the third tab to put this configuration on the Favorites menu in the Eclipse IDE for easy access, as shown in figure 2.21. You can select Run, Debug, or both. Let's choose both for this example, because it makes for easier launching when you want to test or debug the application.

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

2.4.2 Testing your application in the emulator

Now you're finally ready to start the Android emulator to test your tip calculator application. Select the new launch configuration from the Favorites menu, as shown in figure 2.22.

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



Name: Android Tip Calculator

Save as
● Local file
○ Shared file: /Chapter2

Display in favorites menu
☑ Run
☑ Debug

Console Encoding
● Default – inherited (MacRoman)
○ Other ISO-8859-1

Standard Input and Output
☑ Allocate Console (necessary for input)
□ File:
□ Append

☑ Launch in background

Figure 2.21 **Adding the run configuration to the toolbar menu**

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

2.5 Debugging your application

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

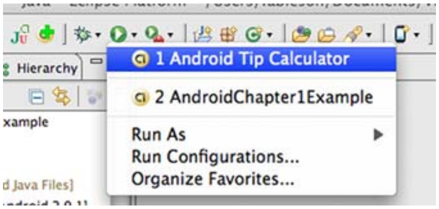
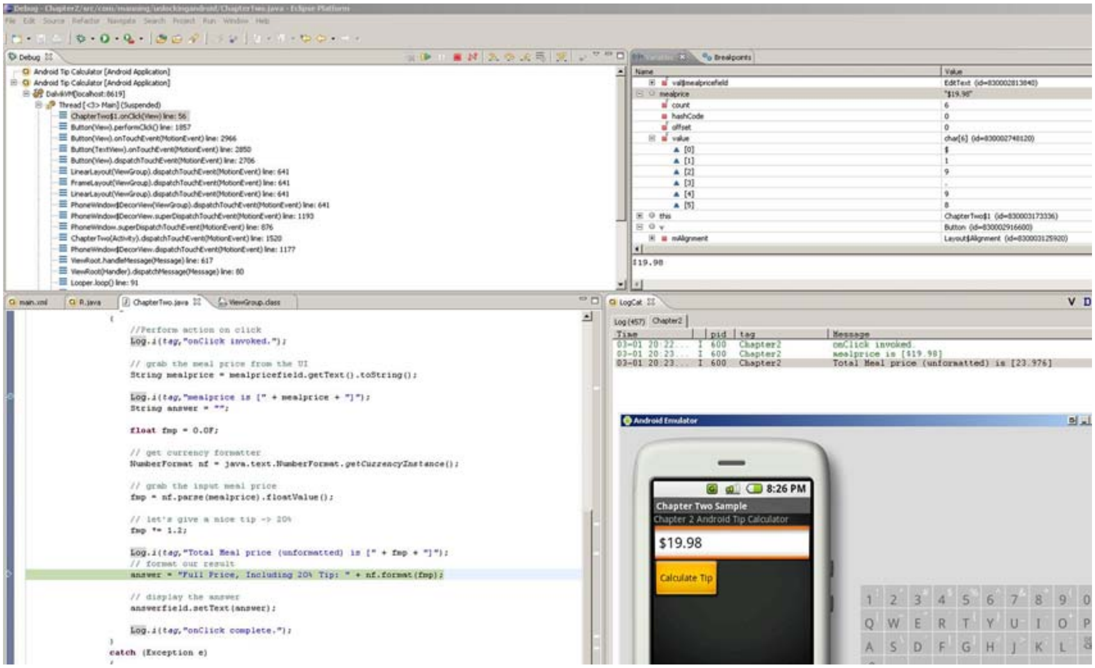


Figure 2.22 **Starting this chapter's sample application, an Android tip calculator.**



```
Android Tip Calculator [Android Application]
Android Tip Calculator [Android Application]
  DalvikVM[localhost:8619]
    Thread [<3> Main] (Suspended)
      ChapterTwo$1.onClick(View) line: 56
      Button(View).performClick() line: 1857
      Button(View).onTouchEvent(MotionEvent) line: 2966
      Button(TextView).onTouchEvent(MotionEvent) line: 2850
      Button(View).dispatchTouchEvent(MotionEvent) line: 2706
      LinearLayout(ViewGroup).dispatchTouchEvent(MotionEvent) line: 641
      FrameLayout(ViewGroup).dispatchTouchEvent(MotionEvent) line: 641
      LinearLayout(ViewGroup).dispatchTouchEvent(MotionEvent) line: 641
      PhoneWindow$DecorView(ViewGroup).dispatchTouchEvent(MotionEvent) line: 641
      PhoneWindow$DecorView.superDispatchTouchEvent(MotionEvent) line: 1193
      PhoneWindow.superDispatchTouchEvent(MotionEvent) line: 876
      ChapterTwo(Activity).dispatchTouchEvent(MotionEvent) line: 1520
      PhoneWindow$DecorView.dispatchTouchEvent(MotionEvent) line: 1177
      ViewRoot.handleMessage(Message) line: 617
      ViewRoot(Handler).dispatchMessage(Message) line: 80
      Looper.loop() line: 91
```

Name	Value
val$mealpricefield	EditText (id=830002813840)
mealprice	"$19.98"
count	6
hashCode	0
offset	0
value	char[6] (id=830002748120)
[0]	$
[1]	1
[2]	9
[3]	.
[4]	9
[5]	8
this	ChapterTwo$1 (id=830003173336)
v	Button (id=830002916600)
mAlignment	Layout$Alignment (id=830003125920)

```
$19.98
```

```
{
    //Perform action on click
    Log.i(tag,"onClick invoked.");

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

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

    float fmp = 0.0F;

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

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

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

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

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

    Log.i(tag,"onClick complete.");
}
catch (Exception e)
{
```

```
Time            pid  tag       Message
03-01 20:22... I 600  Chapter2  onClick invoked.
03-01 20:23... I 600  Chapter2  mealprice is [$19.98]
03-01 20:23... I 600  Chapter2  Total Meal price (unformatted) is [23.976]
```

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

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

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

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

2.6 Summary

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

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

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

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

Part 2

Exercising the Android SDK

The Android SDK provides a rich set of functionality enabling developers to create a wide range of applications. In part 2 we systematically examine the major portions of the Android SDK, including practical examples in each chapter. We start off with a look at the application lifecycle and user interfaces (chapter 3), graduating to `Intents` and `Services` (chapter 4). No platform discussion is complete without a thorough examination of the available persistence and stor-age methods (chapter 5) and in today's connected world, we cannot overlook core networking and web services skills (chapter 6). Because the Android platform is a telephone, among other things, we take a look at the telephony capabilities of the platform (chapter 7). Next we move on to notifications and alarms (chapter 8). Android graphics and animation are covered (chapter 9) as well as multimedia (chapter 10). Part 2 concludes with a look at the location-based services available to the Android developer (chapter 11).

3 User interfaces

This chapter covers

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

With our introductory tour of the main components of the Android platform and development environment complete, it's time to look more closely at the fundamental Android concepts surrounding activities, views, and resources. Activities are essential because, as you learned in chapter 1, they make up the screens of your application and play a key role in the Android application lifecycle. Rather than allowing any one application to wrest control of the device away from the user and from other applications, Android introduces a well-defined lifecycle to manage processes as needed. It's 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 subcomponents called *views*.

Views are what your users see and interact with. Views handle layout, provide text elements for labels and feedback, provide buttons and forms for user input, and draw graphics to the device 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're the conductor, an `Activity` is your symphony, and `View` objects are your musicians.

Musicians need instruments, so we'll stretch this analogy further to bring Android resources into the mix. Views and other Android components use strings, colors, styles, and graphics, which are compiled into a binary form and made available to applications as resources. The automatically generated `R.java` class, which we introduced in chapter 1, provides a reference to individual resources and is the bridge between binary references and the source code of an Android application. You use the `R` class, for example, to grab a string or a color and add it to a `View`. The relationship between activities, views, and resources is depicted in figure 3.1.

Along with the components you use to build an application—views, resources, and activities—Android includes the manifest file we introduced you to in chapter 1, AndroidManifest.xml. This XML file describes where your application begins, what its permissions are, and what activities (and services and receivers, which you'll see in the next two chapters) it includes. Because this file is central to every Android application, we're going to address it in more detail in this chapter, and we'll come back to it frequently in later parts of the book. The manifest file is the one-stop shop for the platform to start and manage your application.

If you've done any development involving UIs of any kind on any platform, the concepts of activities, views, and resources might 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.

Next, we're going to introduce the sample application that we use to walk through these concepts, moving beyond theory and into the code to construct an `Activity`.

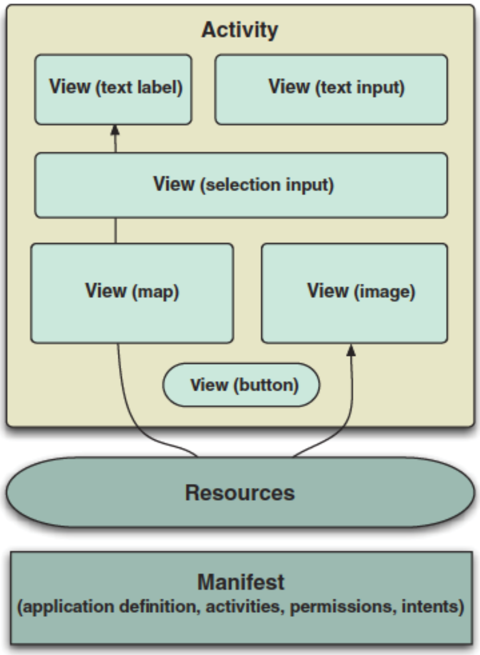


Figure 3.1 High-level diagram of activity, view, resources, and manifest relationship, showing that activities are made up of views, and views use resources.

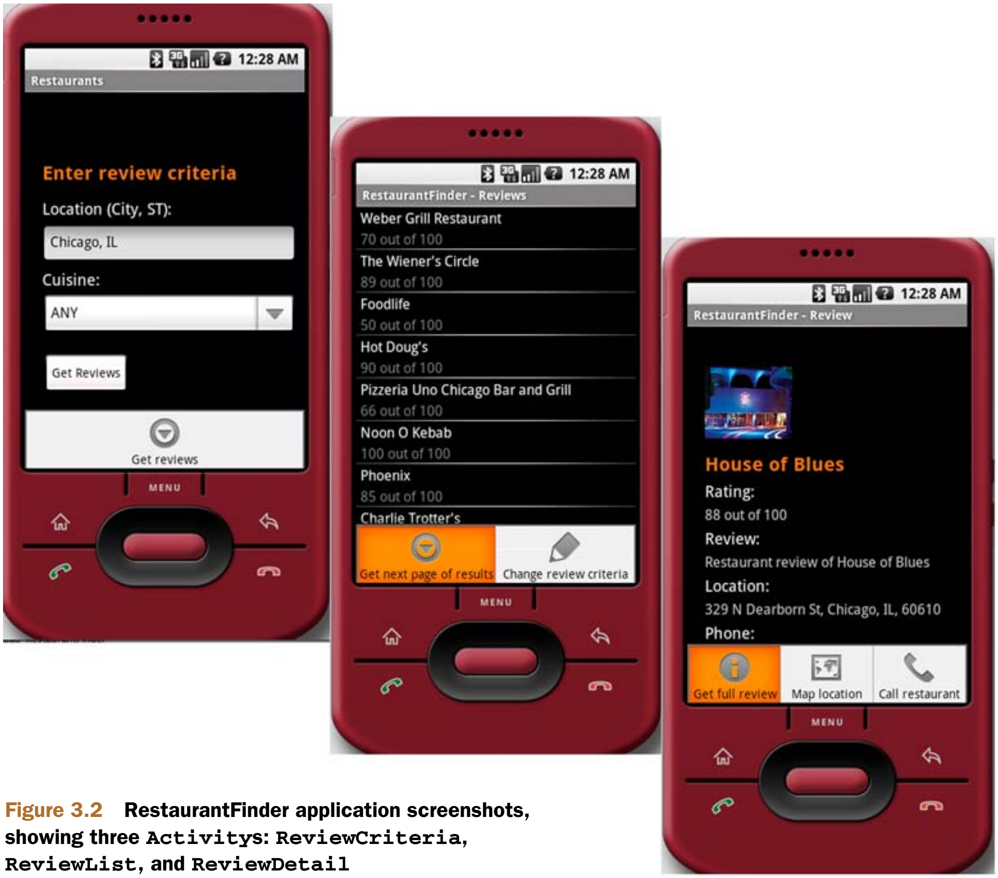
3.1 Creating the Activity

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

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

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

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



Restaurants

Enter review criteria

Location (City, ST): Chicago, IL

Cuisine: ANY

RestaurantFinder - Reviews

Weber Grill Restaurant
70 out of 100

The Wiener's Circle
89 out of 100

Foodlife
50 out of 100

Hot Doug's
90 out of 100

Pizzeria Uno Chicago Bar and Grill
66 out of 100

Noon O Kebab
100 out of 100

Phoenix
85 out of 100

Charlie Trotter's

RestaurantFinder - Review

House of Blues

Rating:
88 out of 100

Review:
Restaurant review of House of Blues

Location:
329 N Dearborn St, Chicago, IL, 60610

Phone:

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

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

3.1.1 Creating an Activity class

To create a screen, extend the `android.app.Activity` base class (as you did in chapter 1) and override the key methods it defines. Listing 3.1 shows the first portion of the RestaurantFinder's `ReviewCriteria` class.

Listing 3.1 The first half of the `ReviewCriteria` `Activity` class

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

1 Override onCreate()

2 Define layout with setContentView

3 Inflate views from XML

4 Define ArrayAdapter instance

5 Set view for dropdown

The `ReviewCriteria` class extends `android.app.Activity`, which does a number of important things: it gives your application a context, because `Activity` itself extends the `android.app.ApplicationContext` class; it brings the Android lifecycle methods into play; it gives the framework a hook to start and run your application; and it provides a container into which `View` elements can be placed.

Because an `Activity` represents an interaction with the user, it needs to provide components on the screen—this is where views come into play. In our `ReviewCriteria` class, we reference three views in the code: `cuisine`, `grabReviews`, and `location` 2. `cuisine` is a fancy select list component, known in Android terms as a

Location as an EditText View

Why are we 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? After all, we could ask the user to select the current location using a Map, rather than requiring the user to type in an address. Good eye, but we're doing this intentionally—we want this early example to be complete and nontrivial, but not too complicated. You'll learn more about using the location support Android provides and MapViews in later chapters.

Spinner. grabReviews is a Button. location is a type of View known as an EditText, a basic text-entry component.

You place View elements like these within an Activity using a particular layout to define the elements of a screen. You can define layout and views directly in code or in a layout XML resource file.

You'll learn more about views as we progress through this section, and we focus specifically on the topic of layouts in section 3.2.5.

After an Activity is started, the Android application lifecycle rules take over and the onCreate() method is invoked ❶. This method is one of a series of important lifecycle methods the Activity class provides. Every Activity overrides onCreate(), where component initialization steps are invoked. Not every Activity will need to override the other available lifecycle methods. The Activity lifecycle is worthy of an in-depth discussion of its own; for that reason we'll explore these methods further in section 3.1.2.

Inside the onCreate() method, the setContentView() method is where you'll typically associate an XML layout file ❷. We say *typically* because you don't have to use an XML file at all; instead, you can define all your layout and View configuration directly in code, as Java objects. This technique is used in applications where a dynamic GUI is required. Generally speaking, it's often easier (and better practice) to use an XML layout resource for each Activity. An XML layout file defines View objects, organized into a hierarchical tree structure. After they're defined in relation to the parent layout, each view can then be inflated at runtime.

Layout and view details, defined in XML or in code, are also topics we'll address 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 runtime manipulation, such as binding to data, can then be referenced in code and cast to their respective subtypes ❸. Views that are static—those you don't need to interact with or update at runtime, such as labels—don't need to be referenced in code at all. These views automatically show up on the screen because they're part of the layout as defined in the XML. They don't need any explicit setup steps in code.

Going back to the screenshots in figure 3.1, note that the ReviewCriteria screen has two labels as well as the three inputs we've already discussed. These labels aren't present in the code; they're simply defined in the review_criteria.xml file that's associated with this Activity. You'll see this layout file when we discuss XML-defined resources.

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

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

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

Listing 3.2 The second half of the ReviewCriteria Activity class

```
. . .
@Override
public boolean onCreateOptionsMenu(Menu menu) {
     super.onCreateOptionsMenu(menu);
     menu.add(0, ReviewCriteria.MENU_GET_REVIEWS, 0,
         R.string.menu_get_reviews).setIcon(
         android.R.drawable.ic_menu_more);
     return true;
 }
@Override
public boolean onMenuItemSelected(int featureId, MenuItem item) {
     switch (item.getItemId()) {
         case MENU_GET_REVIEWS:                         Respond when
             handleGetReviews();                      ❶ menu item selected
             return true;
     }
     return super.onMenuItemSelected(featureId, item);
 }
 private void handleGetReviews() {                        Define method to
     if (!validate()) {                                 ❷ process reviews
         return;
     }
```

```
   RestaurantFinderApplication application =
       (RestaurantFinderApplication)
         getApplication();                                   ❸ Use Application
   application.setReviewCriteriaCuisine(                        object for state
       this.cuisine.getSelectedItem().toString());
   application.setReviewCriteriaLocation(
       this.location.getText().toString());
   Intent intent =
     new
   Intent(Constants.INTENT_ACTION_VIEW_LIST);               ❹ Create Intent
   startActivity(intent);
 }
 private boolean validate() {
   boolean valid = true;
   StringBuilder validationText = new StringBuilder();
   if ((this.location.getText() == null) ||
         this.location.getText().toString().equals("")) {
       validationText.append(getResources().getString(
         R.string.location_not_supplied_message));
       valid = false;
   }
   if (!valid) {
       new AlertDialog.Builder(this).
       setTitle(getResources().getString(R.string.alert_label)).
       setMessage(validationText.toString()).               Use AlertDialog ❺
       SetPositiveButton("Continue",
         new android.content.DialogInterface.
           OnClickListener() {
               public void onClick(
                 DialogInterface dialog, int arg1) {
               }
           }).show();
       validationText = null;
   }
   return valid;
 }
}
```

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

In addition to creating the menu item, we need to perform an action when the MenuItem is selected. We do this in the onMenuItemSelected() event method ❶, where we parse the ID of the multiple possible menu items with a switch statement. When the MENU_GET_REVIEWS item is selected, we invoke the handleGetReviews method ❷. This method stores the user's selection state in the Application object ❸ and prepares to call the next screen. We've moved this logic into its own method

Using the Menu vs. onscreen buttons

We've chosen to use the Menu here, in addition to the onscreen buttons. 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 tapping a selection (button and a tap) is appropriate for what you're doing, or whether an onscreen button (single tap) is more appropriate. Generally, onscreen buttons should be tied to UI elements, such as a search button for a search form input, and menu items should be used for more broad actions such as submitting a form, or performing an action such as creating, saving, editing, or deleting. Because all rules need an exception, if you have the screen real estate, it might be more convenient for users to have onscreen buttons for actions as well, as we've done in the `ReviewCriteria Activity`. The most important thing to keep in mind with these types of UI decisions is to be consistent. If you do it one way on one screen, use that same approach on other screens.

because we're using it from multiple places—both from our onscreen `Button` and our `MenuItem`.

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

After we store the criteria state, we fire off an action in the form of an Android `Intent` ❹. We touched on `Intents` in chapter 1, and we'll delve into them further in the next chapter, but basically we're asking another `Activity` to respond to the user's selection of a menu item by calling `startActivity(intent)`. An alternative way to start an `Activity` is with the `startActivityForResult` method, which we'll introduce later in this book.

Also notable in the `ReviewCriteria` example is that we're using an `AlertDialog` ❺. Before we allow the next `Activity` to be invoked, we call a simple `validate()` method that we've created, where we display a pop-up alert dialog to the user if the location hasn't been properly specified. Along with generally demonstrating the use of `AlertDialog`, this demonstrates how a button can be made to respond to a click event with an `OnClickListener()`.

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

The Builder pattern

You might have noticed the use of the Builder pattern, where we add parameters to the `AlertDialog` we created in the `ReviewCriteria` class. If you aren't familiar with this approach, each of the methods invoked, such as `AlertDialog.setMessage()` and `AlertDialog.setTitle()`, returns a reference to itself (`this`), which means we can continue chaining method calls. This approach avoids either an extra-long constructor with many parameters or repeating the class reference throughout the code. `Intent`s also use this handy pattern; it's something you'll see time and time again in Android.

3.1.2 *Exploring the Activity lifecycle*

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

HOW PROCESSES AND ACTIVITIES RELATE

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

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`) is known as an *empty* process and is last in line.

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

```
adb shell dumpsys activity
```

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

All `Activity` classes have to be able to handle being stopped and shut down at any time. Remember, a user can and will change directions at will. It might be a phone call or an incoming SMS message, but the user will bounce around from one application to the next. If the process your `Activity` is in falls out of the foreground, it's eligible to be killed and it's not up to you; it's up to the platform's algorithm, based on available resources and relative priorities.

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

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

ACTIVITY LIFECYCLE

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

Each of the lifecycle methods Android provides has a distinct purpose, and each happens during part of the foreground, visible, or entire lifecycle phase:

- In the *foreground phase*, the Activity is viewable on the screen and is on top of everything else (when the user is interacting with the Activity to perform a task).
- In the *visible phase*, the Activity is on the screen, but it might not be on top and interacting with the user (when a dialog or floating window is on top of the Activity, for example).

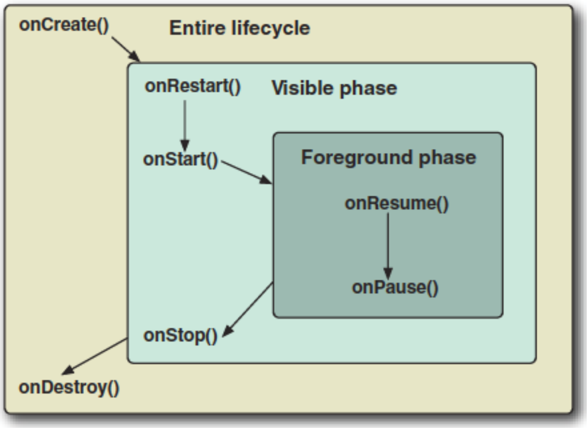


Figure 3.3 Android Activity lifecycle diagram, showing the methods involved in the foreground and visible phases

- The *entire lifecycle phase* refers to the methods that might be called when the application isn't on the screen, before it's created, and after it's gone (prior to being shut down).

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

Table 3.1 Android `Activity` main lifecycle methods and their purpose

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

Beyond the main high-level lifecycle methods outlined in table 3.1, additional, finer-grained methods are available. You don't typically need methods such as `onPostCreate` and `onPostResume`, so we won't go into detail about them, but be aware that they exist if you need that level of control. See the `Activity` documentation for full method details.

As for the main lifecycle methods that you'll use the majority of the time, it's important to know that `onPause()` is the last opportunity you have to clean up and save state information. The processes that host your `Activity` classes won't be killed by the platform until after the `onPause()` method has completed, but they might be killed thereafter. The system will attempt to run through all of the lifecycle methods every time, but if resources are spiraling out of control, as determined by the platform, a fire alarm might be sounded and the processes that are hosting activities that are beyond the `onPause()` method might be killed *at any point*. Any time your `Activity` is moved to the background, `onPause()` is called. Before your `Activity` is completely removed, `onDestroy()` is called, though it might not be invoked in all circumstances.

The `onPause()` method is definitely where you need to save persistent state. Whether that persistent state is specific to your application, such as user preferences, or globally 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'll discuss how to save data in chapter 5, but here the important thing is to know when and where that needs to happen.

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

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

Now that you have some background about the Activity lifecycle and you've created your first screen, let's investigate views and fill in some more detail.

3.2 Working with views

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

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

In this section, we'll discuss the fundamental aspects of views: the common views that Android provides, custom views that you can create as you need them, layout in relation to views, and event handling. We won't address views defined in XML here, because that's covered in section 3.3 as part of a larger resources discussion. We'll begin with the common View elements Android provides by taking a short tour of the API.

3.2.1 Exploring common views

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

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

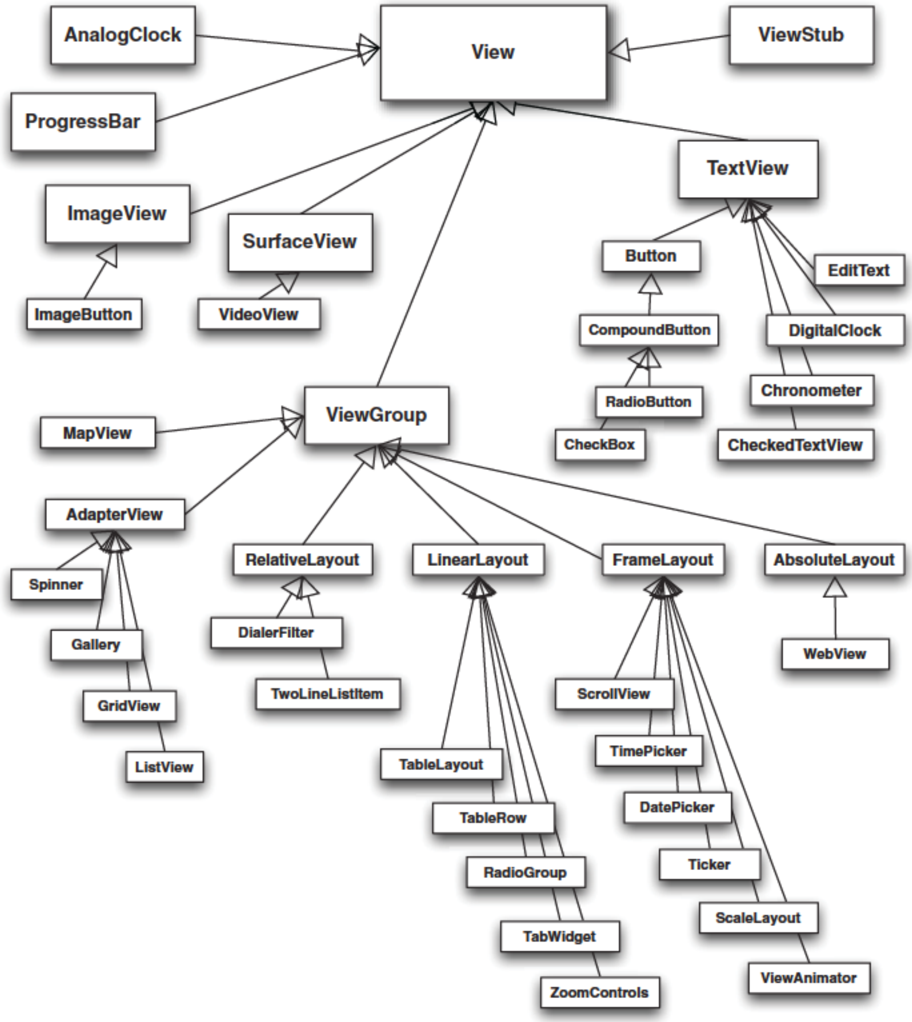


Figure 3.4 **A class diagram of the Android View API, showing the root View class and specializations from there; note that ViewGroup classes, such as layouts, are also a type of View.**

As is evident from the diagram in figure 3.4, which isn't an exhaustive representation, the View is the base class for many classes. ViewGroup is a special subclass of View related to layout, as are other elements such as the commonly used TextView. All UI classes are derived from the View class, including the layout classes (which extend ViewGroup).

Of course, everything that extends View has access to the base class methods. These methods allow you to perform important UI-related operations, such as setting the background, the minimum height and width, padding, layout parameters, and event-related attributes. Table 3.2 lists some of the methods available in the root View class. Beyond the base class, each View subclass typically adds a host of refined methods to further manipulate its respective state, such as what's shown for TextView in table 3.3.

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

Each View element you use has its own path through the API, which means that a particular set of methods is available; for details on all the methods, see the Android Javadocs at http://code.google.com/android/reference/android/view/View.html.

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

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

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

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

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 this initial impression, many of the concepts involved quickly become evident, and their use becomes more intuitive as you move from `View` to `View`, because they're ultimately just specializations on the same base class. When you get familiar with working with `View` classes, learning to use a new `View` becomes intuitive and natural.

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

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

3.2.2 *Using a ListView*

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

This `Activity` uses a `ListView` component to display a list of review data that's obtained from calling the Google Base Atom API using HTTP. We refer to this API generically as a web service, even though it's not technically SOAP or any other standard associated with the web service moniker. We make the HTTP call by appending the user's criteria to the required Google Base URL. We then parse the results with the *Simple API for XML (SAX)* and create a `List` of *reviews*. Neither the details of *XML* parsing nor the use of the network itself are of much concern to us here—rather we'll focus on the `Views` employed to represent the data returned from the web service call. The resulting `List` will be used to populate our screen's list of items to choose from.

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

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

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

1 Use ReviewAdapter

2 Use resourced-defined layout

3 Define TextView for empty

4 Use Application for global state

5 Use Intent extra

The `ReviewList` `Activity` extends `ListActivity`, which is used to host a `ListView`. The default layout of a `ListActivity` is a full-screen, centered list of choices for the user to select from. A `ListView` is similar in concept to a `Spinner`; in fact, they're both subclasses of `AdapterView`, as you saw in the class diagram in figure 3.4. `ListView`, like `Spinner`, also uses an `Adapter` to bind to data. In this case, we're using a custom

ReviewAdapter class ❶. You'll learn more about ReviewAdapter in the next section, when we discuss custom views. The important part here is that we're using a custom Adapter for our ListView, and we use a List of Review objects to populate the Adapter.

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

After we declare our ListView and its data, we move on to the typical onCreate() tasks you've already seen, including using a layout defined in an XML file ❷. This is significant with respect to ListActivity because a ListView with the ID name list is required if you want to customize the layout, as we've done. Note that the ID is defined in the layout XML file; we'll cover that in section 3.3.3. If you don't provide a layout, you can still use ListActivity and ListView, but you just get the system default configuration. We're also defining a UI element that's used to display the message No Data in the event that our List of reviews is empty ❸. We're setting several specific properties on the ListView, using its customization methods: whether the list items themselves are focusable, how many elements can be selected at a time, and the View to use when the list is empty.

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

Listing 3.4 The second half of the ReviewList Activity class

```
     . . .
    @Override
    public boolean onMenuItemSelected
(int featureId, MenuItem item) {
        Intent intent = null;

        switch (item.getItemId()) {
            case MENU_GET_NEXT_PAGE:
                intent = new Intent(Constants.INTENT_ACTION_VIEW_LIST);
                intent.putExtra(Constants.STARTFROM_EXTRA,
                    getIntent().getIntExtra(Constants.STARTFROM_EXTRA, 1)
                    + ReviewList.NUM_RESULTS_PER_PAGE);
                startActivity(intent);
                return true;
            case MENU_CHANGE_CRITERIA:
                intent = new Intent(this, ReviewCriteria.class);
```

❶ **Increment startFrom Intent extra**

```
                startActivity(intent);
                return true;
            }
            return super.onMenuItemSelected(featureId, item);
        }
        @Override
        protected void onListItemClick(ListView l, View v,
            int position, long id) {
            RestaurantFinderApplication application =
                (RestaurantFinderApplication) getApplication();
            application.setCurrentReview(this.reviews.get(position));
            Intent intent = new Intent(Constants.INTENT_ACTION_VIEW_DETAIL);
            intent.putExtra(Constants.STARTFROM_EXTRA, getIntent().getIntExtra(
                Constants.STARTFROM_EXTRA, 1));
            startActivity(intent);
        }
        private void loadReviews(String location, String cuisine,
            int startFrom) {
            final ReviewFetcher rf = new ReviewFetcher(location,
            cuisine, "ALL", startFrom,
                ReviewList.NUM_RESULTS_PER_PAGE);
            this.progressDialog =
                ProgressDialog.show(this, " Working...",
                    " Retrieving reviews", true, false);
            new Thread() {
                public void run() {
                    reviews = rf.getReviews();
                    handler.sendEmptyMessage(0);
                }
            }.start();
        }
    }
```

Get Application object and set state ❷

Create loadReviews method ❸

❹ **Show ProgressDialog**

Make web service call ❺

This Activity has a menu item that 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. When the MENU_GET_NEXT_PAGE menu item is selected, we define a new Intent to reload the screen with an incremented startFrom value, with some assistance from the Intent class's getExtras() and putExtras() methods ❶.

After the menu-related methods, you see a method named onListItemClick(). This method is invoked when one of the list items in a ListView is clicked. We use the ordinal position of the clicked item to reference the particular Review item the user selected, and we set this into the Application for later use in the Review-Detail Activity (which you'll begin to implement in section 3.3) ❷. After we have the data set, we then call the next Activity, including the startFrom extra.

In the ReviewList class, we have the loadReviews() method ❸. This method is significant for several reasons. First, it sets up the ReviewFetcher class instance, which is used to call out to the Google Base API over the network and return a List of Review objects. Then it invokes the ProgressDialog.show() method to show the user we're retrieving data ❹. Finally, it sets up a new Thread ❺, within which the ReviewFetcher is used, and the earlier Handler you saw in the first half of ReviewList is sent an empty message. If you refer to listing 3.3, which is when the Handler was established, you can see where, 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 iterates the Adapter and displays a returned View for every item.

With the Activity created and set up 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 aren't view-specific but are worth a small detour at this point, because you'll want to use these classes when you're trying to perform tasks related to retrieving and manipulating data that the UI needs—a common design pattern when you're building Android applications.

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 to schedule tasks using Message and Runnable objects.

The Android platform monitors the responsiveness of applications and kills those that are considered nonresponsive. An *Application Not Responding (ANR) event* occurs when no response is received to a user input for 5 seconds. When a user interacts with your application by touching the screen, pressing a key, or the like, your application must respond. So does this mean that every operation in your code must complete within 5 seconds? No, of course not, but the main UI thread does have to *respond* within that time frame. To keep the main UI thread snappy, any long-running tasks, such as retrieving data over the network, getting a large amount of data from a database, or performing complicated or time-consuming calculations, should be performed in a separate Thread, apart from the main UI Thread.

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

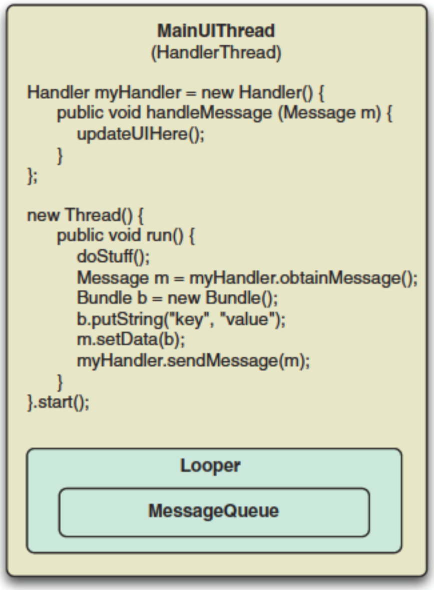
When we used a Handler in listings 3.3 and 3.4, we created a Handler with a no-argument constructor. With this approach, the Handler is automatically associated with the Looper of the currently running thread, typically the main UI thread. The main UI thread, which is created by the process of the running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

When you're implementing a Handler, you'll 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 figure 3.5

demonstrate. Calling sendMessage puts your message on the MessageQueue, which the Looper services.

Along with sending messages into handlers, you can also send Runnable objects directly, and you can schedule things to be run at different times in the future. You *send* messages and you *post* runnables. Each of these concepts supports methods such as sendEmptyMessage(int what), which we've already used, and the counterparts sendEmptyMessageAtTime(int what, long time) and sendEmptyMessageDelayed(int what, long delay). After your Message is in the queue, it's processed either as soon as possible or according to the requested schedule or delay indicated when it was sent or posted.

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



```
Handler myHandler = new Handler() {
    public void handleMessage (Message m) {
        updateUIHere();
    }
};

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

Figure 3.5 Using the Handler class with separate threads, and the relationship between HandlerThread, Looper, and MessageQueue

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

3.2.4 *Creating custom views*

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

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

Listing 3.5 The ReviewAdapter and inner ReviewListView classes

```
public class ReviewAdapter extends BaseAdapter {
    private final Context context;
    private final List<Review> reviews;
    public ReviewAdapter(Context context, List<Review> reviews) {
        this.context = context;
        this.reviews = reviews;
    }
    @Override
    public int getCount() {
        return this.reviews.size();
    }
    @Override
    public Object getItem(int position) {
        return this.reviews.get(position);
    }
    @Override
    public long getItemId(int position) {
        return position;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        Review review = this.reviews.get(position);
        return new ReviewListView(
                   this.context, review.name, review.rating);
    }
    private final class ReviewListView extends LinearLayout {
        private TextView name;
        private TextView rating;
        public ReviewListView(
                   Context context, String name, String rating) {
            super(context);
            setOrientation(LinearLayout.VERTICAL);
            LinearLayout.LayoutParams params =
 new LinearLayout.LayoutParams(
                ViewGroup.LayoutParams.WRAP_CONTENT,
                ViewGroup.LayoutParams.WRAP_CONTENT);
            params.setMargins(5, 3, 5, 0);
            this.name = new TextView(context);
            this.name.setText(name);
            this.name.setTextSize(16f);
            this.name.setTextColor(Color.WHITE);
            this.addView(this.name, params);
            this.rating = new TextView(context);
            this.rating.setText(rating);
            this.rating.setTextSize(16f);
            this.rating.setTextColor(Color.GRAY);
            this.addView(this.rating, params);
        }
    }
}
```

1 Override basic Adapter methods

2 Override Adapter getView

3 Define custom inner View class

4 Set Layout in code

5 Add TextView to tree

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

Our `ReviewAdapter` class accepts two parameters in the constructor and assigns those values to two simple member objects: `Context` and `List<Review>`. This class goes on to implement the straightforward required `Adapter` interface methods that return a count, an item, and an ID—we use the ordinal position in the collection as the ID ❶. The next `Adapter` method we have to implement is the most important—`getView()`. The `Adapter` returns any `View` we create for a particular item in the collection of data that it's supporting. Within this method, we get a particular `Review` object based on the position/ID, then we create an instance of a custom `ReviewListView` object to return as the `View` ❷.

`ReviewListView` itself, which extends `LinearLayout` (something you'll learn more about in section 3.2.4), is an inner class inside `ReviewAdapter`; we never use it except to return a view from `ReviewAdapter` ❸. Within it, you see an example of setting layout and `View` details in code, rather than relying on their definition in XML. In this listing, we set the orientation, parameters, and margin for our layout ❹. Next, we populate the simple `TextView` objects that will be children of our new `View` and represent data. When these are set up via code, we add them to the parent container, which is in this case our custom class `ReviewListView` ❺. This is where the data binding happens—the bridge to the `View` from data. Another important thing to note about this is that we've created not only a custom `View` but also a composite one. We're using simple existing `View` objects in a particular layout to construct a new type of reusable `View`, which shows the detail of a selected `Review` object on screen, as depicted in figure 3.2.

Our custom `ReviewListView` object is intentionally fairly simple. In many cases, you'll be able to create custom views by combining existing views in this manner, though keep in mind that an alternative approach is to extend the `View` class itself. With this approach, you can implement core methods as desired and you have access to the lifecycle methods of a `View`. These `View`-specific methods 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 you've seen how you get the data for your reviews and what the `Adapter` and custom `View` we're using look like, the next thing we need to do is 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, 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; every element can have one or more 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 of the View class. Dimensions—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 can accommodate. The final layout reflects the cumulative dimensions of the parent and its child Views.

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

As figure 3.6 shows, the base ViewGroup.LayoutParams class supports height and width. From there, an AbsoluteLayout type with AbsoluteLayout.LayoutParams allows you to specify the exact *x* and *y* coordinates of the child View objects placed within.

As an alternative to absolute layout, you can use the FrameLayout, LinearLayout, and RelativeLayout subtypes, all of which support variations of LayoutParams that are derived from ViewGroup.MarginLayoutParams. A FrameLayout is intended to frame one child element, such as an image. A FrameLayout does support multiple children, but all the items are pinned to the top left—they'll overlap each other in a stack. A LinearLayout aligns child elements in either a horizontal or a vertical line.

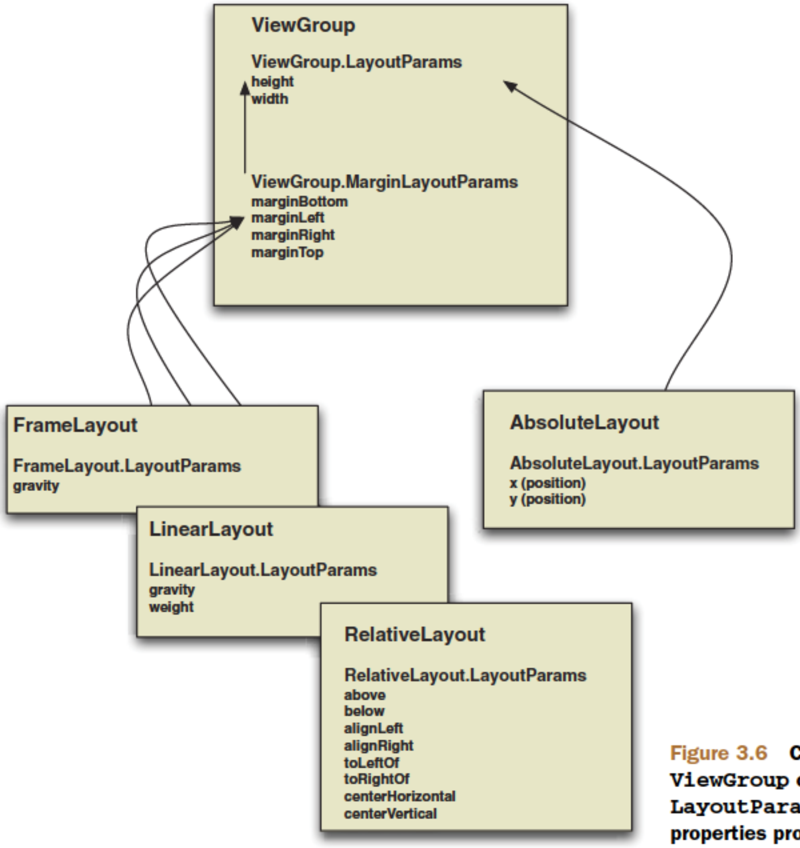


Figure 3.6 Common ViewGroup classes with LayoutParams and properties provided

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'll cover XML resources in detail in section 3.3). A RelativeLayout specifies child elements relative to each other: above, below, toLeftOf, and so on.

To summarize, the container is a ViewGroup, and a ViewGroup supports a particular type of LayoutParams. Child View elements are then added to the container and must fit into the layout specified by their parents. 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 flexible palette upon which you can construct just about any type of screen you want.

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

- An exact number
- FILL_PARENT
- WRAP_CONTENT

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

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

Layout is a two-step process: first, measurements are taken during the *measure pass*, and subsequently, the items are placed to the screen during the *layout pass*, using the associated LayoutParams. Components are drawn to the screen in the order they're found in the layout tree: parents first, then children. Note that parent views end up behind children, if they overlap in positioning.

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

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 UIs support this concept. When you're turning the pages of a book, your focus is on one particular page at a time. Computer interfaces are no different.

Though there can 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 a stylus or finger, a tap, or a keyboard press, might trigger the focus to shift to another component.

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

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

Views can also indicate a particular focus type, `DEFAULT_FOCUS` or `WEAK_FOCUS`, to set the priority of focus to either themselves (default) or their descendants (weak). In addition to hints, such as `UP`, `DOWN`, and `WEAK`, you can use the `View.requestFocus()` method directly, if you need to, to indicate that focus should be set to a particular `View` at a given time. Manipulating the focus manually should be the exception rather than the rule—the platform logic generally does what you would expect, and more importantly, what the user expects. Your application's behavior should be mindful of how other Android applications behave and it should act accordingly.

Focus changes based on event-handling logic using the `OnFocusChangeListener` object and related `setOnFocusChangedListener()` method. This takes us into the world of event handling in general.

3.2.7 Grasping events

Events are used to change the focus and for many other actions. We've already implemented several `onClickListener()` methods for buttons in listing 3.2. Those `OnClickListener` instances were connected to button presses. They were indicating events that said, "Hey, somebody pressed me." This process is the same one 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.

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.

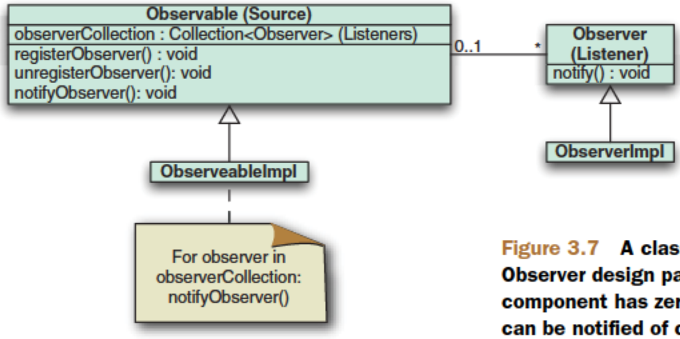

Observable (Source)
observerCollection : Collection<Observer> (Listeners)
registerObserver() : void
unregisterObserver(): void
notifyObserver(): void
0..1
*
Observer (Listener)
notify() : void
ObserveableImpl
ObserverImpl
For observer in observerCollection: notifyObserver()

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.

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` lifecycle methods we've already discussed—`onCreate()`, `onFreeze()`, and such—are also event-related, though on a different level.

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

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

One important aspect of the `View` in Android is that the interface is single-threaded. If you're calling a method on a `View`, you have to be on the UI thread. Recall that this is why we used a `Handler` in listing 3.3—to get data outside the UI thread and to notify the UI thread to update the View after the data was retrieved. The data was sent back to the `Handler` as a `Message` via the `setMessage()` event.

We're discussing events here on a fairly broad level, to make sure that the overarching concepts are clear. We're doing this because we can't cover all the event methods in the Android APIs in one chapter, yet you'll see events in examples throughout the book and in your day-to-day experiences with the platform. We'll call out event examples when necessary, and we'll cover them in more detail as we discuss specific examples.

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

3.3 Using resources

We've mentioned Android resources in several contexts up to now (we initially introduced them in chapter 1). Now we're going to revisit resources in more depth to expand on this important topic and to begin completing the third and final `Activity` in RestaurantFinder—the `ReviewDetail` screen.

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

To begin looking at resources, we'll first discuss how they're classified into types in Android, and then we'll demonstrate examples of each type of resource.

3.3.1 Supported resource types

Looking at the project structure of an Android project, the project's resources are located 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 compiled into a binary representation
- `res/raw`—Arbitrary and uncompiled files that can be added

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

3.3.2 Referencing resources in Java

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

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

```
public class ReviewDetail extends Activity {
    private static final int MENU_CALL_REVIEW = Menu.FIRST + 2;
    private static final int MENU_MAP_REVIEW = Menu.FIRST + 1;
    private static final int MENU_WEB_REVIEW = Menu.FIRST;
    private String imageLink;
    private String link;
    private TextView location;
    private TextView name;
    private TextView phone;
    private TextView rating;
    private TextView review;
    private ImageView reviewImage;
    private Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            if ((imageLink != null) && !imageLink.equals("")) {
                try {
                    URL url = new URL(imageLink);
                    URLConnection conn = url.openConnection();
                    conn.connect();
                    BufferedInputStream bis = new
BufferedInputStream(conn.getInputStream());
                    Bitmap bm = BitmapFactory.decodeStream(bis);
                    bis.close();
                    reviewImage.setImageBitmap(bm);
                } catch (IOException e) {
                    // log and or handle here
                }
            } else {
                reviewImage.setImageResource(R.drawable.no_review_image);
            }
        }
    };
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(R.layout.review_detail);
        this.name =
            (TextView) findViewById(R.id.name_detail);
        this.rating =
            (TextView) findViewById(R.id.rating_detail);
        this.location =
            (TextView) findViewById(R.id.location_detail);
        this.phone =
            (TextView) findViewById(R.id.phone_detail);
        this.review =
            (TextView) findViewById(R.id.review_detail);
        this.reviewImage =
            (ImageView) findViewById(R.id.review_image);
        RestaurantFinderApplication application =
            (RestaurantFinderApplication) getApplication();
        Review currentReview = application.getCurrentReview();
        this.link = currentReview.link;
        this.imageLink = currentReview.imageLink;
```

1 Define inflatable View items

2 Set layout using setContentView()

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

❸ **Use String and Drawable resources**

In the `ReviewDetail` class, we first define `View` components that we'll later reference from resources ❶. Next, you see a `Handler` that's used to perform a network call to populate an `ImageView` based on a URL. (This doesn't relate specifically to our current discussion of resources, but is included here for completeness. Don't worry too much about the details of this idea here; we'll talk about it more when we specifically discuss networking in chapter 5.) After the `Handler`, we set the layout and `View` tree using `setContentView(R.layout.review_detail)` ❷. This maps to an XML layout file at src/res/layout/review_detail.xml. Next, we reference some of the `View` objects in the layout file directly through resources and corresponding IDs.

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

You reference resources in code, as we've been doing here, 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 that's contained in a special file the aapt tool creates and the R.java file uses.

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

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

3.3.3 Defining views and layouts through XML resources

As we've noted in several earlier sections, views and layouts are often defined in XML[1] 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 reusable 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've already discussed: RelativeLayout, LinearLayout, FrameLayout, and so on. Within these root elements are child XML elements that comprise the view/layout tree.

A subtle but important thing to understand here is that resources in the res/layout directory don't have to be complete layouts. For example, you can define a single TextView in a layout file the same way you might define an entire tree starting from an AbsoluteLayout. Yes, this might make the layout name and path potentially confusing, but that's how it's set up. It might make more sense to have separate res/layout and res/view directories, but that might be confusing too, so keep in mind that res/layout is useful for more than layout. You might use this approach when a particularly configured View is used in multiple areas of your application. By defining it as a *standalone* resource, it can be maintained more readily over the lifetime of your project.

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

Listing 3.7 XML layout resource file for review_detail.xml

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

[1] See http://www.xml.com for more information about XML.

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

❶ **Include child element with ID**

❷ **Reference another resource**

In this file, we're using a `RelativeLayout`. 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 such as width or height. Along with layout, you can also define other `View`-related attributes in XML with counterpart XML attributes to the methods available in code, such as `android:padding`, which is analogous to the `setPadding() method`.

After we've defined the `RelativeLayout` parent itself, we add the child `View` elements. Here we're using an `ImageView` and multiple `TextView` components. Each of the components is given an ID using the form `android:id="@+id/[name]"` ❶. When an ID is established in this manner, an `int` reference is defined in the resource table and named with the specified name. Other components can reference the ID by the friendly textual name. Never use the integer value directly, as it'll change over time as your view changes. Always use the constant value defined in the `R` class!

After views are defined as resources, you can use the `Activity` method `findViewById()`to obtain a reference to a particular `View`, using the name. Then you can manipulate that `View` 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. Note that child views of layout files end up as `id` type in R.java (they're not `R.layout.name`; rather they're `R.id.name`, even though they're required to be placed in the res/layout directory).

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

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

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

3.3.4 Externalizing values

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

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

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name_criteria">RestaurantFinder – Criteria</string>
    <string name="app_name_reviews">RestaurantFinder - Reviews</string>
    <string name="app_name_review">RestaurantFinder - Review</string>
    <string name="app_short_name">Restaurants</string>
    <string name="menu_get_reviews">Get reviews</string>
    <string name="menu_web_review">Get full review</string>
    <string name="menu_map_review">Map location</string>
    <string name="menu_call_review">Call restaurant</string>
    <string name="menu_change_criteria">Change review criteria</string>
    <string name="menu_get_next_page">Get next page of results</string>
    <string name="intro_blurb_criteria">Enter review criteria</string>
    <string name="intro_blurb_detail">Review details</string>
    . . .
</resources>
```

As is evident from the strings.xml example, this is straightforward. This file uses a <string> element with a name attribute for each string value you define. We've used this file for the application name, menu buttons, labels, and alert validation messages. This format is known as *simple value* in Android terminology. This file is placed in source at the res/values/strings.xml location. In addition to strings, you can define colors and dimensions 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)
- density-independent pixels (dp)
- scaled pixels (sp)

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

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

Listing 3.9 Values resource defining reusable styles, styles.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="intro_blurb">
        <item name="android:textSize">22sp</item>
        <item name="android:textColor">#ee7620</item>
        <item name="android:textStyle">bold</item>
    </style>
    <style name="label">
        <item name="android:textSize">18sp</item>
        <item name="android:textColor">#ffffff</item>
    </style>
    <style name="edit_text">
        <item name="android:textSize">16sp</item>
        <item name="android:textColor">#000000</item>
    </style>
     . . .
</resources>
```

The Android styles approach is similar in concept to using *Cascading Style Sheets (CSS)* with HTML. Styles are defined in styles.xml and then referenced from other resources or code. Each `<style>` element has one or more `<item>` children that define a single setting. Styles are made up of the various `View` settings: sizes, colors, margins, and such. Styles are helpful because they facilitate easy reuse 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 isn't prefixed with the android: namespace; it's a custom local style, not one provided by the platform.

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

Along with styles and themes, Android supports a specific XML structure for defining arrays as a resource. 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. The following listing 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">
        <item>ANY</item>
        <item>American</item>
        <item>Barbeque</item>
        <item>Chinese</item>
        <item>French</item>
        <item>German</item>
        <item>Indian</item>
        <item>Italian</item>
        <item>Mexican</item>
        <item>Thai</item>
        <item>Vegetarian</item>
        <item>Kosher</item>
    </array>
</resources>
```

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

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'll examine is the animation resource.

3.3.5 Providing animations

Animations[2] are more complicated than other Android resources, but they're also the most visually impressive. Android allows you to define animations that can rotate, fade, move, or stretch graphics or text. Though you don't want to go overboard with a constantly blinking animated shovel, an initial splash or occasional subtle animated effect can enhance your UI.

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

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

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

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

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

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

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

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

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

This will scale the `view` element up in size on both the X and Y axes. Though we don't have any animations in the RestaurantFinder sample application by default, to see this animation work, you can add the `startAnimation` method to any view element in the

[2] For a good start in understanding the animation frameworks that are part of the SDK: http://developer-life.com/tutorials/?p=343.

code and reload the application. Animations can come in handy, so you should be aware of them. We'll cover animations and other graphics topics in detail in chapter 9.

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

3.4 Exploring the AndroidManifest file

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

Listing 3.12 The RestaurantFinder AndroidManifest.xml file

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

1 Define ReviewCriteria Activity

2 Define MAIN LAUNCHER Intent filter

```
            </intent-filter>
        </activity>
    </application>
    <uses-permission android:name="android.permission.CALL_PHONE" />
    <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

In the RestaurantFinder descriptor file, you first see the root <manifest> element declaration, which includes the application's package declaration and the Android namespace. Then you see the <application> element with both the name and icon attributes defined. You don't 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. We took this approach so we could access the Application instance to store the current Review object. The icon is also optional; if you don't specify one, a system default is used to represent your application on the main menu. It's highly recommended that you provide an attractive icon for your application to make it stand out.

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

After the ReviewCriteria Activity, you see another <activity> designation for ReviewList. This Activity also includes an <intent-filter>, but for our own action, com.msi.manning.restaurant.VIEW_LIST. This tells the platform that this Activity should be invoked for this Intent. Last in our manifest file, we have a <uses-permission> element. This element also relates to Intents and tells the platform that this application needs the CALL_PHONE permission. We touched on permissions briefly in chapter 2 and in other places in this book—mainly when we're adding a new feature and require a <uses-permission> element to allow the desired behavior.

The RestaurantFinder sample application uses a fairly basic manifest file with three activities and a series of Intents. Wrapping up the description of the manifest file completes our discussion of views, activities, resources, and working with UIs in Android.

3.5 Summary

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

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

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

4

Intents and Services

This chapter covers

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

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

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

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

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

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

4.1 Serving up RestaurantFinder with Intent

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

Intent requests are *late binding*; they're mapped and routed to a component that can handle a specified task at runtime rather than at build or compile time. One Activity tells the platform, "I need a map to Langtry, TX, US," and another component returns the result. With this approach, individual components are decoupled and can be modified, enhanced, and maintained without requiring changes to a larger application or system.

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

4.1.1 Defining Intents

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

Table 4.1 `Intent` elements and descriptions

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

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

4.1.2 *Implicit and explicit invocation*

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

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

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

4.1.3 Adding external links to RestaurantFinder

When we started the RestaurantFinder in listing 3.6, we used Intent objects to move between screens in our application. In the following listing, we finish the ReviewDetail Activity by using a new set of implicit Intent objects to link the user to other applications on the phone.

Listing 4.1 Second section of ReviewDetail, demonstrating Intent invocation

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

1 Declare Intent

2 Display web page

3 Set Intent for map menu item

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

❹ Set Intent for call menu item

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

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

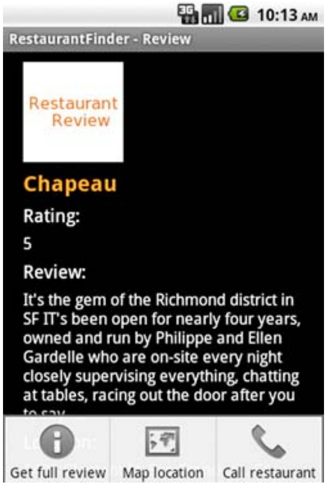
For the data component of the Intent, we use Uri.parse(link) to create a Uri. We'll look at Uri in more detail in the next section; for now, just know that this allows the correct component to answer the startActivity(Intent i) request ❷ and render the resource identified by the Uri. We don't directly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

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

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

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

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



Chapeau

Rating:

5

Review:

It's the gem of the Richmond district in SF IT's been open for nearly four years, owned and run by Philippe and Ellen Gardelle who are on-site every night closely supervising everything, chatting at tables, racing out the door after you to say

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

4.1.4 Finding your way with Intent

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

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

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

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

Let's look at these components in more detail.

ACTIONS AND CATEGORIES

Each individual IntentFilter can specify zero or more actions and zero or more categories. If the action isn't specified in the IntentFilter, it'll match any Intent; otherwise, it'll match only if the Intent has the same action.

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

You can work with action and category without specifying any data. We used this technique in the ReviewList Activity you built in chapter 3. In that example, we defined the IntentFilter in the manifest XML, as shown in the following listing.

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

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

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

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

DATA

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

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

```
audio/mpeg
```

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

1. If a scheme is present and type is *not* present, Intents with any type will match.
2. If a type is present and scheme is *not* present, Intents with any scheme will match.
3. If neither a scheme nor a type is present, only Intents with neither scheme nor type will match.
4. If an authority is specified, a scheme must also be specified.
5. If a path is specified, a scheme and an authority must also be specified.

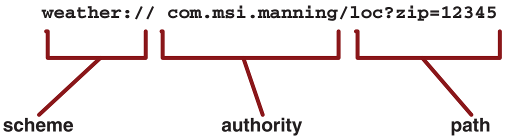
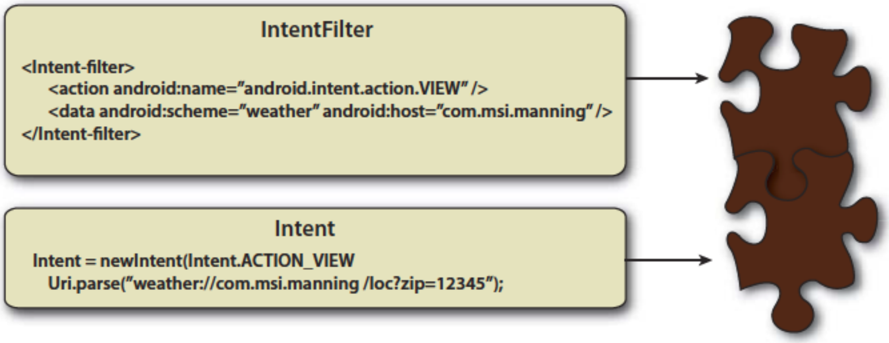


Figure 4.2 The portions of a URI that are used in Android, showing scheme, authority, and path



```
<Intent-filter>
   <action android:name="android.intent.action.VIEW" />
   <data android:scheme="weather" android:host="com.msi.manning" />
</Intent-filter>
```

```
Intent = newIntent(Intent.ACTION_VIEW
  Uri.parse("weather://com.msi.manning /loc?zip=12345");
```

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

Most matches are straightforward, but as you can see, it can get complicated. Think of `Intent` and `IntentFilter` as separate pieces of the same puzzle. When you call an `Intent` in an Android application, the system resolves the `Activity`, `Service`, or `BroadcastReceiver` to handle your request through this resolution process using the actions, categories, and data provided. The system searches all the pieces of the puzzle it has until it finds one that meshes with the `Intent` you've provided, and then it snaps those pieces together to make the late-binding connection.

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

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

4.1.5 *Taking advantage of Android-provided activities*

In addition to the examples in our RestaurantFinder application, Android ships with a useful set of core applications that allow access via the formats shown in table 4.2. Using the actions and URIs shown in table 4.2, you can hook into the built-in maps

Table 4.2 Common Android application `Intent` action and `Uri` combinations

Action	`Uri`	Description
`Intent.ACTION_CALL`	tel:phone_number	Opens the phone application and calls the specified number
`Intent.ACTION_DIAL`	tel:phone_number	Opens the phone application and dials (but doesn't call) the specified number

Table 4.2 Common Android application `Intent` action and `Uri` combinations *(continued)*

Action	`Uri`	Description
`Intent.ACTION_DIAL`	voicemail:	Opens the phone application and dials (but doesn't call) the voicemail number
`Intent.ACTION_VIEW`	geo:latitude,longitude	Opens the maps application to the specified latitude and longitude
`Intent.ACTION_VIEW`	geo:0,0?q=street+address	Opens the maps application to the specified address
`Intent.ACTION_VIEW`	http://web_address	Opens the browser application to the specified URL
`Intent.ACTION_VIEW`	https://web_address	Opens the browser application to the specified secure URL
`Intent.ACTION_WEB_SEARCH`	plain_text	Opens the browser application and uses Google Search

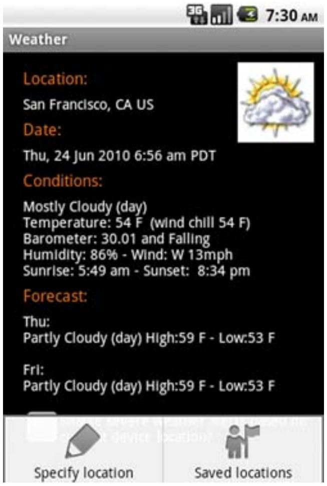
application, phone application, or browser application. By experimenting with these, you can get a feel for how `Intent` resolution works in Android.

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

4.2 Checking the weather with a custom URI

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

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



Location:
San Francisco, CA US
Date:
Thu, 24 Jun 2010 6:56 am PDT
Conditions:
Mostly Cloudy (day)
Temperature: 54 F (wind chill 54 F)
Barometer: 30.01 and Falling
Humidity: 86% - Wind: W 13mph
Sunrise: 5:49 am - Sunset: 8:34 pm
Forecast:
Thu:
Partly Cloudy (day) High:59 F - Low:53 F

Fri:
Partly Cloudy (day) High:59 F - Low:53 F

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

4.2.1 Offering a custom URI

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

Listing 4.3 The Android manifest file for the WeatherReporter application

```
<?xml version="1.0" encoding="utf-8"?>
    <manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.msi.manning.weather">
        <application android:icon="@drawable/weather_sun_clouds_120"
          android:label="@string/app_name"
          android:theme="@android:style/Theme.Black"
          android:allowClearUserData="true">
        <activity android:name="ReportViewSavedLocations"
          android:label="@string/app_name_view_saved_locations" />
        <activity android:name="ReportSpecifyLocation"              ❶
          android:label=                                            Define
            "@string/app_name_specify_location" />                  activities
        <activity android:name="ReportViewDetail"
          android:label="@string/app_name_view_detail">
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="weather"
                  android:host="com.msi.manning" />
            </intent-filter>
            <intent-filter>
                <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">          ❷ Define receiver
            <intent-filter>
                <action android:name=
                  "android.intent.action.BOOT_COMPLETED" />
            </intent-filter>
          </receiver>                                         ❸ Define
        <service android:name=".service.WeatherAlertService" />    service
    </application>
    <uses-permission
      android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
    <uses-permission
      android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name=                    Include necessary
      "android.permission.ACCESS_FINE_LOCATION" />           permissions ❹
    <uses-permission  android:name=
      "android.permission.ACCESS_LOCATION_EXTRA_COMMANDS" />
    <uses-permission android:name="android.permission.INTERNET" />
    </manifest>
```

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

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

After we define these activities, we use the <receiver> element in the manifest file to refer to a BroadcastReceiver class ❷. We'll examine BroadcastReceiver more closely in section 4.3, but for now know that an <intent-filter> associates this receiver with an Intent—in this case, for the BOOT_COMPLETED action. This filter tells the platform to invoke the WeatherAlertServiceReceiver class after it completes the boot up sequence.

We also define a Service ❸. You'll see how this Service is built, and how it polls for severe weather alerts in the background, in section 4.3. Finally, our manifest includes a set of required permissions ❹.

4.2.2 Inspecting a custom Uri

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

Listing 4.4 onStart method of the ReportViewDetail Activity

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

❶ Create database helper

❷ Get device location postal code

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

❸ **Set status of alert-enabled check box**

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

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

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

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

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

Finally, we call the `loadReport` method, which makes the call out to the Yahoo! Weather API[1] to obtain data; then we use a `Handler` to send a `Message` to update the needed UI `View` elements.

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

[1] For more on the Yahoo! Weather API, go here: http://developer.yahoo.com/weather/.

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

4.3 Checking the weather with broadcast receivers

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

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

4.3.1 Broadcasting Intent

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

Table 4.3 Methods for broadcasting intents

Method	Description
sendBroadcast(Intent intent)	Simple form for broadcasting an Intent.
sendBroadcast(Intent intent, String receiverPermission)	Broadcasts an Intent with a permission String that receivers must declare in order to receive the broadcast.
sendOrderedBroadcast(Intent intent, String receiverPermission)	Broadcasts an Intent call to the receivers one-by-one serially, stopping after a receiver consumes the message.

Table 4.3 Methods for broadcasting intents *(continued)*

Method	Description
`sendOrderedBroadcast(Intent intent, String receiverPermission, BroadcastReceiver resultReceiver, Handler scheduler, int initialCode, String initialData, Bundle initialExtras)`	Broadcasts an `Intent` and gets a response back through the provided `BroadcastReceiver`. All receivers can append data that will be returned in the `BroadcastReceiver`. When you use this method, the receivers are called serially.
`sendStickyBroadcast(Intent intent)`	Broadcasts an `Intent` that remains a short time after broadcast so that receivers can retrieve data. Applications using this method must declare the `BROADCAST_STICKY` permission.

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

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

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

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

Table 4.4 Broadcast actions provided by the Android platform

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

4.3.2 Creating a receiver

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

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

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

Listing 4.5 The WeatherAlertServiceReceiver BroadcastReceiver class

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

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

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

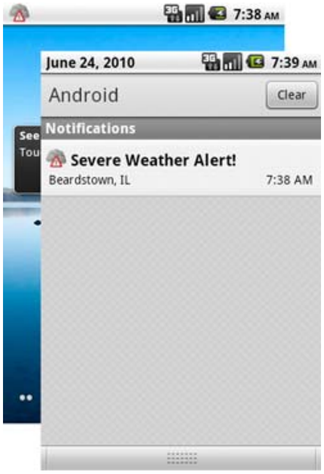
Our receiver has started the WeatherAlertService, which will run in the background and warn users of severe weather in the forecast with a Notification-based alert. Let's look more deeply into the concept of an Android Service.

4.4 Building a background weather service

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

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

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



Severe Weather Alert!
Beardstown, IL 7:38 AM

Figure 4.5 Warning from a background application about severe weather

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

```
public class WeatherAlertService extends Service {
    private static final String LOC = "LOC";
    private static final String ZIP = "ZIP";
    private static final long ALERT_QUIET_PERIOD = 10000;
    private static final long ALERT_POLL_INTERVAL = 15000;
    public static String deviceLocationZIP = "94102";
    private Timer timer;
    private DBHelper dbHelper;
    private NotificationManager nm;
    private TimerTask task = new TimerTask() {
        public void run() {
            List<Location> locations =
              dbHelper.getAllAlertEnabled();
            for (Location loc : locations) {
                WeatherRecord record = loadRecord(loc.zip);
                if (record.isSevere()) {
                    if ((loc.lastalert +
                      WeatherAlertService.ALERT_QUIET_PERIOD)
            < System.currentTimeMillis()) {
                        loc.lastalert = System.currentTimeMillis();
                        dbHelper.update(loc);
                        sendNotification(loc.zip, record);
                    }
                }
            }
            . . . device location alert omitted for brevity
        }
    };
    private Handler handler = new Handler() {
        public void handleMessage(Message msg) {
```

1 Get locations with alerts enabled

2 Fire alert if severe

```
            notifyFromHandler((String) msg.getData()
               .get(WeatherAlertService.LOC), (String) msg.getData()
               .get(WeatherAlertService.ZIP));                       ❸ Notify UI from handler
        }
    };
    @Override
    public void onCreate() {
        dbHelper = new DBHelper(this);
        timer = new Timer();                                  ❹ Initialize timer
        timer.schedule(task, 5000,
            WeatherAlertService.ALERT_POLL_INTERVAL);
        nm = (NotificationManager)
          getSystemService(Context.NOTIFICATION_SERVICE);
    }
   . . . onStart with LocationManager and LocationListener \
           omitted for brevity
    @Override
    public void onDestroy() {
        super.onDestroy();
        dbHelper.cleanup();                                   ❺ Clean up database connection
    }
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
    protected WeatherRecord loadRecord(String zip) {
        final YWeatherFetcher ywh =  #10
          new YWeatherFetcher(zip, true);
        return ywh.getWeather();
    }
    private void sendNotification(String zip,
      WeatherRecord record) {   #11
        Message message = Message.obtain();
        Bundle bundle = new Bundle();
        bundle.putString(WeatherAlertService.ZIP, zip);
        bundle.putString(WeatherAlertService.LOC, record.getCity()
          + ", " + record.getRegion());
        message.setData(bundle);
        handler.sendMessage(message);
    }
    private void notifyFromHandler(String location, String zip) {      ❻ Display actionable notification
        Uri uri = Uri.parse("weather://com.msi.manning/loc?zip=" + zip);
        Intent intent = new Intent(Intent.ACTION_VIEW, uri);
        PendingIntent pendingIntent =
        PendingIntent.getActivity(this, Intent.FLAG_ACTIVITY_NEW_TASK,
            intent,PendingIntent.FLAG_ONE_SHOT);
        final Notification n =
            new Notification(R.drawable.severe_weather_24,
                 "Severe Weather Alert!",
               System.currentTimeMillis());
        n.setLatestEventInfo(this, "Severe Weather Alert!",
            location, pendingIntent);
        nm.notify(Integer.parseInt(zip), n);
    }
}
```

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

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

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

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

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

Next, we override the `Service` lifecycle methods, starting with `onCreate`. Here comes the meat of our `Service`: a `Timer` ❹ that we configure to repeatedly fire. For as long as the `Service` continues to run, the timer will allow us to update weather information. After `onCreate,` we see `onDestroy`, where we clean up our database connection ❺. `Service` classes provide these lifecycle methods so you can control how resources are allocated and deallocated, similar to `Activity` classes.

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

Next, we implement our helper methods. First, `loadRecord` calls out to the Yahoo! Weather API via `YWeatherFetcher`. (We'll cover networking tasks, similar to those this class performs, in chapter 6.) Then `sendNotification` configures a `Message` with location details to activate the `Handler` we declared earlier. Last of all, you see the `notifyFromHandler` method. This method fires off a `Notification` with `Intent` objects that will call back into the WeatherReporter `Activity` if the user clicks on the `Notification` ❻.

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

A warning about long-running services

Our sample application starts a `Service` and leaves it running in the background. Our service is designed to have a minimal footprint, but Android best practices discourage long-running services. Services that run continually and constantly use the network or perform CPU-intensive tasks will eat up the device's battery life and might slow down other operations. Even worse, because they run in the background, the user won't know what applications are to blame for her device's poor performance. The OS will eventually kill running services if it needs to acquire additional memory, but otherwise won't interfere with poorly designed services. If your use case no longer requires the service, you should stop it. If you do require a long-running service, you might want to give the user the option of whether to use it.

4.5 Communicating with the WeatherAlertService from other apps

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

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

4.5.1 Android Interface Definition Language

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

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

Listing 4.7 IWeatherReporter.aidl remote IDL file

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

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

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

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

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

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

The interface generated through AIDL includes an inner static abstract class named `Stub` 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 to the remote object and use it to invoke remote methods. The AIDL process generates a `Proxy` class (another inner class, this time inside `Stub`) that connects all these components and returns to callers from the `asInterface` method. Figure 4.6 depicts this IPC local/remote relationship.

Table 4.5 Android IDL allowed types

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

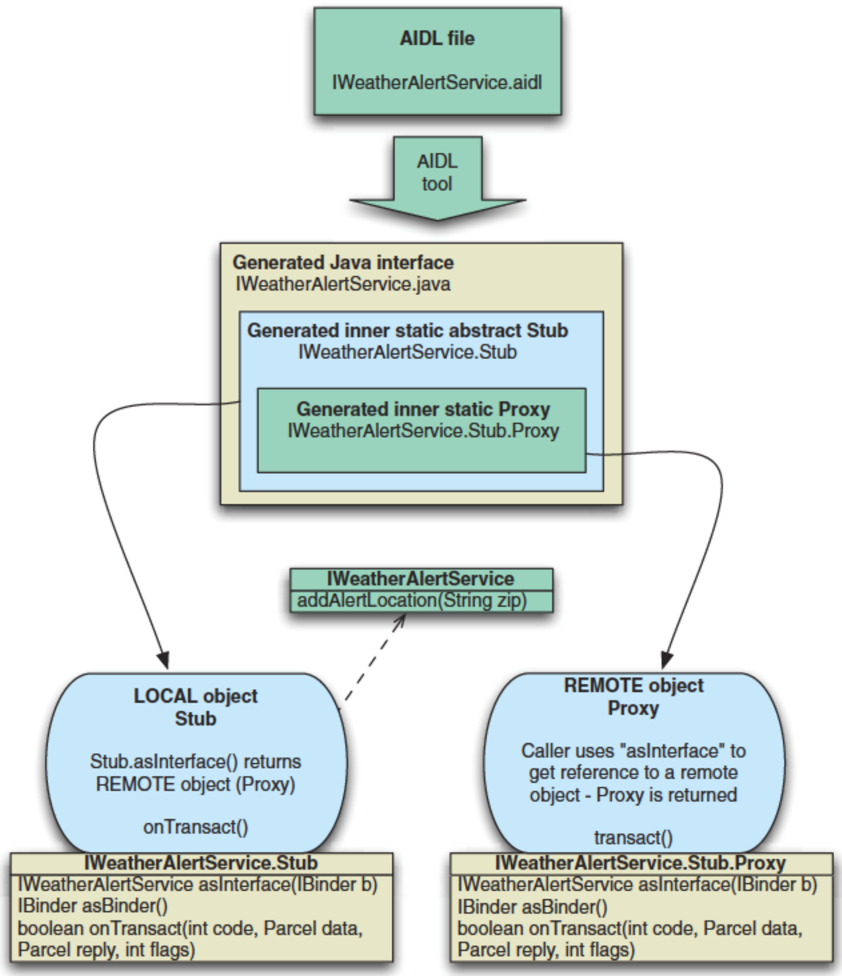

AIDL file
IWeatherAlertService.aidl
AIDL tool
Generated Java interface
IWeatherAlertService.java
Generated inner static abstract Stub
IWeatherAlertService.Stub
Generated inner static Proxy
IWeatherAlertService.Stub.Proxy
IWeatherAlertService
addAlertLocation(String zip)
LOCAL object
Stub
Stub.asInterface() returns REMOTE object (Proxy)
onTransact()
REMOTE object
Proxy
Caller uses "asInterface" to get reference to a remote object - Proxy is returned
transact()
IWeatherAlertService.Stub
IWeatherAlertService asInterface(IBinder b)
IBinder asBinder()
boolean onTransact(int code, Parcel data, Parcel reply, int flags)
IWeatherAlertService.Stub.Proxy
IWeatherAlertService asInterface(IBinder b)
IBinder asBinder()
boolean onTransact(int code, Parcel data, Parcel reply, int flags)

Figure 4.6 Diagram of the Android AIDL process

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

4.5.2 *Binder and Parcelable*

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

The `IBinder.transact()` method and corresponding `Binder.onTransact()` method form the backbone of the remoting process. Each method you define using

AIDL is handled synchronously through the transaction process, enabling the same semantics as if the method were local.

All the objects you pass in and out through the interface methods that you define using AIDL use the transact process. These objects must be `Parcelable` in order to be able to be placed inside 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 only the default allowable types in your interface definition files—primitives, `String`, `CharSequence`, `List`, and `Map`—AIDL automatically handles everything.

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

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

4.5.3 Exposing a remote interface

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

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

Listing 4.8 Implementing a weather service that publishes a remotable object

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

❷ Return IBinder representing remotable object

Our concrete instance of the generated AIDL Java interface must return an `IBinder` to any caller that binds to this `Service`. We create an implementation by extending the `Stub` class that the aidl tool generated ❶. Recall that this `Stub` class implements the AIDL interface and extends `Binder`. After we've defined our `IBinder`, we can create and return it from the `onBind` method ❷.

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

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

Listing 4.9 Exporting a service for other applications to access

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

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

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

4.5.4 Binding to a Service

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

When an `Activity` class binds to a `Service` using the `Context.bindService(Intent i, ServiceConnection connection, int flags)` method, the `ServiceConnection` object that we pass in will send several callbacks from the `Service` back to the `Activity`. The callback `onServiceConnected (ComponentName className, IBinder binder)` lets you know when the binding process completes. The platform automatically injects the `IBinder` returned from the service's `onBind` method into this callback, where you can save it for future calls. The following listing shows an `Activity` that binds to our weather reporting service and invokes remote methods on it. You can see the complete source code for this project in the chapter downloads.

Listing 4.10 Binding to a `Service` within an `Activity`

```
package com.msi.manning.weatherchecker;
. . . Imports omitted for brevity
public class WeatherChecker extends Activity {
                                                         ❶ Use generated
   private IWeatherReporter reporter;                      interface
   private boolean bound;
   private EditText zipEntry;
   private Handler uiHandler;                            ❷ Define
                                                           ServiceConnection
   private ServiceConnection connection =                  behavior
      new ServiceConnection() {
      public void onServiceConnected
        (ComponentName name, IBinder service) {
         reporter = IWeatherReporter.Stub.                 ❸ Retrieve remotely
           asInterface(service);                             callable interface
         Toast.makeText(WeatherChecker.this, "Connected to Service",
               Toast.LENGTH_SHORT).show();
         bound = true;
      }
      public void onServiceDisconnected
        (ComponentName name) {
         reporter = null;
         Toast.makeText(WeatherChecker.this, "Disconnected from Service",
               Toast.LENGTH_SHORT).show();
         bound = false;
      }
   };

. . . onCreate method omitted for brevity

   public void checkWeather(View caller) {
      final String zipCode = zipEntry.getText().toString();
      if (zipCode != null && zipCode.length() == 5) {        Don't block
         new Thread() {                                      UI thread
            public void run() {
               try {
```

```
               final String currentWeather =
                  reporter.getWeatherFor(zipCode);
               uiHandler.post(new Runnable() {
                  public void run() {
                     Toast.makeText(WeatherChecker.this, currentWeather,
                           Toast.LENGTH_LONG).show();
                  }
               });
            } catch (DeadObjectException e) {
               e.printStackTrace();
            } catch (RemoteException e) {
               e.printStackTrace();
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
      }.start();
   }
}

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

public void onPause() {
   super.onPause();
```

4 Invoke remote method

Show feedback on UI thread

Show feedback on UI thread

5 Start binding to service

```
      if (this.bound){
         bound = false;
         unbindService(connection);
      }
   }
}
```

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

We use the `ServiceConnection` object ❷ to bind and unbind using `Context` methods. After a `Service` is bound, the platform notifies us through the `onServiceConnected` callback. This callback returns the remote `IBinder` reference, which we assign to the remotable type ❸ so we can invoke it later. Next, a similar `onServiceDisconnected` callback will fire when a `Service` is unbound.

After we've established a connection, we can use the AIDL-generated interface to perform the operations it defines ❹. When we call `getWeatherFor` (or later, `addLocation`), Android will dispatch our invocation across the process boundary, where the service we created in listing 4.8 will execute the methods. The return values will be sent back across the process boundary and arrive as shown at ❹. This sequence can take a long time, so you should avoid calling remote methods from the UI thread.

In `onStart`, we establish the binding using `bindService` ❺; later, in `onPause`, we use `unbindService`. The system can choose to clean up a `Service` that's been bound but not started. You should always unbind an unused `Service` so the device can reclaim its resources and perform better. Let's look more closely at the difference between starting and binding a service.

4.5.5 Starting versus binding

`Services` serve two purposes in Android, and you can use them in two different ways:

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

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

Binding to a `Service`, as you did with `WeatherReporterService`, gave you a handle to a remote object, which let you call the service's exported methods from an `Activity`. Because every Android application runs in its own process, using a bound `Service` lets you pass data between processes.

The actual process of marshaling and unmarshaling remotable objects across process boundaries is complicated. Fortunately, you don't have to deal with all the internals, because Android handles all the complexity through AIDL. Instead, you can stick to a simple recipe that will enable you to create and use remotable objects:

1. Define your interface using AIDL, in the form of a .aidl file; see listing 4.7.
2. Generate a Java interface for your .aidl file. This happens automatically in Eclipse.
3. Extend from the generated `.Stub` class and implement your interface methods; see listing 4.8.
4. Expose your interface to clients through a `Service` and the `Service onBind(Intent i)` method; see listing 4.8.
5. If you want to make your service available to other applications, export the `Service` in your manifest; see listing 4.9.
6. Client applications will bind to your `Service` with a `ServiceConnection` to get a handle to the remotable object; see listing 4.10.

As we discussed earlier in the chapter, services running in the background can have a detrimental impact on overall device performance. To mitigate these problems, Android enforces a special lifecycle for services, which we're going to discuss now.

4.5.6 Service lifecycle

We want our weather alerting service to constantly lurk in the background, letting us know of potential dangers. On the other hand, we want our weather reporting service to run only while another application actually needs it. Services follow their own well-defined process phases, similar to those followed by an `Activity` or an `Application`. A `Service` will follow a different lifecycle, depending on whether you start it, bind it, or both.

SERVICE-STARTED LIFECYCLE

If you start a `Service` by calling `Context.startService(Intent service, Bundle b)`, as shown in listing 4.5, it runs in the background whether or not anything binds to it. If the service hasn't been created, the `Service onCreate()` method is called. The `onStart(int id, Bundle args)` method is called each time someone tries to start the service, whether or not it's already running. Additional instances of the `Service` won't be created.

The `Service` will continue to run in the background until someone explicitly stops it with the `Context.stopService()` method or when the `Service` calls its own `stopSelf()` method. You should also keep in mind that the platform might kill services if resources are running low, so your application needs to be able to react accordingly. You can choose to restart the service automatically, fall back to a more limited feature set without it, or take some other appropriate action.

SERVICE-BOUND LIFECYCLE

If an `Activity` binds a `Service` by calling `Context.bindService(Intent service, ServiceConnection connection, int flags)`, as shown in listing 4.10, it'll run as long as the connection is open. An `Activity` establishes the connection using the `Context` and is also responsible for closing it.

When a Service is only bound in this manner and not also started, its onCreate() method is invoked, but onStart(int id, Bundle args) is *not* used. In these cases, the platform can stop and clean up the Service after it's unbound.

SERVICE-STARTED AND SERVICE-BOUND LIFECYCLE

If a Service is both started and bound, it'll keep running in the background, much like in the started lifecycle. In this case, both onStart(int id, Bundle args) and onCreate() are called.

CLEANING UP WHEN A SERVICE STOPS

When a Service stops, its onDestroy() method is invoked. Inside onDestroy(), every Service should perform final cleanup, stopping any spawned threads, terminating network connections, stopping services it had started, and so on.

And that's it! From birth to death, from invocation to dismissal, you've learned how to wrangle an Android Service. They might seem complex, but they offer extremely powerful capabilities that can go far beyond what a single foregrounded application can offer.

4.6 Summary

In this chapter, we covered a broad swath of Android territory. We first focused on the Intent component, seeing how it works, how it resolves using Intent-Filter objects, and how to take advantage of built-in platform-provided Intent handlers. We also looked at the differences between explicit Intent invocation and implicit Intent invocation, and the reasons you might choose one type over another. Along the way, you completed the RestaurantFinder sample application, and with just a bit more code, you drastically expanded the usefulness of that app by tapping into preloaded Android applications.

After we covered the Intent class, we moved on to a new sample application, WeatherReporter. You saw how a BroadcastReceiver could respond to notifications sent by the platform or other applications. You used the receiver to listen for a boot event and start the Service. The Service sends notification alerts from the background when it learns of severe weather events. You also saw another flavor of Service, one that provides communication between different processes. Our other weather service offered an API that third-party developers could use to leverage the low-level network and storage capabilities of our weather application. We covered the difference between starting and binding services, and you saw the moving parts behind the Android IPC system, which uses the AIDL to standardize communication between applications.

By seeing all these components interact in several complete examples, you now understand the fundamentals behind Android Intents and Services. In the next chapter, you'll see how to make services and other applications more useful by using persistent storage. We'll look at the various options Android provides for retrieving and storing data, including preferences, the file system, databases, and how to create a custom ContentProvider.

5 Storing and retrieving data

This chapter covers

- Storing and retrieving data with `SharedPreferences`
- Using the filesystem
- Working with a SQLite database
- Accessing and building a `ContentProvider`

Android provides several ways to store and share data, including access to the filesystem, a local relational database through SQLite, and a preferences system that allows you to store simple key/value pairs within applications. In this chapter, we'll start with preferences and you'll create a small sample application to exercise those concepts. From there, you'll create another sample application to examine using the filesystem to store data, both internal to our application and external using the platform's Secure Digital (SD) card support. You'll also see how to create and access a database.

Beyond the basics, Android also allows applications to share data through a clever URI-based approach called a `ContentProvider`. This technique combines several other Android concepts, such as the URI-based style of intents and the

Cursor result set seen in SQLite, to make data accessible across different applications. To demonstrate how this works, you'll create another small sample application that uses built-in providers, then we'll walk through the steps required to create your own ContentProvider.

We'll begin with preferences, the simplest form of data storage and retrieval Android provides.

5.1 Using preferences

If you want to share simple application data from one Activity to another, use a SharedPreferences object. You can save and retrieve data, and also choose whether to make preferences private to your application or accessible to other applications on the same device.

5.1.1 Working with SharedPreferences

You access a SharedPreferences object through your current Context, such as the Activity or Service. Context defines the method getSharedPreferences(String name, int accessMode) that allows you to get a preferences handle. The name you specify will be the name for the file that backs these preferences. If no such file exists when you try to get preferences, one is automatically created. The access mode refers to what permissions you want to allow.

The following listing demonstrates allowing the user to input and store data through SharedPreferences objects with different access modes.

Listing 5.1 Storing SharedPreferences using different modes

```
package com.msi.manning.chapter5.prefs;
// imports omitted for brevity
public class SharedPrefTestInput extends Activity {
    public static final String PREFS_PRIVATE = "PREFS_PRIVATE";
    public static final String PREFS_WORLD_READ = "PREFS_WORLD_READABLE";
    public static final String PREFS_WORLD_WRITE = "PREFS_WORLD_WRITABLE";
    public static final String PREFS_WORLD_READ_WRITE =
      "PREFS_WORLD_READABLE_WRITABLE";
    public static final String KEY_PRIVATE = "KEY_PRIVATE";
    public static final String KEY_WORLD_READ = "KEY_WORLD_READ";
    public static final String KEY_WORLD_WRITE = "KEY_WORLD_WRITE";
    public static final String KEY_WORLD_READ_WRITE =
      "KEY_WORLD_READ_WRITE";
    . . . view element variable declarations omitted for brevity
    private SharedPreferences prefsPrivate;
    private SharedPreferences prefsWorldRead;
    private SharedPreferences prefsWorldWrite;
    private SharedPreferences prefsWorldReadWrite;
    @Override
    public void onCreate(Bundle icicle) {
    ... view inflation omitted for brevity
        button.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                boolean valid = validate();
```

❶ Declare SharedPreferences variables

```
        if (valid) {
          prefsPrivate =
           getSharedPreferences(
            SharedPrefTestInput.PREFS_PRIVATE,          ❷ Use Context.getSharedPreferences for references
            Context.MODE_PRIVATE);                      ❸ Use different modes
          prefsWorldRead =
             getSharedPreferences(
              SharedPrefTestInput.PREFS_WORLD_READ,
            Context.MODE_WORLD_READABLE);
          prefsWorldWrite =
           getSharedPreferences(
            SharedPrefTestInput.PREFS_WORLD_WRITE,
            Context.MODE_WORLD_WRITEABLE);
          prefsWorldReadWrite =
           getSharedPreferences(
            SharedPrefTestInput.PREFS_WORLD_READ_WRITE,
            Context.MODE_WORLD_READABLE
            + Context.MODE_WORLD_WRITEABLE);
          Editor prefsPrivateEditor =
           prefsPrivate.edit();                         ❹ Get SharedPreferences editor
          Editor prefsWorldReadEditor =
           prefsWorldRead.edit();
          Editor prefsWorldWriteEditor =
           prefsWorldWrite.edit();
          Editor prefsWorldReadWriteEditor =
           prefsWorldReadWrite.edit()
          prefsPrivateEditor.putString(
           SharedPrefTestInput.KEY_PRIVATE,
            inputPrivate.getText.toString());           ❺ Store values with editor
          prefsWorldReadEditor.putString(
           SharedPrefTestInput.KEY_WORLD_READ,
            inputWorldRead.getText().toString());
          prefsWorldWriteEditor.putString(
           SharedPrefTestInput.KEY_WORLD_WRITE,
            inputWorldWrite.getText().toString());
          prefsWorldReadWriteEditor.putString(
           SharedPrefTestInput.KEY_WORLD_READ_WRITE,
            inputWorldReadWrite.getText().toString());
          prefsPrivateEditor.commit();                  ❻ Persist changes
          prefsWorldReadEditor.commit();
          prefsWorldWriteEditor.commit();
          prefsWorldReadWriteEditor.commit();
          Intent intent =
           new Intent(SharedPrefTestInput.this,
            SharedPrefTestOutput.class);
          startActivity(intent);
        }
      }
    });
  }
 . . . validate omitted for brevity
}
```

After you have a `SharedPreferences` variable ❶, you can acquire a reference through the `Context` ❷. Note that for each `SharedPreferences` object we get, we use a different constant value for the access mode, and in some cases we also add modes ❸. We repeat this coding for each mode we retrieve. Modes specify whether the preferences should be private, world-readable, or world-writable.

To modify preferences, you must get an `Editor` handle ❹. With the `Editor`, you can set `String`, `boolean`, `float`, `int`, and `long` types as key/value pairs ❺. This limited set of types can be restrictive, but often preferences are adequate, and they're simple to use.

After storing with an `Editor`, which creates an in-memory `Map`, you have to call `commit()` to persist it to the preferences backing file ❻. After data is committed, you can easily get it from a `SharedPreferences` object. The following listing gets and displays the data that was stored in listing 5.1.

Listing 5.2 Getting `SharedPreferences` data stored in the same application

```
package com.msi.manning.chapter5.prefs;
// imports omitted for brevity
public class SharedPrefTestOutput extends Activity {
    . . . view element variable declarations omitted for brevity
    private SharedPreferences prefsPrivate;
    private SharedPreferences prefsWorldRead;
    private SharedPreferences prefsWorldWrite;
    private SharedPreferences prefsWorldReadWrite;
    . . . onCreate omitted for brevity
    @Override
    public void onStart() {
        super.onStart();
        prefsPrivate =
          getSharedPreferences(SharedPrefTestInput.PREFS_PRIVATE,
            Context.MODE_PRIVATE);
        prefsWorldRead =
          getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_READ,
            Context.MODE_WORLD_READABLE);
        prefsWorldWrite =
          getSharedPreferences(SharedPrefTestInput.PREFS_WORLD_WRITE,
            Context.MODE_WORLD_WRITEABLE);
        prefsWorldReadWrite =
        getSharedPreferences(
          SharedPrefTestInput.PREFS_WORLD_READ_WRITE,
          Context.MODE_WORLD_READABLE
          + Context.MODE_WORLD_WRITEABLE);
        outputPrivate.setText(prefsPrivate.getString(
          SharedPrefTestInput.KEY_PRIVATE, "NA"));
        outputWorldRead.setText(prefsWorldRead.getString(          Get values ❶
          SharedPrefTestInput.KEY_WORLD_READ, "NA"));
        outputWorldWrite.setText(prefsWorldWrite.getString(
          SharedPrefTestInput.KEY_WORLD_WRITE, "NA"));
        outputWorldReadWrite.setText(prefsWorldReadWrite.getString(
          SharedPrefTestInput.KEY_WORLD_READ_WRITE,
          "NA"));
    }
}
```

To retrieve previously stored values, we again declare variables and assign references. When these are in place, we can get values using methods such as `getString(String key, String default)` ❶. The `default` value is returned if no data was previously stored with that key.

Setting and getting preferences is straightforward. Access modes, which we'll focus on next, add a little more complexity.

5.1.2 Preference access permissions

You can open and create `SharedPreferences` with any combination of several `Context` mode constants. Because these values are `int` types, you can add them, as in listings 5.1 and 5.2, to combine permissions. The following mode constants are supported:

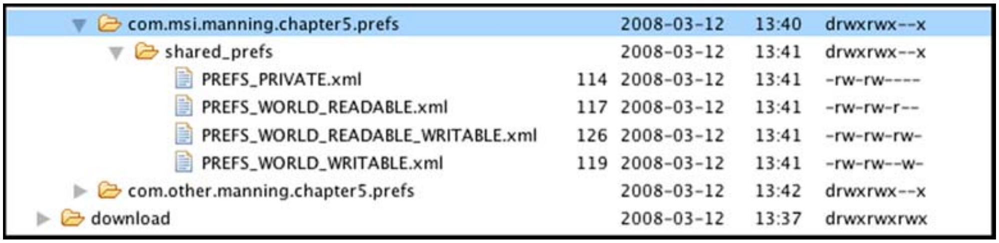
- `Context.MODE_PRIVATE` *(value 0)*
- `Context.MODE_WORLD_READABLE` *(value 1)*
- `Context.MODE_WORLD_WRITEABLE` *(value 2)*

These modes allow you to tune who can access this preference. If you take a look at the filesystem on the emulator after you've created `SharedPreferences` objects (which themselves create XML files to persist the data), you can see how setting permissions works using a Linux-based filesystem.

Figure 5.1 shows the Android Eclipse plug-in File Explorer view. Within the explorer, you can see the Linux-level permissions for the `SharedPreferences` XML files that we created from the `SharedPreferences` in listing 5.1.

Each Linux file or directory has a type and three sets of permissions, represented by a `drwxrwxrwx` notation. The first character indicates the type (`d` means directory, `-` means regular file type, and other types such as symbolic links have unique types as well). After the type, the three sets of `rwx` represent the combination of read, write, and execute permissions for user, group, and *world*, in that order. Looking at this notation, we can tell which files are accessible by the user they're owned by, by the group they belong to, or by everyone else on the device. Note that the user and group always have full permission to read and write, whereas the final set of permissions fluctuates based on the preference's mode.

Android puts `SharedPreferences` XML files in the /data/data/PACKAGE_NAME/shared_prefs path on the filesystem. An application or package usually has its own



```
▼ 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 The Android File Explorer view showing preferences file permissions

Directories with the *world* x permission

In Android, each package directory is created with the *world* x permission. This permission means anyone can search and list the files in the directory, which means that Android packages have directory-level access to one another's files. From there, file-level access determines file permissions.

user ID. When an application creates files, including SharedPreferences, they're owned by that application's user ID. To allow other applications to access these files, you have to set the *world* permissions, as shown in figure 5.1.

If you want to access another application's files, you must know the starting path. The path comes from the Context. To get files from another application, you have to know and use that application's Context. Android doesn't officially condone sharing preferences across multiple applications; in practice, apps should use a content provider to share this kind of data. Even so, looking at SharedPreferences does show the underlying data storage models in Android. The following listing shows how to get the SharedPreferences we set in listing 5.1 again, this time from a different application (different .apk and different package).

Listing 5.3 Getting SharedPreferences data stored in a different application

```
package com.other.manning.chapter5.prefs;
. . . imports omitted for brevity
public class SharedPrefTestOtherOutput extends Activity {
    . . . constants and variable declarations omitted for brevity
    . . . onCreate omitted for brevity
    @Override
    public void onStart() {
        super.onStart();
        Context otherAppsContext = null;
        try {
            otherAppsContext =
              createPackageContext("com.msi.manning.chapter5.prefs",
                Context.MODE_WORLD_WRITEABLE);
        } catch (NameNotFoundException e) {
            // log and/or handle
        }
        prefsPrivate =
          otherAppsContext.getSharedPreferences(
            SharedPrefTestOtherOutput.PREFS_PRIVATE, 0);
        prefsWorldRead =
          otherAppsContext.getSharedPreferences(
            SharedPrefTestOtherOutput.PREFS_WORLD_READ, 0);
        prefsWorldWrite =
          otherAppsContext.getSharedPreferences(
            SharedPrefTestOtherOutput.PREFS_WORLD_WRITE, 0);
        prefsWorldReadWrite =
          otherAppsContext.getSharedPreferences(
            SharedPrefTestOtherOutput.PREFS_WORLD_READ_WRITE, 0);
        outputPrivate.setText(
```

1 Use different package

2 Get another application's context

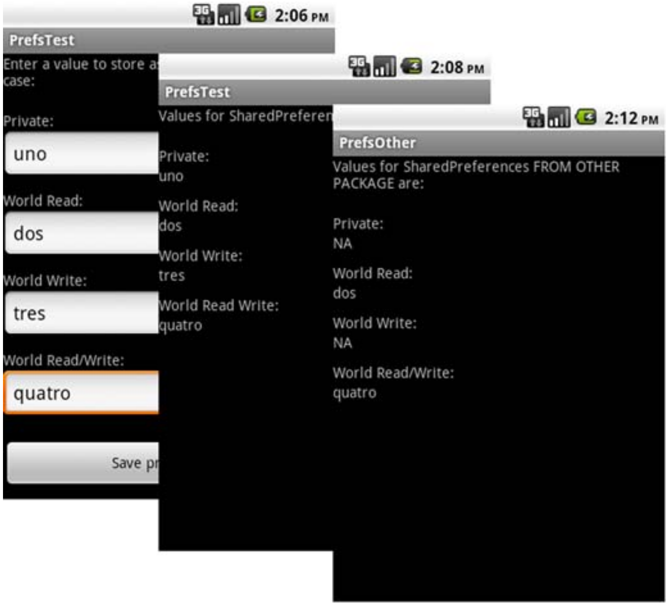
3 Use otherAppsContext

```
             prefsPrivate.getString(
               SharedPrefTestOtherOutput.KEY_PRIVATE, "NA"));
        outputWorldRead.setText(
          prefsWorldRead.getString(
            SharedPrefTestOtherOutput.KEY_WORLD_READ, "NA"));
        outputWorldWrite.setText(
          prefsWorldWrite.getString(
            SharedPrefTestOtherOutput.KEY_WORLD_WRITE, "NA"));
        outputWorldReadWrite.setText(
          prefsWorldReadWrite.getString(
            SharedPrefTestOtherOutput.KEY_WORLD_READ_WRITE,"NA"));
    }
}
```

To get one application's `SharedPreferences` from another application's package ❶, we use the `createPackageContext(String contextName, int mode)` method ❷. When we have the other application's `Context`, we can use the same names for the `SharedPreferences` objects that the other application created to access those preferences ❸.

With these examples, we now have one application that sets and gets `SharedPreferences`, and a second application with a different .apk file that gets the preferences set by the first. The composite screen shot shown in figure 5.2 shows what the apps look like. *NA* indicates a preference we couldn't access from the second application, either as the result of permissions that were set or because no permissions had been created.

Though `SharedPreferences` are ultimately backed by XML files on the Android filesystem, you can also directly create, read, and manipulate files, as we'll discuss in the next section.



Enter a value to store a
case:

Private:
uno

World Read:
dos

World Write:
tres

World Read/Write:
quatro

Values for SharedPreferen

Private:
uno

World Read:
dos

World Write:
tres

World Read Write:
quatro

Values for SharedPreferences FROM OTHER PACKAGE are:

Private:
NA

World Read:
dos

World Write:
NA

World Read/Write:
quatro

Figure 5.2 Two separate applications getting and setting `SharedPreferences`

5.2 Using the filesystem

Android's filesystem is based on Linux and supports mode-based permissions. You can access this filesystem in several ways. You can create and read files from within applications, you can access raw resource files, and you can work with specially compiled custom XML files. In this section, we'll explore each approach.

5.2.1 Creating files

Android's stream-based system of manipulating files will feel familiar to anyone who's written I/O code in Java SE or Java ME. You can easily create files in Android and store them in your application's data path. The following listing demonstrates how to open a `FileOutputStream` and use it to create a file.

Listing 5.4 Creating a file in Android from an `Activity`

```
public class CreateFile extends Activity {
    private EditText createInput;
    private Button createButton;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.create_file);
        createInput =
          (EditText) findViewById(R.id.create_input);
        createButton =
          (Button) findViewById(R.id.create_button);
        createButton.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                FileOutputStream fos = null;
                try {
                    fos = openFileOutput("filename.txt",
                      Context.MODE_PRIVATE);
                    fos.write(createInput.getText().
                      toString().getBytes());
                } catch (FileNotFoundException e) {
                    Log.e("CreateFile", e.getLocalizedMessage());
                } catch (IOException e) {
                    Log.e("CreateFile", e.getLocalizedMessage());
                } finally {
                    if (fos != null) {
                        try {
                            fos.flush();
                            fos.close();
                        } catch (IOException e) {
                            // swallow
                        }
                    }
                }
                startActivity(
                  new Intent(CreateFile.this, ReadFile.class));
            }
        });
    }
}
```

1 Use openFileOutput

2 Write data to stream

3 Flush and close stream

Android provides a convenience method on `Context` to get a `FileOutputStream`—namely `openFileOutput(String name, int mode)` ❶. Using this method, you can create a stream to a file. That file will ultimately be stored at the data/data/[PACKAGE_NAME]/files/file.name path on the platform. After you have the stream, you can write to it as you would with typical Java ❷. After you're finished with a stream, you should flush and close it to clean up ❸.

Reading from a file within an application context (within the package path of the application) is also simple; in the next section we'll show you how.

5.2.2 Accessing files

Similarly to `openFileOutput`, the `Context` also has a convenience `openFileInput` method. You can use this method to access a file on the filesystem and read it in, as shown in the following listing.

Listing 5.5 Accessing an existing file in Android from an `Activity`

```
public class ReadFile extends Activity {
    private TextView readOutput;
    private Button gotoReadResource;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_file);
        readOutput =
          (TextView) findViewById(R.id.read_output);
        FileInputStream fis = null;
        try {
            fis = openFileInput("filename.txt");
            byte[] reader = new byte[fis.available()];
            while (fis.read(reader) != -1) {}
            readOutput.setText(new String(reader));
        } catch (IOException e) {
            Log.e("ReadFile", e.getMessage(), e);
        } finally {
            if (fis != null) {
                try {
                    fis.close();
                } catch (IOException e) {
                    // swallow
                }
            }
        }
        . . . goto next Activity via startActivity omitted for brevity
    }
}
```

❶ **Use openFileInput for stream**

❷ **Read data from stream**

For input, you use `openFileInput(String name, int mode)` to get the stream ❶, and then read the file into a byte array as with standard Java ❷. Afterward, close the stream properly to avoid hanging on to resources.

With `openFileOutput` and `openFileInput`, you can write to and read from any file within the files directory of the application package you're working in. Also, as we

Running a bundle of apps with the same user ID

Occasionally, setting the user ID of your application can be extremely useful. For instance, if you have multiple applications that need to share data with one another, but you also don't want that data to be accessible outside that group of applications, you might want to make the permissions private and share the UID to allow access. You can allow a shared UID by using the `sharedUserId` attribute in your manifest: `android:sharedUserId="YourID"`.

discussed in the previous section, you can access files across different applications if the permissions allow it and if you know the package used to obtain the full path to the file.

In addition to creating files from within your application, you can push and pull files to the platform using the `adb` tool, described in section 2.2.3. The File Explorer window in Eclipse provides a UI for moving files on and off the device or simulator. You can optionally put such files in the directory for your application; when they're there, you can read these files just like you would any other file. Keep in mind that outside of development-related use, you won't usually push and pull files. Rather, you'll create and read files from within the application or work with files included with an application as a raw resource, as you'll see next.

5.2.3 Files as raw resources

If you want to include raw files with your application, you can do so using the res/raw resources location. We discussed resources in general in chapter 3. When you place a file in the res/raw location, it's not compiled by the platform, but is available as a *raw* resource, as shown in the following listing.

Listing 5.6 Accessing a noncompiled raw file from res/raw

```
public class ReadRawResourceFile extends Activity {
    private TextView readOutput;
    private Button gotoReadXMLResource;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_rawresource_file);
        readOutput =
          (TextView) findViewById(R.id.readrawres_output);
        Resources resources = getResources();
        InputStream is = null;
        try {
            is = resources.openRawResource(R.raw.people);
            byte[] reader = new byte[is.available()];
            while (is.read(reader) != -1) {}
            readOutput.setText(new String(reader));
        } catch (IOException e) {
```

1 Hold raw resource with InputStream

2 Use getResources().openRawResource()

```
            Log.e("ReadRawResourceFile", e.getMessage(), e);
        } finally {
            if (is != null) {
                try {
                    is.close();
                } catch (IOException e) {
                    // swallow
                }
            }
        }
        . . . go to next Activity via startActivity omitted for brevity
    }
}
```

Accessing raw resources closely resembles accessing files. You open a handle to an `InputStream` ❶. You call `Context.getResources()` to get the `Resources` for your current application's context, and then call `openRawResource(int id)` to link to the particular item you want ❷. Android will automatically generate the ID within the `R` class if you place your asset in the res/raw directory. You can use any file as a raw resource, including text, images, documents, or videos. The platform doesn't precompile raw resources.

The last type of file resource we need to discuss is the res/xml type, which the platform compiles into an efficient binary type accessed in a special manner.

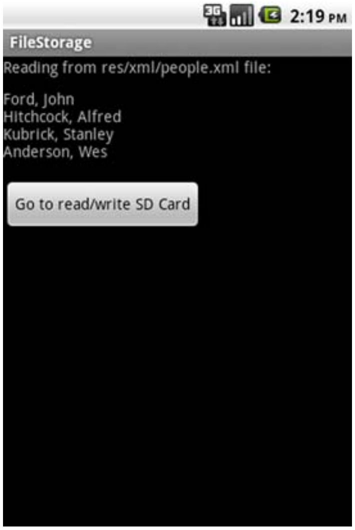
5.2.4 XML file resources

The term *XML resources* sometimes confuses new Android developers. XML resources might mean resources in general that are defined in XML—such as layout files, styles, arrays, and the like—or it can specifically mean res/xml XML files.

In this section, we'll deal with res/xml XML files. These files are different from raw files in that you don't use a stream to access them because they're compiled into an efficient binary form when deployed. They're different from other resources in that they can be of any custom XML structure.

To demonstrate this concept, we're going to use an XML file named people.xml that defines multiple <person> elements and uses attributes for `firstname` and `lastname`. We'll grab this resource and display its elements in *last-name, first-name* order, as shown in figure 5.3.

Our data file for this process, which we'll place in res/xml, is shown in the following listing.



Reading from res/xml/people.xml file:

Ford, John
Hitchcock, Alfred
Kubrick, Stanley
Anderson, Wes

Figure 5.3 The example `ReadXMLResourceFile` `Activity` that we'll create in listing 5.8, which reads a res/xml resource file

Listing 5.7 A custom XML file included in res/xml

```
<people>
     <person firstname="John" lastname="Ford" />
     <person firstname="Alfred" lastname="Hitchcock" />
     <person firstname="Stanley" lastname="Kubrick" />
     <person firstname="Wes" lastname="Anderson" />
</people>
```

If you're using Eclipse, it'll automatically detect a file in the res/xml path and compile it into a resource asset. You can then access this asset in code by parsing its binary XML, as shown in the following listing.

Listing 5.8 Accessing a compiled XML resource from res/xml

```
public class ReadXMLResourceFile extends Activity {
    private TextView readOutput;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_xmlresource_file);
        readOutput = (TextView)
          findViewById(R.id.readxmlres_output);
        XmlPullParser parser =
          getResources().getXml(R.xml.people);
        StringBuffer sb = new StringBuffer();
        try {
            while (parser.next() != XmlPullParser.END_DOCUMENT) {
                String name = parser.getName();
                String first = null;
                String last = null;
                if ((name != null) && name.equals("person")) {
                    int size = parser.getAttributeCount();
                    for (int i = 0; i < size; i++) {
                        String attrName =
                          parser.getAttributeName(i);
                        String attrValue =
                          parser.getAttributeValue(i);
                        if ((attrName != null)
                          && attrName.equals("firstname")) {
                            first = attrValue;
                        } else if ((attrName != null)
                          && attrName.equals("lastname")) {
                            last = attrValue;
                        }
                    }
                    if ((first != null) && (last != null)) {
                        sb.append(last + ", " + first + "\n");
                    }
                }
            }
          readOutput.setText(sb.toString());
        } catch (Exception e) {
            Log.e(“ReadXMLResourceFile”, e.getMessage(), e);
        }
```

1 Parse XML with XMLPullParser

2 Walk XML tree

3 Get attributeCount for element

4 Get attribute name and value

```
            . . . goto next Activity via startActivity omitted for brevity
        }
}
```

To process a binary XML resource, you use an `XmlPullParser` ❶. This class supports SAX-style tree traversal. The parser provides an event type for each element it encounters, such as `DOCDECL`, `COMMENT`, `START_DOCUMENT`, `START_TAG`, `END_TAG`, `END_DOCUMENT`, and so on. By using the `next()` method, you can retrieve the current event type value and compare it to event constants in the class ❷. Each element encountered has a name, a text value, and an optional set of attributes. You can examine the document contents by getting the `attributeCount` ❸ for each item and grabbing each name and value ❹. SAX is covered in more detail in chapter 13.

In addition to local file storage on the device filesystem, you have another option that's more appropriate for certain types of content: writing to an external SD card filesystem.

5.2.5 External storage via an SD card

One of the advantages the Android platform provides over some other smartphones is that it offers access to an available SD flash memory card. Not every Android device will necessarily have an SD card, but almost all do, and the platform provides an easy way for you to use it.

SD cards and the emulator

To work with an SD card image in the Android emulator, you'll first need to use the mksdcard tool provided to set up your SD image file (you'll find this executable in the tools directory of the SDK). After you've created the file, you'll need to start the emulator with the `-sdcard <path_to_file>` option in order to have the SD image mounted. Alternately, use the Android SDK Manager to create a new virtual device and select the option to create a new SD card.

Generally, you should use the SD card if you use large files such as images and video, or if you don't need to have permanent secure access to certain files. On the other hand, for permanent application-specialized data, you should use the internal filesystem.

The SD card is removable, and SD card support on most devices (including Android-powered devices) supports the File Allocation Table (FAT)) filesystem. The SD card doesn't have the access modes and permissions that come from the Linux filesystem.

Using the SD card is fairly basic. The standard `java.io.File` and related objects can create, read, and remove files on the external storage path, typically /sdcard, assuming it's available. You can acquire a `File` for this location by using the method `Environment.getExternalStorageDirectory()`. The following listing shows how to check that the SD card's path is present, create another subdirectory inside, and then write and subsequently read file data at that location.

Listing 5.9 Using standard `java.io.File` techniques with an SD card

```
public class ReadWriteSDCardFile extends Activity {
    private TextView readOutput;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.read_write_sdcard_file);
        readOutput = (TextView)
           findViewById(R.id.readwritesd_output);
        String fileName = "testfile-"
          + System.currentTimeMillis() + ".txt";                        ❶ Establish filename
        File sdDir = Environment.getExternalStorageDirectory();         ❷ reference
        if (sdDir.exists() && sdDir.canWrite()) {
            File uadDir = new File(sdDir.getAbsolutePath()
              + "/unlocking_android");                                  ❸ Instantiate File for path
            uadDir.mkdir();
            if (uadDir.exists() && uadDir.canWrite()) {
                File file = new File(uadDir.getAbsolutePath()
                  + "/" + fileName);                                    ❹ Get reference to File
                try {
                    file.createNewFile();
                } catch (IOException e) {
                    // log and or handle
                }
                if (file.exists() && file.canWrite()) {
                    FileOutputStream fos = null;
                    try {
                        fos = new FileOutputStream(file);
                        fos.write("I fear you speak upon the rack,"
                           + "where men enforced do speak "
                           + "anything.".getBytes());                    ❺ Write with FileOutputStream
                    } 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 and or handle - error writing to file
                }
            } else {
                // log and or handle -
                // unable to write to /sdcard/unlocking_android
            }
        } else {
            Log.e("ReadWriteSDCardFile.LOGTAG",
              "ERROR /sdcard path not available (did you create "
                     + " an SD image with the mksdcard tool,"
                     + " and start emulator with -sdcard "
                     + <path_to_file> option?");
        }
        File rFile =
          new File("/sdcard/unlocking_android/" + fileName);         ❻
        if (rFile.exists() && rFile.canRead()) {
            FileInputStream fis = null;
            try {
                fis = new FileInputStream(rFile);
                byte[] reader = new byte[fis.available()];   Read with
                while (fis.read(reader) != -1) {           ❼ FileInputStream
                }
                readOutput.setText(new String(reader));
            } catch (IOException e) {
                // log and or handle
            } finally {
                if (fis != null) {
                    try {
                        fis.close();
                    } catch (IOException e) {
                        // swallow
                    }
                }
            }
        } else {
            readOutput.setText(
              "Unable to read/write sdcard file, see logcat output");
        }
    }
}
```

We first define a name for the file to create ❶. In this example, we append a timestamp to create a unique name each time this example application runs. After we have the filename, we create a `File` object reference to the removable storage directory ❷. From there, we create a `File` reference to a new subdirectory, /sdcard/unlocking_android ❸. The `File` object can represent both files and directories. After we have the subdirectory reference, we call `mkdir()` to create it if it doesn't already exist.

With our directory structure in place, we follow a similar pattern to create the actual file. We instantiate a reference `File` object ❹, and then call `createFile()` to create a file on the filesystem. When we have the `File` and know it exists and that we're allowed to write to it, we use a `FileOutputStream` to write data into the file ❺.

After we create the file and have data in it, we create another `File` object with the full path to read the data back ❻. With the `File` reference, we then create a `FileInputStream` and read back the data that was earlier stored in the file ❼.

As you can see, working with files on the SD card resembles standard `java.io.File` code. A fair amount of boilerplate Java code is required to make a robust solution, with permissions and error checking every step of the way, and logging about what's happening, but it's still familiar and powerful. If you need to do a lot of `File` handling, you'll probably want to create some simple local utilities for wrapping the mundane tasks so you don't have to repeat them over and over again. You might want to use or port something like the Apache `commons.io` package, which includes a `FileUtils` class that handles these types of tasks and more.

The SD card example completes our exploration of the various ways to store different types of file data on the Android platform. If you have static predefined data, you can use res/raw; if you have XML files, you can use res/xml. You can also work directly with the filesystem by creating, modifying, and retrieving data in files, either in the local internal filesystem or on the SD card, if one is available.

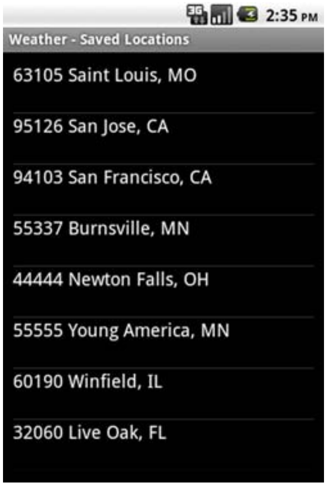
A more complex way to deal with data—one that supports more robust and specialized ways to persist information—is to use a database, which we'll cover in the next section.

5.3 Persisting data to a database

Android conveniently includes a built-in relational database.[1] SQLite doesn't have all the features of larger client/server database products, but it includes everything you need for local data storage. At the same time, it's quick and relatively easy to work with.

In this section, we'll cover working with the built-in SQLite database system, from creating and querying a database to upgrading and working with the sqlite3 tool available in the adb shell. We'll demonstrate these features by expanding the WeatherReporter application from chapter 4. This application uses a database to store the user's saved locations and persists user preferences for each location. The screenshot shown in figure 5.4 displays the saved data that the user can select from; when the user selects a location, the app retrieves information from the database and shows the corresponding weather report.

We'll start by creating WeatherReporter's database.



2:35 PM

Weather - Saved Locations

63105 Saint Louis, MO

95126 San Jose, CA

94103 San Francisco, CA

55337 Burnsville, MN

44444 Newton Falls, OH

55555 Young America, MN

60190 Winfield, IL

32060 Live Oak, FL

Figure 5.4 The WeatherReporter Saved Locations screen, which pulls data from a SQLite database

5.3.1 Building and accessing a database

To use SQLite, you have to know a bit about SQL in general. If you need to brush up on the background of the basic commands, such as `CREATE`, `INSERT`, `UPDATE`, `DELETE`, and `SELECT`, then you might want to take a look at the SQLite documentation at http://www.sqlite.org/lang.html.

[1] Check out Charlie Collins' site for Android SQLLite basics: http://www.screaming-penguin.com/node/7742.

For now, we'll jump right in and build a database helper class for our application. You need to create a helper class so that the details concerning creating and upgrading the database, opening and closing connections, and running through specific queries are all encapsulated in one place and not otherwise exposed or repeated in your application code. Your `Activity` and `Service` classes can use simple `get` and `insert` methods, with specific bean objects representing your model, rather than database-specific abstractions such as the Android `Cursor` object. You can think of this class as a miniature Data Access Layer (DAL).

The following listing shows the first part of our `DBHelper` class, which includes a few useful inner classes.

Listing 5.10 Portion of the `DBHelper` class showing the `DBOpenHelper` inner class

```
public class DBHelper {
    public static final String DEVICE_ALERT_ENABLED_ZIP = "DAEZ99";
    public static final String DB_NAME = "w_alert";
    public static final String DB_TABLE = "w_alert_loc";
    public static final int DB_VERSION = 3;
    private static final String CLASSNAME = DBHelper.class.getSimpleName();
    private static final String[] COLS = new String[]
      { "_id", "zip", "city", "region", "lastalert", "alertenabled" };
    private SQLiteDatabase db;
    private final DBOpenHelper dbOpenHelper;
    public static class Location {
        public long id;
        public long lastalert;
        public int alertenabled;
        public String zip;
        public String city;
        public String region;

        . . . Location constructors and toString omitted for brevity
    }
        private static class DBOpenHelper extends
        SQLiteOpenHelper {

        private static final String DB_CREATE = "CREATE TABLE "
                + DBHelper.DB_TABLE
                + " (_id INTEGER PRIMARY KEY, zip TEXT UNIQUE NOT NULL,"
                + "city TEXT, region TEXT, lastalert INTEGER, "
                + "alertenabled INTEGER);";

        public DBOpenHelper(Context context, String dbName, int version) {
            super(context, DBHelper.DB_NAME, null, DBHelper.DB_VERSION);
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
            try {
                db.execSQL(DBOpenHelper.DB_CREATE);
            } catch (SQLException e) {
                Log.e("ProviderWidgets", DBHelper.CLASSNAME, e);
            }
        }
```

Define constants for database properties 1

2 **Define inner Location bean**

3 **Define inner DBOpenHelper class**

4

5 **Override helper callbacks**

```
        @Override
        public void onOpen(SQLiteDatabase db) {
            super.onOpen(db);
        }

        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion,
            int newVersion) {
            db.execSQL("DROP TABLE IF EXISTS " + DBHelper.DB_TABLE);
            onCreate(db);
        }
    }
```

Within our DBHelper class, we first create constants that define important values for the database we want to work with, such as its name, version, and table ❶. Then we show several inner classes that we created to support the WeatherReporter application.

The first inner class is a simple Location bean that represents a user's selected location ❷. This class intentionally doesn't provide accessors and mutators, because these add overhead and we don't expose the class externally. The second inner class is a SQLiteOpenHelper implementation ❸.

Our DBOpenHelper inner class extends SQLiteOpenHelper, which Android provides to help with creating, upgrading, and opening databases. Within this class, we include a String that represents the CREATE query we'll use to build our database table; this shows the exact columns and types our table will have ❹. We also implement several key SQLiteOpenHelper callback methods ❺, notably onCreate and onUpgrade. We'll explain how these callbacks are invoked in the outer part of our DBHelper class, which is shown in the following listing.

Listing 5.11 Portion of the `DBHelper` class showing convenience methods

```
    public DBHelper(Context context) {
        dbOpenHelper = new DBOpenHelper(context, "WR_DATA", 1);
        establishDb();
    }
    private void establishDb() {
        if (db == null) {
            db = dbOpenHelper.getWritableDatabase();
        }
    }
    public void cleanup() {
        if (db != null) {
            db.close();
            db = null;
        }
    }
    public void insert(Location location) {
        ContentValues values = new ContentValues();
        values.put("zip", location.zip);
        values.put("city", location.city);
        values.put("region", location.region);
        values.put("lastalert", location.lastalert);
        values.put("alertenabled", location.alertenabled);
```

instance ❶

❷ Open database connection

❸ Tear down database connection

❹ Provide convenience insert, update, delete, get

```
        db.insert(DBHelper.DB_TABLE, null, values);
    }
```

4 Provide convenience insert, update, delete, get

```
    public void update(Location location) {
        ContentValues values = new ContentValues();
        values.put("zip", location.zip);
        values.put("city", location.city);
        values.put("region", location.region);
        values.put("lastalert", location.lastalert);
        values.put("alertenabled", location.alertenabled);
        db.update(DBHelper.DB_TABLE, values,
          "_id=" + location.id, null);
    }
    public void delete(long id) {
        db.delete(DBHelper.DB_TABLE, "_id=" + id, null);
    }
    public void delete(String zip) {
        db.delete(DBHelper.DB_TABLE, "zip='" + zip + "'", null);
    }
    public Location get(String zip) {
        Cursor c = null;
        Location location = null;
        try {
            c = db.query(true, DBHelper.DB_TABLE, DBHelper.COLS,
                 "zip = '" + zip + "'", null, null, null, null,
                    null);
            if (c.getCount() > 0) {
                c.moveToFirst();
                location = new Location();
                location.id = c.getLong(0);
                location.zip = c.getString(1);
                location.city = c.getString(2);
                location.region = c.getString(3);
                location.lastalert = c.getLong(4);
                location.alertenabled = c.getInt(5);
            }
        } catch (SQLException e) {
            Log.v("ProviderWidgets", DBHelper.CLASSNAME, e);
        } finally {
            if (c != null && !c.isClosed()) {
                c.close();
            }
        }
        return location;
    }
```

5 Provide additional get methods

```
    public List<Location> getAll() {
        ArrayList<Location> ret = new ArrayList<Location>();
        Cursor c = null;
        try {
            c = 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);
                }
                c.moveToNext();
            }
        } catch (SQLException e) {
            Log.v("ProviderWidgets", DBHelper.CLASSNAME, e);
        } finally {
            if (c != null && !c.isClosed()) {
                c.close();
            }
        }
        return ret;
    }
    . . . getAllAlertEnabled omitted for brevity
}
```

Our DBHelper class contains a member-level variable reference to a SQLiteDatabase object, as you saw in listing 5.10. We use this object as a workhorse to open database connections, to execute SQL statements, and more.

In the constructor, we instantiate the DBOpenHelper inner class from the first part of the DBHelper class listing ❶. Inside the establishDb method, we use dbOpenHelper to call openDatabase with the current Context, database name, and database version ❷. db is established as an instance of SQLiteDatabase through DBOpenHelper.

Although you can also just open a database connection directly on your own, using the open helper in this way invokes the provided callbacks and makes the process easier. With this technique, when you try to open your database connection, it's automatically created, upgraded, or just returned, through your DBOpenHelper. Though using a DBOpenHelper requires a few extra steps up front, it's extremely handy when you need to modify your table structure. You can simply increment the database's version number and take appropriate action in the onUpgrade callback.

Callers can invoke the cleanup method ❸ when they pause, in order to close connections and free up resources.

After the cleanup method, we include the raw SQL convenience methods that encapsulate our helper's operations. In this class, we have methods to insert, update, delete, and get data ❹. We also have a few additional specialized get and getAll methods ❺. Within these methods, you can see how to use the db object to run queries. The SQLiteDatabase class itself has many convenience methods, such as insert, update, and delete, and it provides direct query access that returns a Cursor over a result set.

You can usually get a lot of mileage and utility from basic uses of the SQLiteDatabase class. The final aspect for us to explore is the sqlite3 tool, which you can use to manipulate data outside your application.

Databases are application private

Unlike the SharedPreferences you saw earlier, you can't make a database WORLD_READABLE. Each database is accessible only by the package in which it was created. If you need to pass data across processes, you can use AIDL/Binder (as in chapter 4) or create a ContentProvider (as we'll discuss in section 5.4), but you can't use a database directly across the process/package boundary.

5.3.2 Using the sqlite3 tool

When you create a database for an application in Android, it creates files for that database on the device in the /data/data/[PACKAGE_NAME]/database/db.name location. These files are SQLite proprietary, but you can manipulate, dump, restore, and work with your databases through these files in the adb shell by using the sqlite3 tool.

> **DATA PERMISSIONS** Most devices lock down the data directory and will not allow you to browse their content using standalone tools. Use sqlite3 in the emulator or on a phone with firmware that allows you to access the /data/data directory.

You can access this tool by issuing the following commands on the command line. Remember to use your own package name; here we use the package name for the WeatherReporter sample application:

```
cd [ANDROID_HOME]/tools
adb shell
sqlite3 /data/data/com.msi.manning.chapter4/databases/w_alert.db
```

When you're in the shell and see the # prompt, you can then issue sqlite3 commands. Type .help to get started; if you need more help, see the tool's documentation at http://www.sqlite.org/sqlite.html. Using the tool, you can issue basic commands, such as SELECT or INSERT, or you can go further and CREATE or ALTER tables. Use this tool to explore, troubleshoot, and to .dump and .load data. As with many command-line SQL tools, it takes some time to get used to the format, but it's the best way to back up or load your data. Keep in mind that this tool is available only through the development shell; it's not something you can use to load a real application with data.

Now that we've shown you how to use the SQLite support provided in Android, you can do everything from creating and accessing tables to investigating databases with the provided tools in the shell. Now we'll examine the last aspect of handling data on the platform, building and using a ContentProvider.

5.4 Working with ContentProvider classes

A ContentProvider in Android shares data between applications. Each application usually runs in its own process. By default, applications can't access the data and files of other applications. We explained earlier that you can make preferences and files available across application boundaries with the correct permissions and if each application knows the context and path. This solution applies only to related applications that already know details about one another. In contrast, with a ContentProvider you can publish and expose a particular data type for other applications to query, add, update, and delete, and those applications don't need to have any prior knowledge of paths, resources, or who provides the content.

The canonical ContentProvider in Android is the contacts list, which provides names, addresses, and phone numbers. You can access this data from any application by using the correct URI and a series of methods provided by the Activity and ContentResolver classes to retrieve and store data. You'll learn more about ContentResolver as we explore provider details. One other data-related concept that a ContentProvider offers is the Cursor, the same object we used previously to process SQLite database result sets.

In this section, you'll build another application that implements its own ContentProvider and includes a similar explorer-type Activity to manipulate that data.

NOTE For a review of content providers, please see chapter 1. You can also find a complete example of working with the Contacts content provider in chapter 15.

To begin, we'll explore the syntax of URIs and the combinations and paths used to perform different types of operations with the ContentProvider and ContentResolver classes.

5.4.1 Using an existing ContentProvider

Each ContentProvider exposes a unique CONTENT_URI that identifies the content type it'll handle. This URI can query data in two forms, singular or plural, as shown in table 5.1.

A provider can offer as many types of data as it likes. By using these formats, your application can either iterate through all the content offered by a provider or retrieve a specific datum of interest.

The Activity class has a managedQuery method that makes calls into registered ContentProvider classes. When you create your own ContentProvider in section 5.4.2, we'll show you how a provider is registered with the platform. Each provider is required to advertise the CONTENT_URI it supports. To query the contacts provider, you

Table 5.1 ContentProvider URI variations for different purposes

URI	Purpose
content://food/ingredients/	Return List of all ingredients from the provider registered to handle content://food
content://food/meals/	Return List of all meals from the provider registered to handle content://food
content://food/meals/1	Return or manipulate single meal with ID 1 from the provider registered to handle content://food

Managed Cursor

To obtain a Cursor reference, you can also use the managedQuery method of the Activity class. The activity automatically cleans up any managed Cursor objects when your Activity pauses and restarts them when it starts. If you just need to retrieve data within an Activity, you'll want to use a managed Cursor, as opposed to a ContentResolver.

have to know this URI and then get a Cursor by calling managedQuery. When you have the Cursor, you can use it, as we showed you in listing 5.11.

A ContentProvider typically supplies all the details of the URI and the types it supports as constants in a class. In the android.provider package, you can find classes that correspond to built-in Android content providers, such as the MediaStore. These classes have nested inner classes that represent types of data, such as Audio and Images. Within those classes are additional inner classes, with constants that represent fields or columns of data for each type. The values you need to query and manipulate data come from the inner classes for each type.

What if the content changes after the fact?

When you use a ContentProvider to make a query, you get only the current state of the data. The data could change after your call, so how do you stay up to date? To receive notifications when a Cursor changes, you can use the ContentObserver API. ContentObserver supports a set of callbacks that trigger when data changes. The Cursor class provides register and unregister methods for ContentObserver objects.

For additional information, see the android.provider package in the Javadocs, which lists all the built-in providers. Now that we've covered a bit about using a provider, we'll look at the other side of the coin—creating a ContentProvider.

5.4.2 Creating a ContentProvider

In this section, you'll build a provider that handles data responsibilities for a generic Widget object you'll define. This simple object includes a name, type, and category; in a real application, you could represent any type of data.

To start, define a provider constants class that declares the CONTENT_URI and MIME_TYPE your provider will support. In addition, you can place the column names your provider will handle here.

DEFINING A CONTENT_URI AND MIME_TYPE

In the following listing, as a prerequisite to extending the ContentProvider class for a custom provider, we define necessary constants for our Widget type.

Listing 5.12 WidgetProvider constants, including columns and URI

```
public final class Widget implements BaseColumns {
    public static final String MIME_DIR_PREFIX =
      "vnd.android.cursor.dir";
    public static final String MIME_ITEM_PREFIX =
      "vnd.android.cursor.item";
    public static final String MIME_ITEM = "vnd.msi.widget";
    public static final String MIME_TYPE_SINGLE =
     MIME_ITEM_PREFIX + "/" + MIME_ITEM;
    public static final String MIME_TYPE_MULTIPLE =
     MIME_DIR_PREFIX + "/" + MIME_ITEM;
    public static final String AUTHORITY =
      "com.msi.manning.chapter5.Widget";
    public static final String PATH_SINGLE = "widgets/#";
    public static final String PATH_MULTIPLE = "widgets";
    public static final Uri CONTENT_URI =
      Uri.parse("content://" + AUTHORITY + "/" + PATH_MULTIPLE);
    public static final String DEFAULT_SORT_ORDER = "updated DESC";
    public static final String NAME = "name";
    public static final String TYPE = "type";
    public static final String CATEGORY = "category";
    public static final String CREATED = "created";
    public static final String UPDATED = "updated";
}
```

❶ Define authority

Define ultimate CONTENT_URI ❷

In our Widget-related provider constants class, we first extend the BaseColumns class. Now our class has a few base constants, such as _ID. Next, we define the MIME_TYPE prefix for a set of multiple items and a single item. By convention, vnd.android.cursor.dir represents multiple items, and vnd.android.cursor.item represents a single item. We can then define a specific MIME item and combine it with the single and multiple paths to create two MIME_TYPE representations.

After we have the MIME details out of the way, we define the authority ❶ and path for both single and multiple items that will be used in the CONTENT_URI that callers pass in to use our provider. Callers will ultimately start from the multiple-item URI, so we publish this one ❷.

After taking care of all the other details, we define column names that represent the variables in our Widget object, which correspond to fields in the database table we'll use. Callers will use these constants to get and set specific fields. Now we're on to the next part of the process, extending ContentProvider.

EXTENDING CONTENTPROVIDER

The following listing shows the beginning of our ContentProvider implementation class, WidgetProvider. In this part of the class, we do some housekeeping relating to the database we'll use and the URI we're supporting.

Listing 5.13 The first portion of the `WidgetProvider` `ContentProvider`

```
public class WidgetProvider extends ContentProvider {
    private static final String CLASSNAME =
        WidgetProvider.class.getSimpleName();
    private static final int WIDGETS = 1;
    private static final int WIDGET = 2;
    public static final String DB_NAME = "widgets_db";
    public static final String DB_TABLE = "widget";
    public static final int DB_VERSION = 1;
    private static UriMatcher URI_MATCHER = null;
    private static HashMap<String, String> PROJECTION_MAP;
    private SQLiteDatabase db;
    static {
        WidgetProvider.URI_MATCHER = new UriMatcher(UriMatcher.NO_MATCH);
        WidgetProvider.URI_MATCHER.addURI(Widget.AUTHORITY,
          Widget.PATH_MULTIPLE, WidgetProvider.WIDGETS);
        WidgetProvider.URI_MATCHER.addURI(Widget.AUTHORITY,
           Widget.PATH_SINGLE, WidgetProvider.WIDGET);
        WidgetProvider.PROJECTION_MAP = new HashMap<String, String>();
        WidgetProvider.PROJECTION_MAP.put(BaseColumns._ID, "_id");
        WidgetProvider.PROJECTION_MAP.put(Widget.NAME, "name");
        WidgetProvider.PROJECTION_MAP.put(Widget.TYPE, "type");
        WidgetProvider.PROJECTION_MAP.put(Widget.CATEGORY, "category");
        WidgetProvider.PROJECTION_MAP.put(Widget.CREATED, "created");
        WidgetProvider.PROJECTION_MAP.put(Widget.UPDATED, "updated");
    }
    private static class DBOpenHelper extends SQLiteOpenHelper {
        private static final String DB_CREATE = "CREATE TABLE "
                + WidgetProvider.DB_TABLE
                + " (_id INTEGER PRIMARY KEY, name TEXT UNIQUE NOT NULL,"
                   + "type TEXT, category TEXT, updated INTEGER, created"
                   + "INTEGER);";
        public DBOpenHelper(Context context) {
            super(context, WidgetProvider.DB_NAME, null,
              WidgetProvider.DB_VERSION);
        }
        @Override
        public void onCreate(SQLiteDatabase db) {
            try {
                db.execSQL(DBOpenHelper.DB_CREATE);
            } catch (SQLException e) {
                // log and or handle
            }
        }
        @Override
        public void onOpen(SQLiteDatabase db) {
        }
        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion,
          int newVersion) {
            db.execSQL("DROP TABLE IF EXISTS "
              + WidgetProvider.DB_TABLE);
            onCreate(db);
        }
```

1 Define database constants

2 Use SQLiteDatabase reference

3 Create and open database

```
    }
    @Override
    public boolean onCreate() {                                    ❹ Override onCreate
        DBOpenHelper dbHelper = new DBOpenHelper(getContext());
        db = dbHelper.getWritableDatabase();
        if (db == null) {
            return false;
        } else {
            return true;
        }
    }
    @Override
    public String getType(Uri uri) {                               ❺ Implement getType method
        switch (WidgetProvider.URI_MATCHER.match(uri)) {
        case WIDGETS:
            return Widget.MIME_TYPE_MULTIPLE;
        case WIDGET:
            return Widget.MIME_TYPE_SINGLE;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
        }
    }
```

Our provider extends `ContentProvider`, which defines the methods we'll need to implement. We use several database-related constants to define the database name and table we'll use ❶. After that, we include a `UriMatcher`, which we'll use to match types, and a projection `Map` for field names.

We include a reference to a `SQLiteDatabase` object; we'll use this to store and retrieve the data that our provider handles ❷. We create, open, or upgrade the database using a `SQLiteOpenHelper` in an inner class ❸. We've used this helper pattern before, when we worked directly with the database in listing 5.10. In the `onCreate` method, the open helper sets up the database ❹.

After our setup-related steps, we come to the first method `ContentProvider` requires us to implement, `getType` ❺. The provider uses this method to resolve each passed-in `Uri` to determine whether it's supported. If it is, the method checks which type of data the current call is requesting. The data might be a single item or the entire set.

Next, we need to cover the remaining required methods to satisfy the `ContentProvider` contract. These methods, shown in the following listing, correspond to the CRUD-related activities: `query`, `insert`, `update`, and `delete`.

Listing 5.14 The second portion of the `WidgetProvider` `ContentProvider`

```
    @Override
    public Cursor query(Uri uri, String[] projection,
      String selection, String[] selectionArgs,                      ❶
        String sortOrder) {
        SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
        String orderBy = null;
        switch (WidgetProvider.URI_MATCHER.match(uri)) {            ❷ Set up query based on URI
        case WIDGETS:
            queryBuilder.setTables(WidgetProvider.DB_TABLE);
            queryBuilder.setProjectionMap(WidgetProvider.PROJECTION_MAP);
            break;
        case WIDGET:
            queryBuilder.setTables(WidgetProvider.DB_TABLE);
            queryBuilder.appendWhere("_id="
             + uri.getPathSegments().get(1));
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
        }
        if (TextUtils.isEmpty(sortOrder)) {
            orderBy = Widget.DEFAULT_SORT_ORDER;
        } else {
            orderBy = sortOrder;
        }
        Cursor c = queryBuilder.query(db, projection,
            selection, selectionArgs, null, null,
            orderBy);
        c.setNotificationUri(
          getContext().getContentResolver(), uri);
        return c;
    }
    @Override
    public Uri insert(Uri uri, ContentValues initialValues) {
        long rowId = 0L;
        ContentValues values = null;
        if (initialValues != null) {
            values = new ContentValues(initialValues);
        } else {
            values = new ContentValues();
        }
        if (WidgetProvider.URI_MATCHER.match(uri) !=
          WidgetProvider.WIDGETS) {
            throw new IllegalArgumentException("Unknown URI " + uri);
        }
        Long now = System.currentTimeMillis();
        . . . omit defaulting of values for brevity
        rowId = db.insert(WidgetProvider.DB_TABLE, "widget_hack",
           values);
        if (rowId > 0) {
            Uri result = ContentUris.withAppendedId(Widget.CONTENT_URI,
              rowId);
            getContext().getContentResolver().
              notifyChange(result, null);
            return result;
        }
        throw new SQLException("Failed to insert row into " + uri);
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection,
         String[] selectionArgs) {
        int count = 0;
        switch (WidgetProvider.URI_MATCHER.match(uri)) {
```

3 Perform query to get Cursor

4 Set notification Uri on Cursor

5 Use ContentValues in insert method

6 Call database insert

7 Get Uri to return

8 Notify listeners data was inserted

9 Provide update method

```
        case WIDGETS:
            count = 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 = db.update(WidgetProvider.DB_TABLE, values,
                 "_id=" + segment + where, selectionArgs);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
        }
        getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }
    @Override
    public int delete(
     Uri uri, String selection, String[] selectionArgs) {
        int count;
        switch (WidgetProvider.URI_MATCHER.match(uri)) {
        case WIDGETS:
            count = 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 = db.delete(WidgetProvider.DB_TABLE,
                "_id=" + segment + where, selectionArgs);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
        }
        getContext().getContentResolver().notifyChange(uri, null);
        return count;
    }
}
```

❿ **Provide delete method**

The last part of our WidgetProvider class shows how to implement the ContentProvider methods. First, we use a SQLQueryBuilder inside the query method to append the projection map passed in ❶ and any SQL clauses, along with the correct URI based on our matcher ❷, before we make the actual query and get a handle on a Cursor to return ❸.

At the end of the query method, we use the setNotificationUri method to watch the returned Uri for changes ❹. This event-based mechanism keeps track of when Cursor data items change, regardless of who changes them.

Next, you see the insert method, where we validate the passed-in ContentValues object and populate it with default values, if the values aren't present ❺. After we have the values, we call the database insert method ❻ and get the resulting Uri to return with the appended ID of the new record ❼. After the insert is complete, we use another notification system, this time for ContentResolver. Because we've made a data change, we inform the ContentResolver what happened so that any registered listeners can be updated ❽.

After completing the insert method, we come to the update ❾ and delete ❿ methods. These methods repeat many of the previous concepts. First, they match the Uri passed in to a single element or the set, and then they call the respective update and delete methods on the database object. Again, at the end of these methods, we notify listeners that the data has changed.

Implementing the needed provider methods completes our class. After we register this provider with the platform, any application can use it to query, insert, update, or delete data. Registration occurs in the application manifest, which we'll look at next.

PROVIDER MANIFESTS

Content providers must be defined in an application manifest file and installed on the platform so the platform can learn that they're available and what data types they offer. The following listing shows the manifest for our provider.

Listing 5.15 WidgetProvider AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.chapter5.widget">
    <application android:icon="@drawable/icon"
        android:label="@string/app_short_name">
        <activity android:name=".WidgetExplorer"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name=
                  "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider android:name="WidgetProvider"
                android:authorities=
                  "com.msi.manning.chapter5.Widget" />
    </application>
</manifest>
```

❶ **Declare provider's authority**

The <provider> element ❶ defines the class that implements the provider and associates a particular authority with that class.

A completed project that supports inserting, retrieving, updating, and deleting records rounds out our exploration of using and building ContentProvider classes. And with that, we've also now demonstrated the ways to locally store and retrieve data on the Android platform.

Additional ContentProvider manifest properties

The properties of a ContentProvider can configure several important settings beyond the basics, such as specific permissions, initialization order, multiprocess capability, and more. Though most ContentProvider implementations won't need to delve into these details, you should still keep them in mind. For complete and up-to-date ContentProvider properties, see the SDK documentation.

5.5 Summary

From a simple SharedPreferences mechanism to file storage, 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 storage types can share data across application and process boundaries, and several can't. You can create SharedPreferences with a permissions mode, allowing the flexibility to keep things private, or to share data globally with read-only or read-write permissions. The filesystem provides more flexible and powerful data storage for a single application.

Android also provides a relational database system based on SQLite. Use this lightweight, speedy, and capable system for local data persistence within a single application. To share data, you can still use a database, but you need to expose an interface through a ContentProvider. Providers expose data types and operations through a URI-based approach.

In this chapter, we examined each of the data paths available to an Android application. You built several small, focused sample applications to use preferences and the filesystem, and you expanded the WeatherReporter sample application that you began in the last chapter. This Android application uses a SQLite database to access and persist data. You also built your own custom content provider from the ground up.

To expand your Android horizons beyond data, we'll move on to general networking in the next chapter. We'll cover networking basics and the networking APIs Android provides. We'll also expand on the data concepts we've covered in this chapter to use the network itself as a data source.

6 Networking and web services

This chapter covers

- Networking basics
- Determining network status
- Using the network to retrieve and store data
- Working with web services

With the ubiquity of high-speed networking, mobile devices are now expected to perform many of the same data-rich functions of traditional computers such as email, providing web access, and the like. Furthermore, because mobile phones offer such items as GPS, microphones, CDMA/GSM, built in cameras, accelerometers, and many others, user demand for applications that leverage all the features of the phone continues to increase.

You can build interesting applications with the open `Intent`- and `Service`-based approach you learned about in previous chapters. That approach combines built-in (or custom) intents, such as fully capable web browsing, with access to hardware components, such as a 3D graphics subsystem, a GPS receiver, a camera, removable storage, and more. This combination of open platform, hardware capability, software architecture, and access to network data makes Android compelling.

This doesn't mean that the voice network isn't important—we'll cover telephony explicitly in chapter 7—but we admit that voice is a commodity—and data is what we'll focus on when talking about the network.

Android provides access to networking in several ways, including mobile *Internet Protocol* (IP), Wi-Fi, and Bluetooth. It also provides some open and closed source third-party implementations of other networking standards such as ZigBee and Worldwide Interoperability for Microwave Access (WiMAX). In this chapter, though, we'll concentrate on getting your Android applications to communicate using IP network data, using several different approaches. We'll cover a bit of networking background, and then we'll deal with Android specifics as we explore communication with the network using sockets and higher-level protocols such as *Hypertext Transfer Protocol* (HTTP).

Android provides a portion of the `java.net` package and the `org.apache.httpclient` package to support basic networking. Other related packages, such as `android.net`, address internal networking details and general connectivity properties. You'll encounter all these packages as we progress though networking scenarios in this chapter.

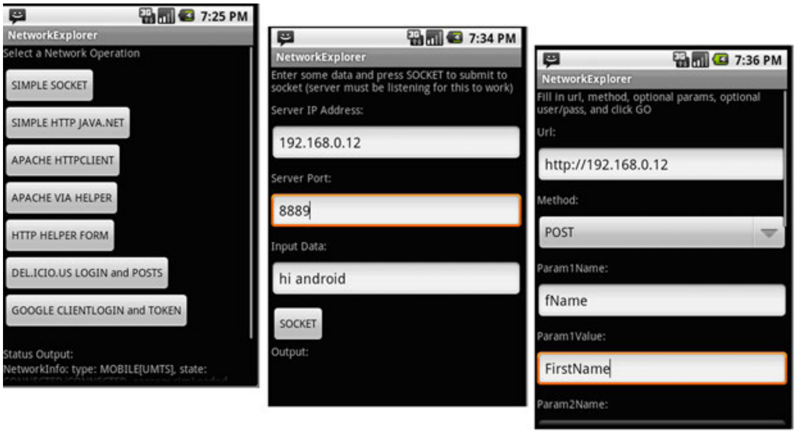
In terms of connectivity properties, we'll look at using the `ConnectivityManager` class to determine when the network connection is active and what type of connection it is: mobile or Wi-Fi. From there, we'll use the network in various ways with sample applications.

One caveat to this networking chapter is that we won't dig into the details concerning the Android Wi-Fi or Bluetooth APIs. Bluetooth is an important technology for close-range wireless networking between devices, but it isn't available in the Android emulator (see chapter 14 for more on Bluetooth). On the other hand, Wi-Fi has a good existing API but also doesn't have an emulation layer. Because the emulator doesn't distinguish the type of network you're using and doesn't know anything about either Bluetooth or Wi-Fi, and because we think the importance lies more in how you use the network, we aren't going to cover these APIs. If you want more information on the Wi-Fi APIs, please see the Android documentation (http://code.google.com/android/reference/android/net/wifi/package-summary.html).

The aptly named sample application for this chapter, NetworkExplorer, will look at ways to communicate with the network in Android and will include some handy utilities. Ultimately, this application will have multiple screens that exercise different networking techniques, as shown in figure 6.1.

After we cover general IP networking with regard to Android, we'll discuss turning the server side into a more robust API itself by using web services. On this topic, we'll work with *plain old XML over HTTP* (*POX*) and *Representational State Transfer* (REST). We'll also discuss the *Simple Object Access Protocol* (SOAP). We'll address the pros and cons of the various approaches and why you might want to choose one method over another for an Android client.

Before we delve into the details of networked Android applications, we'll begin with an overview of networking basics. If you're already well versed in general



Select a Network Operation

Status Output:
NetworkInfo: type: MOBILE[UMTS], state:

Enter some data and press SOCKET to submit to socket (server must be listening for this to work)

Server IP Address: 192.168.0.12

Server Port: 8889

Input Data: hi android

Output:

Fill in url, method, optional params, optional user/pass, and click GO

Url: http://192.168.0.12

Method: POST

Param1Name: fName

Param1Value: FirstName

Param2Name:

Figure 6.1 The NetworkExplorer application you'll build to cover networking topics

networking, you can skip ahead to section 6.2, but it's important to have this foundation if you think you need it, and we promise to keep it short.

6.1 An overview of networking

A group of interconnected computers is a *network*. Over time, networking has grown from something that was available only to governments and large organizations to the almost ubiquitous and truly amazing internet. Though the concept is simple—allow computers to communicate—networking does involve advanced technology. We won't get into great detail here, but we'll cover the core tenets as a background to the general networking you'll do in the remainder of this chapter.

6.1.1 Networking basics

A large percentage of the time, the APIs you use to program Android applications abstract the underlying network details. This is good. The APIs and the network protocols themselves are designed so that you can focus on your application and not worry about routing, reliable packet delivery, and so on.

Nevertheless, it helps to have some understanding of the way a network works so that you can better design and troubleshoot your applications. To that end, let's cover some general networking concepts, with a focus on *Transmission Control Protocol/Internet Protocol* (TCP/IP).[1] We'll begin with nodes, layers, and protocols.

NODES

The basic idea behind a network is that data is sent between connected devices using particular addresses. Connections can be made over wire, over radio waves, and so on. Each addressed device is known as a *node*. A node can be a mainframe, a PC, a fancy

[1] For an in-depth study of all things TCP/IP related, take a look at Craig Hunt's book: http://oreilly.com/catalog/9780596002978.

toaster, or any other device with a network stack and connectivity, such as an Android-enabled handheld.

LAYERS AND PROTOCOLS

Protocols are a predefined and agreed-upon set of rules for communication. Protocols are often layered on top of one another because they handle different levels of responsibility. The following list describes the main layers of the TCP/IP stack, which is used for the majority of web traffic and with Android:

- The *Link Layer*—including physical device address resolution protocols such as ARP and RARP
- The *Internet Layer*—including IP itself, which has multiple versions, the `ping` protocol, and ICMP, among others
- The *Transport Layer*—where different types of delivery protocols such as TCP and UDP are found
- The *Application Layer*—which includes familiar protocols such as HTTP, FTP, SMTP, IMAP, POP, DNS, SSH, and SOAP

Layers are an abstraction of the different levels of a network protocol stack. The lowest level, the Link Layer, is concerned with physical devices and physical addresses. The next level, the Internet Layer, is concerned with addressing and general data details. After that, the Transport Layer is concerned with delivery details. And, finally, the top-level Application Layer protocols, which make use of the stack beneath them, are application-specific for sending files or email or viewing web pages.

IP

IP is in charge of the addressing system and delivering data in small chunks called *packets*. Packets, known in IP terms as *datagrams*, define how much data can go in each chunk, where the boundaries for payload versus header information are, and the like. IP addresses tell where each packet is from (its source) and where it's going (its destination).

IP addresses come in different sizes, depending on the version of the protocol being used, but by far the most common at present is the 32-bit address. 32-bit IP addresses (TCP/IP version 4, or IPv4) are typically written using a decimal notation that separates the 32 bits into four sections, each representing 8 bits (an octet), such as 74.125.45.100.

Certain IP address classes have special roles and meaning. For example, 127 always identifies a *loopback*[2] or local address on every machine; this class doesn't communicate with any other devices (it can be used internally, on a single machine only). Addresses that begin with 10 or 192 aren't routable, meaning they can communicate with other devices on the same local network segment but can't connect to other seg-

[2] The TCP/IP Guide provides further explanation of datagrams and loopbacks: http://www.tcpipguide.com/index.htm.

ments. Every address on a particular network segment must be unique or collisions can occur and it gets ugly.

The routing of packets on an IP network—how packets traverse the network and go from one segment to another—is handled by *routers*. Routers speak to each other using IP addresses and other IP-related information.

TCP AND UDP

TCP and UDP (User Datagram Protocol) are different delivery protocols that are commonly used with TCP/IP. TCP is reliable, and UDP is fire and forget. What does that mean? It means that TCP includes extra data to guarantee the order of packets and to send back an acknowledgment when a packet is received. The common analogy is certified mail: the sender gets a receipt that shows the letter was delivered and signed for, and therefore knows the recipient got the message. UDP, on the other hand, doesn't provide any ordering or acknowledgment. It's more like a regular letter: it's cheaper and faster to send, but you basically just hope the recipient gets it.

APPLICATION PROTOCOLS

After a packet is sent and delivered, an application takes over. For example, to send an email message, Simple Mail Transfer Protocol (SMTP) defines a rigorous set of procedures that have to take place. You have to say hello in a particular way and introduce yourself; then you have to supply from and to information, followed by a message body in a particular format. Similarly, HTTP defines the set of rules for the Internet—which methods are allowed (`GET`, `POST`, `PUT`, `DELETE`) and how the overall request/response system works between a client and a server.

When you're working with Android (and Java-related APIs in general), you typically don't need to delve into the details of any of the lower-level protocols, but you might need to know the major differences we've outlined here for troubleshooting. You should also be well-versed in IP addressing, know a bit more about clients and servers, and how connections are established using ports.

6.1.2 Clients and servers

Anyone who's ever used a web browser is familiar with the client/server computing model. Data, in one format or another, is stored on a centralized, powerful server. Clients then connect to that server using a designated protocol, such as HTTP, to retrieve the data and work with it.

This pattern is, of course, much older than the web, and it has been applied to everything from completely dumb terminals that connect to mainframes to modern desktop applications that connect to a server for only a portion of their purpose. A good example is iTunes, which is primarily a media organizer and player, but also has a store where customers can connect to the server to get new content. In any case, the concept is the same: The client makes a type of request to the server and the server responds. This model is the same one that the majority of Android applications (at least those that use a server side at all) generally follow. Android applications typically end up as the client.

In order to handle many client requests that are often for different purposes and that come in nearly simultaneously to a single IP address, modern server operating systems use the concept of *ports*. Ports aren't physical; they're a representation of a particular area of the computer's memory. A server can listen on multiple designated ports at a single address; for example, one port for sending email, one port for web traffic, two ports for file transfer, and so on. Every computer with an IP address also supports a range of thousands of ports to enable multiple conversations to happen at the same time.

Ports are divided into three ranges:

- *Well-known ports*—0 through 1023
- *Registered ports*—1024 through 49151
- *Dynamic and/or private ports*—49152 through 65535

The well-known ports are all published and are just that—well known. HTTP is port 80 (and HTTP Secure, or HTTPS, is port 443), FTP is ports 20 (control) and 21 (data), SSH is port 22, SMTP is port 25, and so on.

Beyond the well-known ports, the registered ports are still controlled and published, but for more specific purposes. Often these ports are used for a particular application or company; for example, MySQL is port 3306 (by default). For a complete list of well-known and registered ports, see the Internet Corporation for Assigned Names and Numbers (ICANN) port-numbers document: http://www.iana.org/assignments/port-numbers.

The dynamic or private ports are intentionally unregistered because they're used by the TCP/IP stack to facilitate communication. These ports are dynamically registered on each computer and used in the conversation. Dynamic port 49500, for example, might be used to handle sending a request to a web server and dealing with the response. When the conversation is over, the port is reclaimed and can be reused locally for any other data transfer.

Clients and servers communicate as nodes with addresses, using ports, on a network that supports various protocols. The protocols Android uses are based on the IP network the platform is designed to participate in and involve the TCP/IP family. Before you can build a full-on client/server Android application using the network, you need to handle the prerequisite task of determining the state of the connection.

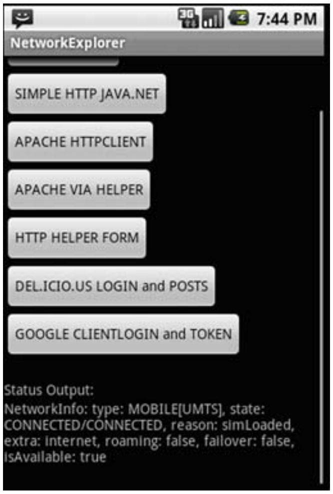
6.2 Checking the network status

Android provides a host of utilities that determine the device configuration and the status of various services, including the network. You'll typically use the `ConnectivityManager` class to determine whether network connectivity exists and to get notifications of network changes. The following listing, which is a portion of the main `Activity` in the NetworkExplorer application, demonstrates basic usage of the `ConnectivityManager`.

Listing 6.1 The `onStart` method of the NetworkExplorer main `Activity`

```
@Override
public void onStart() {
    super.onStart();
    ConnectivityManager cMgr =  (ConnectivityManager)
 this.getSystemService(Context.CONNECTIVITY_SERVICE);
    NetworkInfo netInfo = cMgr.getActiveNetworkInfo();
    this.status.setText(netInfo.toString());
}
```

This short example shows that you can get a handle to the `ConnectivityManager` through the context's `getSystemService` method by passing the `CONNECTIVITY_SERVICE` constant. When you have the manager, you can obtain network information via the `NetworkInfo` object. The `toString` method of the `NetworkInfo` object returns the output shown in figure 6.2.



```
Status Output:
NetworkInfo: type: MOBILE[UMTS], state:
CONNECTED/CONNECTED, reason: simLoaded,
extra: internet, roaming: false, failover: false,
isAvailable: true
```

Figure 6.2 The output of the `NetworkInfo` `toString` method

Of course, you won't normally just display the `String` output from `NetworkInfo`, but this example does give you a glance at what's available. More often, you'll use the `isAvailable` or `isConnected` methods (which return a `boolean` value), or you'll directly query the `NetworkInfo.State` using the `getState` method. `NetworkInfo. State` is an `enum` that defines the coarse state of the connection. The possible values are `CONNECTED`, `CONNECTING`, `DISCONNECTED`, and `DISCONNECTING`. The `NetworkInfo` object also provides access to more detailed information, but you won't normally need more than the basic state.

When you know that you're connected, either via mobile or Wi-Fi, you can use the IP network. For the purposes of our NetworkExplorer application, we're going to start with the most rudimentary IP connection, a raw socket, and work our way up to HTTP and web services.

6.3 Communicating with a server socket

A server socket is a stream that you can read or write raw bytes to, at a specified IP address and port. You can deal with data and not worry about media types, packet sizes, and so on. A server socket is yet another network abstraction intended to make the programmer's job a bit easier. The philosophy that sockets take on—that everything should look like file input/output (I/O) to the developer—comes from the Portable Operating System Interface for UNIX (POSIX) family of standards and has been adopted by most major operating systems in use today.

We'll move on to higher levels of network communication in a bit, but we'll start with a raw socket. For that, we need a server listening on a particular port. The

EchoServer code shown in the next listing fits the bill. This example isn't an Android-specific class; rather, it's an oversimplified server that can run on any host machine with Java. We'll connect to it later from an Android client.

Listing 6.2 A simple echo server for demonstrating socket usage

```
public final class EchoServer extends Thread {
    private static final int PORT = 8889;
    private EchoServer() {}
    public static void main(String args[]) {
        EchoServer echoServer = new EchoServer();
        if (echoServer != null) {
            echoServer.start();
        }
    }
    public void run() {
        try {
            ServerSocket server = new ServerSocket(PORT, 1);
            while (true) {
                Socket client = server.accept();
                System.out.println("Client connected");
                while (true) {
                    BufferedReader reader =
                        new BufferedReader(new InputStreamReader(
                        client.getInputStream()));
                    System.out.println("Read from client");
                    String textLine = reader.readLine() + "\n";
                    if (textLine.equalsIgnoreCase("EXIT\n")) {
                        System.out.println("EXIT invoked, closing client");
                        break;
                       }
                    BufferedWriter writer = new BufferedWriter(
                        new OutputStreamWriter(
                        client.getOutputStream()));
                    System.out.println("Echo input to client");
                    writer.write("ECHO from server: "
                         + textLine, 0, textLine.length() + 18);
                    writer.flush();
                   }
                client.close();
           }
        } catch (IOException e) {
            System.err.println(e);
        }
    }
}
```

Use java.net.ServerSocket ❶

Read input with BufferedReader ❷

EXIT, break the loop ❸

The EchoServer class we're using is fairly basic Java I/O. It extends Thread and implements run, so that each client that connects can be handled in its own context. Then we use a ServerSocket ❶ to listen on a defined port. Each client is then an implementation of a Socket. The client input is fed into a BufferedReader that each line is read from ❷. The only special consideration this simple server has is that if the input is EXIT, it breaks the loops and exits ❸. If the input doesn't prompt an exit, the server echoes the input back to the client's OuputStream with a BufferedWriter.

This example is a good, albeit intentionally basic, representation of what a server does. It handles input, usually in a separate thread, then responds to the client, based on the input. To try out this server before using Android, you can telnet to the specified port (after the server is running, of course) and type some input; if all is well, it will echo the output.

To run the server, you need to invoke it locally with Java. The server has a main method, so it'll run on its own; start it from the command line or from your IDE. Be aware that when you connect to a server from the emulator (this one or any other), you need to connect to the IP address of the host you run the server process on, not the loopback (not 127.0.0.1). The emulator thinks of itself as 127.0.0.1, so use the nonloopback address of the server host when you attempt to connect from Android. (You can find out the IP address of the machine you're on from the command line by entering ifconfig on Linux or Mac and ipconfig on Windows.)

The client portion of this example is where NetworkExplorer itself begins, with the callSocket method of the SimpleSocket Activity, shown in the next listing.

Listing 6.3 An Android client invoking a raw socket server resource, the echo server

```
public class SimpleSocket extends Activity {
  . . . View variable declarations omitted for brevity
  @Override
  public void onCreate(final Bundle icicle) {
    super.onCreate(icicle);
    this.setContentView(R.layout.simple_socket);
    . . . View inflation omitted for brevity
    this.socketButton.setOnClickListener(new OnClickListener() {
      public void onClick(final View v) {
        socketOutput.setText("");
        String output = callSocket(
          ipAddress.getText().toString(),
          port.getText().toString(),                   ❶ Use callSocket
        socketInput.getText().toString());               method
        socketOutput.setText(output);
      }
    });
  }
  private String callSocket(String ip, String port, String socketData) {
    Socket socket = null;
    BufferedWriter writer = null;
    BufferedReader reader = null;
    String output = null;                                  ❷ Create
    try {                                                    client
      socket = new Socket(ip, Integer.parseInt(port));       Socket
      writer = new BufferedWriter(
        new OutputStreamWriter(
          socket.getOutputStream()));

      reader = new BufferedReader(
             new InputStreamReader(
               socket.getInputStream()));
           String input = socketData;
           writer.write(input + "\n", 0, input.length() + 1);
           writer.flush();
           output = reader.readLine();
           this.socketOutput.setText(output);
           // send EXIT and close
           writer.write("EXIT\n", 0, 5);
           writer.flush();
. . . catches and reader, writer, and socket closes omitted for brevity
. . . onCreate omitted for brevity
          return output;
  }
```

❸ Write to socket

❹ Get socket output

In this listing, we use the `onCreate` method to call a private helper `callSocket` method ❶ and set the output to a `TextView`. Within the `callSocket` method, we create a `Socket` to represent the client side of our connection ❷, and we establish a writer for the input and a reader for the output. With the housekeeping taken care of, we then write to the socket ❸, which communicates with the server, and get the output value to return ❹.

A socket is probably the lowest-level networking usage in Android you'll encounter. Using a raw socket, though abstracted a great deal, still leaves many of the details up to you, especially the server-side details of threading and queuing. Although you might run up against situations in which you either have to use a raw socket (the server side is already built) or you elect to use one for one reason or another, higher-level solutions such as leveraging HTTP usually have decided advantages.

6.4 Working with HTTP

As we discussed in the previous section, you can use a raw socket to transfer IP data to and from a server with Android. This approach is an important one to be aware of so that you know you have that option and understand a bit about the underlying details. Nevertheless, you might want to avoid this technique when possible, and instead take advantage of existing server products to send your data. The most common way to do this is to use a web server and leverage HTTP.

Now we're going to take a look at making HTTP requests from an Android client and sending them to an HTTP server. We'll let the HTTP server handle all the socket details, and we'll focus on our client Android application.

The HTTP protocol itself is fairly involved. If you're unfamiliar with it or want the complete details, information is readily available via Requests for Comments (RFCs) (such as for version 1.1: http://www.w3.org/Protocols/rfc2616/rfc2616.html). The short story is that the protocol is stateless and involves several different methods that allow users to make requests to servers, and those servers return responses. The entire web is, of course, based on HTTP. Beyond the most basic concepts, there are ways to pass data into and out of requests and responses and to authenticate with servers.

Here we're going to use some of the most common methods and concepts to talk to network resources from Android applications.

To begin, we'll retrieve data using HTTP GET requests to a simple HTML page, using the standard java.net API. From there, we'll look at using the Android-included Apache HttpClient API. After we use HttpClient directly to get a feel for it, we'll also make a helper class, `HttpRequestHelper`, that you can use to simplify the process and encapsulate the details. This class—and the Apache networking API in general—has a few advantages over rolling your own networking with java.net, as you'll see. When the helper class is in place, we'll use it to make additional HTTP and HTTPS requests, both GET and POST, and we'll look at basic authentication.

Our first HTTP request will be an HTTP GET call using an `HttpUrlConnection`.

6.4.1 Simple HTTP and java.net

The most basic HTTP request method is GET. In this type of request, any data that's sent is embedded in the URL, using the query string. The next class in our Network-Explorer application, which is shown in the following listing, has an `Activity` that demonstrates the GET request method.

Listing 6.4 The `SimpleGet Activity` showing `java.net.UrlConnection`

```
public class SimpleGet extends Activity {
    . . . other portions of onCreate omitted for brevity
        this.getButton.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                getOutput.setText("");
                String output =
                    getHttpResponse(getInput.getText().toString());
                if (output != null) {
                    getOutput.setText(output);
                }
            }
        });
    };
   . . .
    private String getHttpResponse(String location) {
        String result = null;
        URL url = null;
        try {
            url = new URL(location);

        } catch (MalformedURLException e) {
            // log and or handle
        }
        if (url != null) {
            try {
                HttpURLConnection urlConn =
                     (HttpURLConnection) url.openConnection();
                BufferedReader in =
                     new BufferedReader(
                       new InputStreamReader(
```

1 Invoke getHttpResponse method

2 Construct URL object

3 Open connection using HttpURLConnection

```
            urlConn.getInputStream()));
        String inputLine;
        int lineCount = 0; // limit lines for example
        while ((lineCount < 10)
           && ((inputLine = in.readLine()) != null)) {      ❹ Read data

          lineCount++;
          result += "\n" + inputLine;      ❺ Append to result
        }
        in.close();
        urlConn.disconnect();
      } catch (IOException e) {
        // log and or handle
      }
    } else {
      // log and or handle
    }
    return result;
  }
}
```

To get an HTTP response and show the first few lines of it in our `SimpleGet` class, we call a `getHttpResponse` method that we've built ❶. Within this method, we construct a `java.net.URL` object ❷, which takes care of many of the details for us, and then we open a connection to a server using an `HttpURLConnection` ❸.

We then use a `BufferedReader` to read data from the connection one line at a time ❹. Keep in mind that as we're doing this, we're using the same thread as the UI and therefore blocking the UI. This isn't a good idea. We're using the same thread here only to demonstrate the network operation; we'll explain more about how to use a separate thread shortly. After we have the data, we append it to the result `String` that our method returns ❺, and we close the reader and the connection. Using the plain and simple `java.net` support that has been ported to Android this way provides quick and dirty access to HTTP network resources.

Communicating with HTTP this way is fairly easy, but it can quickly get cumbersome when you need to do more than just retrieve simple data, and, as noted, the blocking nature of the call is bad form. You could get around some of the problems with this approach on your own by spawning separate threads and keeping track of them and by writing your own small framework/API structure around that concept for each HTTP request, but you don't have to. Fortunately, Android provides another set of APIs in the form of the Apache HttpClient[3] library that abstract the `java.net` classes further and are designed to offer more robust HTTP support and help handle the separate-thread issue.

6.4.2 Robust HTTP with HttpClient

To get started with HttpClient, we're going to look at using core classes to perform HTTP `GET` and `POST` method requests. We're going to concentrate on making network

[3] You'll find more about the Apache HttpClient here: http://hc.apache.org/httpclient-3.x/.

requests in a Thread separate from the UI, using a combination of the Apache ResponseHandler and Android Handler (for different but related purposes, as you'll see). The following listing shows our first example of using the HttpClient API.

Listing 6.5 Apache HttpClient with Android `Handler` and Apache `ResponseHandler`

```
. . . .
private final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            progressDialog.dismiss();
            String bundleResult =
              msg.getData().getString("RESPONSE");
            output.setText(bundleResult);
        }
    };
. . . onCreate omitted for brevity
private void performRequest() {
        final ResponseHandler<String> responseHandler =
          new ResponseHandler<String>() {
            public String handleResponse(HttpResponse response) {
                StatusLine status = response.getStatusLine();
                HttpEntity entity = response.getEntity();
                String result = null;
                try {
                    result = StringUtils.inputStreamToString(
                      entity.getContent());
                    Message message = handler.obtainMessage();
                    Bundle bundle = new Bundle();
                    bundle.putString("RESPONSE", result);
                    message.setData(bundle);
                    handler.sendMessage(message);
                } catch (IOException e) {
                    // log and or handle
                }
                return result;
            }
        };
        this.progressDialog =
          ProgressDialog.show(this, "working . . .",
            "performing HTTP request");
        new Thread() {
            public void run() {
                try {
                    DefaultHttpClient client = new DefaultHttpClient();
                    HttpGet httpMethod =
                      new HttpGet(
                        urlChooser.getSelectedItem().toString());
                    client.execute(
                      httpMethod, responseHandler);
                } catch (ClientProtocolException e) {
                    // log and or handle
                } catch (IOException e) {
                    // log and or handle
```

1 Use Handler to update UI

2 Create ResponseHandler for asynchronous HTTP

3 Get HTTP response payload

Use separate Thread for HTTP call

Create HttpGet object

Execute HTTP with HttpClient

```
                }
            }
        }.start();
    }
```

The first thing we do in our initial HttpClient example is create a Handler that we can send messages to from other threads. This technique is the same one we've used in previous examples; it allows background tasks to send Message objects to hook back into the main UI thread ❶. After we create an Android Handler, we create an Apache ResponseHandler ❷. This class can be used with HttpClient HTTP requests to pass in as a callback point. When an HTTP request that's fired by HttpClient completes, it calls the onResponse method if a ResponseHandler is used. When the response comes in, we get the payload using the HttpEntity the API returns ❸. 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.

Now that you've seen HttpClient at work and understand the basic approach, the next thing we'll do is encapsulate a few of the details into a convenient helper class so that we can call it over and over without having to repeat a lot of the setup.

6.4.3 Creating an HTTP and HTTPS helper

The next Activity in our NetworkExplorer application, which is shown in listing 6.6, is a lot more straightforward and Android-focused than our other HTTP-related classes up to this point. We've used the helper class we mentioned previously, which

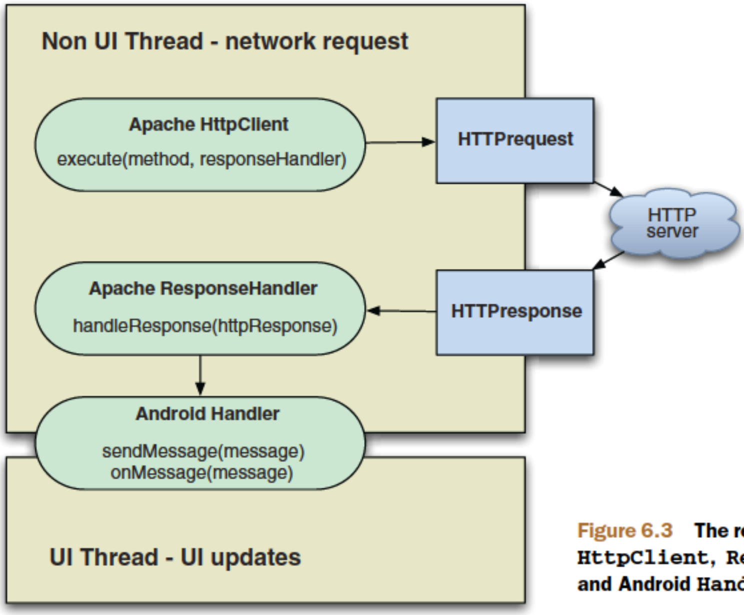


Figure 6.3 The relationship between HttpClient, ResponseHandler, and Android Handler

hides some of the complexity. We'll examine the helper class itself after we look at this first class that uses it.

Listing 6.6 Using Apache HttpClient via a custom `HttpRequestHelper`

```
public class ApacheHTTPViaHelper extends Activity {
    . . . other member variables omitted for brevity
    private final Handler handler = new Handler() {
        public void handleMessage(Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            output.setText(bundleResult);

        }
    };
    @Override
    public void onCreate(final Bundle icicle) {
        super.onCreate(icicle);
        . . . view inflation and setup omitted for brevity
        this.button.setOnClickListener(new OnClickListener() {
            public void onClick(final View v) {
                output.setText("");
                performRequest(
                  urlChooser.getSelectedItem().toString());     ❶ Call local performRequest
            }
        });
    };
    . . . onPause omitted for brevity
    private void performRequest(String url) {
        final ResponseHandler<String> responseHandler =
          HTTPRequestHelper.getResponseHandlerInstance(
            this.handler);                                      ❷ Get ResponseHandler from RequestHelper

        this.progressDialog =
          ProgressDialog.show(this, "working . . .",
            "performing HTTP request");
        new Thread() {
            public void run() {
                HTTPRequestHelper helper = new
                HTTPRequestHelper(responseHandler);             ❸ Instantiate RequestHelper with ResponseHandler
                helper.performGet(url, null, null, null);
            }
        }.start();
    }
}
```

The first thing we do in this class is create another `Handler`. From within it, we update a UI `TextView` based on data in the `Message`. Further on in the code, in the `onCreate` method, we call a local `performRequest` method when the Go button is clicked, and we pass a selected `String` representing a URL ❶.

Inside the `performRequest` method, we use a static convenience method to return an `HttpClient ResponseHandler`, passing in the Android `Handler` that it'll use ❷. We'll examine the helper class next to get a look at exactly how this works, but the important part for now is that the `ResponseHandler` is created for us by the static method. With the ResponseHandler instance taken care of, we instantiate an HttpRequestHelper instance 3 and use it to make a simple HTTP GET call (passing in only the String URL). Similar to what happened in listing 6.5, when the request completes, the ResponseHandler fires the onResponse method, and our Handler is sent a Message, completing the process.

The example Activity in listing 6.6 is fairly clean and simple, and it's asynchronous and doesn't block the UI thread. The heavy lifting is taken care of by HttpClient itself and by the setup our custom HttpRequestHelper makes possible. The first part of the all-important HttpRequestHelper, which we'll explore in three listings (listing 6.7, 6.8, and 6.9), is shown in the following listing.

Listing 6.7 The first part of the HttpRequestHelper class

```
public class HTTPRequestHelper {
    private static final int POST_TYPE = 1;
    private static final int GET_TYPE = 2;
    private static final String CONTENT_TYPE = "Content-Type";       Require
    public static final String MIME_FORM_ENCODED =              ResponseHandler
      "application/x-www-form-urlencoded";                         to construct  1
    public static final String MIME_TEXT_PLAIN = "text/plain";
    private final ResponseHandler<String> responseHandler;
    public HTTPRequestHelper(ResponseHandler<String> responseHandler) {
        this.responseHandler = responseHandler;
    }
    public void performGet(String url, String user, String pass,
        final Map<String, String> additionalHeaders) {                   Provide
        performRequest(null, url, user, pass,                            simple
             additionalHeaders, null, HTTPRequestHelper.GET_TYPE);       GET
    }                                                                 2  method
    public void performPost(String contentType, String url,
      String user, String pass,
      Map<String, String> additionalHeaders,              3 Provide simple
      Map<String, String> params) {                         POST methods

        performRequest(contentType, url, user, pass,
          additionalHeaders, params, HTTPRequestHelper.POST_TYPE);
    }
    public void performPost(String url, String user, String pass,
      Map<String, String> additionalHeaders,
      Map<String, String> params) {
        performRequest(HTTPRequestHelper.MIME_FORM_ENCODED,
          url, user, pass,
            additionalHeaders, params, HTTPRequestHelper.POST_TYPE);
    }
    private void performRequest(
        String contentType,
        String url,
        String user,
        String pass,
        Map<String, String> headers,                           4 Handle
        Map<String, String> params,                              combinations in
        int requestType) {                                       private method
    DefaultHttpClient client = new DefaultHttpClient();
    if ((user != null) && (pass != null)) {
      client.getCredentialsProvider().setCredentials(
       AuthScope.ANY,
      new UsernamePasswordCredentials(user, pass));
    }
    final Map<String, String> sendHeaders =
     new HashMap<String, String>();
    if ((headers != null) && (headers.size() > 0)) {
      sendHeaders.putAll(headers);
    }
    if (requestType == HTTPRequestHelper.POST_TYPE) {
      sendHeaders.put(HTTPRequestHelper.CONTENT_TYPE, contentType);
    }
    if (sendHeaders.size() > 0) {                        ❺ Use Interceptor
      client.addRequestInterceptor(                        for request
       new HttpRequestInterceptor() {                      headers

        public void process(
         final HttpRequest request, final HttpContext context)
          throws HttpException, IOException {
            for (String key : sendHeaders.keySet()) {
              if (!request.containsHeader(key)) {
                request.addHeader(key,
                 sendHeaders.get(key));
              }
            }
        }
      });
    }
    . . . POST and GET execution in listing 6.8
  }
```

The first thing of note in the HttpRequestHelper class is that a ResponseHandler is required to be passed in as part of the constructor ❶. This ResponseHandler will be used when the HttpClient request is ultimately invoked. After the constructor, we see a public HTTP GET-related method ❷ and several different public HTTP POST-related methods ❸. Each of these methods is a wrapper around the private performRequest method that can handle all the HTTP options ❹. The performRequest method supports a content-type header value, URL, username, password, Map of additional headers, similar Map of request parameters, and request method type.

Inside the performRequest method, a DefaultHttpClient is instantiated. Next, we check whether the user and pass method parameters are present; if they are, we set the request credentials with a UsernamePasswordCredentials type (HttpClient supports several types of credentials; see the Javadocs for details). At the same time as we set the credentials, we also set an AuthScope. The scope represents which server, port, authentication realm, and authentication scheme the supplied credentials are applicable for.

You can set any of the HttpClient parameters as finely or coarsely grained as you want; we're using the default ANY scope that matches anything. What we notably haven't set in all of this is the specific authentication scheme to use. HttpClient supports various schemes, including basic authentication, digest authentication, and a Windows-specific NT Lan Manager (NTLM) scheme. Basic authentication (simple username/password challenge from the server) is the default. Also, if you need to, you can use a preemptive form login for form-based authentication—submit the form you need, get the token or session ID, and set default credentials.

After the security is out of the way, we use an HttpRequestInterceptor to add HTTP headers ❺. Headers are name/value pairs, so adding the headers is pretty easy. After we have all of the properties that apply regardless of our request method type, we then add additional settings that are specific to the method. The following listing, the second part of our helper class, shows the POST- and GET-specific settings and the execute method.

Listing 6.8 The second part of the HttpRequestHelper class

```
     . . .
     if (requestType == HTTPRequestHelper.POST_TYPE) {
        HttpPost method = new HttpPost(url);

        List<NameValuePair> nvps = null;
        if ((params != null) && (params.size() > 0)) {
            nvps = new ArrayList<NameValuePair>();
            for (String key : params.keySet()) {
                nvps.add(new BasicNameValuePair(key,
                  params.get(key)));

            }
        }
        if (nvps != null) {
            try {
                method.setEntity(
                  new UrlEncodedFormEntity(nvps, HTTP.UTF_8));
            } catch (UnsupportedEncodingException e) {
                // log and or handle
            }
        }
     execute(client, method);
    } else if (requestType == HTTPRequestHelper.GET_TYPE) {
        HttpGet method = new HttpGet(url);
        execute(client, method);
  }
. . .
private void execute(HttpClient client, HttpRequestBase method) {
    BasicHttpResponse errorResponse =
        new BasicHttpResponse(
          new ProtocolVersion("HTTP_ERROR", 1, 1),
            500, "ERROR");

    try {
        client.execute(method, this.responseHandler);
    } catch (Exception e) {
        errorResponse.setReasonPhrase(e.getMessage());
```

❶ Create HttpPost object

❷ Add name/value parameters

❸ Call execute method

❹ Set up an error handler

```
            try {
                this.responseHandler.handleResponse(errorResponse);
            } catch (Exception ex) {
                // log and or handle
            }
        }
    }
```

When the specified request is a POST type, we create an HttpPost object to deal with it ❶. Then we add POST request parameters, which are another set of name/value pairs and are built with the BasicNameValuePair object ❷. After adding the parameters, we're ready to perform the request, which we do with our local private execute method using the method object and the client ❸.

Our execute method sets up an error response handler (we want to return a response, error or not, so we set this up just in case) ❹ and wraps the HttpClient execute method, which requires a method object (either POST or GET in our case, preestablished) and a ResponseHandler as input. If we don't get an exception when we invoke HttpClient execute, all is well and the response details are placed into the ResponseHandler. If we do get an exception, we populate the error handler and pass it through to the ResponseHandler.

We call the local private execute method with the established details for either a POST or a GET request. The GET method is handled similarly to the POST, but we don't set parameters (with GET requests, we expect parameters encoded in the URL itself). Right now, our class supports only POST and GET, which cover 98 percent of the requests we generally need, but it could easily be expanded to support other HTTP method types.

The final part of the request helper class, shown in the following listing, takes us back to the first example (listing 6.7), which used the helper. Listing 6.9 outlines exactly what the convenience getResponseHandlerInstance method returns (constructing our helper requires a ResponseHandler, and this method returns a default one).

Listing 6.9 The final part of the HttpRequestHelper class

```
    public static ResponseHandler<String>
      getResponseHandlerInstance(final Handler handler) {     ❶ Require Handler parameter
        final ResponseHandler<String> responseHandler =
          new ResponseHandler<String>() {
            public String handleResponse(final HttpResponse response) {
                Message message = handler.obtainMessage();
                Bundle bundle = new Bundle();
                StatusLine status = response.getStatusLine();
                HttpEntity entity = response.getEntity();
                String result = null;
                if (entity != null) {
                    try {
                        result = StringUtils.inputStreamToString(
                          entity.getContent());                 ❷
                        bundle.putString(
                            "RESPONSE", result);            Put result value into Bundle
                        message.setData(bundle);
                        handler.sendMessage(message);      Set Bundle as
                                                           data into Message
                    } catch (IOException 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);      Send Message
                                                       via Handler
                }
                return result;
            }
        };
        return responseHandler;
    }
}
```

As we discuss the getResponseHandlerInstance method of our helper, we should note that although we find it helpful, it's entirely optional. You can still use the helper class without using this method. To do so, construct your own ResponseHandler and pass it in to the helper constructor, which is a perfectly plausible case. The getResponseHandlerInstance method builds a convenient default ResponseHandler that hooks in a Handler via a parameter ❶ and parses the response as a String ❷. The response String is sent back to the caller using the Handler Bundle and Message pattern we've seen used time and time again to pass messages between threads in our Android screens.

With the gory HttpRequestHelper details out of the way, and having already explored basic usage, we'll next turn to more involved uses of this class in the context of web service calls.

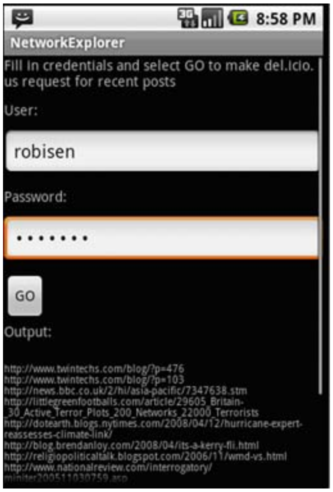
6.5 Web services

The term *web services* means many different things, depending on the source and the audience. To some, it's a nebulous marketing term that's never pinned down; to others, it's a rigid and specific set of protocols and standards. We're going to tackle it as a general concept, without defining it in depth, but not leaving it entirely undefined either.

Web services are a means of exposing an API over a technology-neutral network endpoint. They're a means to call a remote method or operation that's not tied to a specific platform or vendor and get a result. By this definition, POX over the network is included; so are REST and SOAP—and so is any other method of exposing operations and data on the wire in a neutral manner.

POX, REST, and SOAP are by far the most common web services around, so they're what we'll focus on in this section. Each provides a general guideline for accessing data and exposing operations, each in a more rigorous manner than the previous. POX basically exposes chunks of XML over the wire, usually over HTTP. REST is more detailed in that it uses the concept of *resources* to define data and then manipulates them with different HTTP methods using a URL-style approach (much like the Android `Intent` system in general, which we explored in previous chapters). SOAP is the most formal of them all, imposing strict rules about types of data, transport mechanisms, and security.

All these approaches have advantages and disadvantages, and these differences are amplified on a mobile platform like Android. Though we can't possibly cover all the details here, we'll touch on the differences as we discuss each of these concepts. We'll examine using a POX approach to return recent posts from the Delicious API (formerly del.icio.us), then we'll look at using REST with the Google GData AtomPub API. Up first is probably the most ubiquitous type of web service in use on the internet today, and therefore one you'll come across again and again when connecting Android applications—POX.



Fill in credentials and select GO to make del.icio.us request for recent posts

User:

robisen

Password:

•••••••

GO

Output:

```
http://www.twintechs.com/blog/?p=476
http://www.twintechs.com/blog/?p=103
http://news.bbc.co.uk/2/hi/asia-pacific/7347638.stm
http://littlegreenfootballs.com/article/29605_Britain-
_30_Active_Terror_Plots_200_Networks_22000_Terrorists
http://dotearth.blogs.nytimes.com/2008/04/12/hurricane-expert-
reassesses-climate-link/
http://blog.brendanloy.com/2008/04/its-a-kerry-fli.html
http://religiopoliticaltalk.blogspot.com/2006/11/wmd-vs.html
http://www.nationalreview.com/interrogatory/
miniter200511030759.asp
```

Figure 6.4 The Delicious recent posts screen from the NetworkExplorer application

6.5.1 POX—Putting it together with HTTP and XML

To work with POX, we're going to make network calls to the popular Delicious online social bookmarking site. We'll specify a username and password to log in to an HTTPS resource and return a list of recent posts, or *bookmarks*. This service returns raw XML data, which we'll parse into a JavaBean-style class and display as shown in figure 6.4.

The following listing shows the Delicious login and HTTPS `POST` `Activity` code from our NetworkExplorer application.

Listing 6.10 The Delicious HTTPS POX API with authentication from an `Activity`

```
public class DeliciousRecentPosts extends Activity {
    private static final String CLASSTAG =
     DeliciousRecentPosts.class.getSimpleName();
    private static final String URL_GET_POSTS_RECENT =
      "https://api.del.icio.us/v1/posts/recent?";

    . . . member var declarations for user, pass, output,
         and button (Views) omitted for brevity,
    private final Handler handler = new Handler() {

        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
```

1 Include Delicious URL

2 Provide Handler to update UI

```
        String bundleResult = msg.getData().getString("RESPONSE");
        output.setText(parseXMLResult(bundleResult));
    }
};
@Override
public void onCreate(final Bundle icicle) {
    super.onCreate(icicle);
    this.setContentView(R.layout.delicious_posts);
    . . .  inflate views omitted for brevity
    this.button.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            output.setText("");
            performRequest(user.getText().toString(),
              pass.getText().toString());                  Pass credentials to
        }                                                ❸ performRequest
    });
};
. . . onPause omitted for brevity
private void performRequest(String user, String pass) {
    this.progressDialog = ProgressDialog.show(this,
        "working . . .", "performing HTTP post to del.icio.us");
    final ResponseHandler<String> responseHandler =
        HTTPRequestHelper.getResponseHandlerInstance(this.handler);
    new Thread() {
        public void run() {
            HTTPRequestHelper helper =
              new HTTPRequestHelper(responseHandler);
            helper.performPost(URL_GET_POSTS_RECENT,           ❹ Use helper
              user, pass, null, null);                           for HTTP

        }
    }.start();
}                                                            ❺ Parse XML
private String parseXMLResult(String xmlString) {              String result
    StringBuilder result = new StringBuilder();
    try {
        SAXParserFactory spf = SAXParserFactory.newInstance();
        SAXParser sp = spf.newSAXParser();
        XMLReader xr = sp.getXMLReader();
        DeliciousHandler handler = new DeliciousHandler();
        xr.setContentHandler(handler);
        xr.parse(new InputSource(new StringReader(xmlString)));
        List<DeliciousPost> posts = handler.getPosts();
        for (DeliciousPost p : posts) {
            result.append("\n" + p.getHref());
        }
    } catch (Exception e) {
        // log and or handle
    }
    return result.toString();
}
```

To use a POX service, we need to know a bit about it, beginning with the URL endpoint ❶. To call the Delicious service, we again use a Handler to update the UI ❷, and we use the HttpRequestHelper we previously built and walked through in the last section. Again in this example, we have many fewer lines of code than if we didn't use the helper—lines of code we'd likely be repeating in different Activity classes. With the helper instantiated, we call the performRequest method with a username and password ❸. This method, via the helper, will log in to Delicious and return an XML chunk representing the most recently bookmarked items ❹.

To turn the raw XML into useful types, we then also include a parseXMLResult method ❺. Parsing XML is a subject in its own right, and we'll cover it in more detail in chapter 13, but the short takeaway with this method is that we walk the XML structure with a parser and return our own DeliciousPost data beans for each record. That's it—that's using POX to read data over HTTPS.

Building on the addition of XML to HTTP, above and beyond POX, is the REST architectural principle, which we'll explore next.

6.5.2 REST

While we look at REST, we'll also try to pull in another useful concept in terms of Android development: working with the various Google GData APIs (http://code.google.com/apis/gdata/). We used the GData APIs for our RestaurantFinder review information in chapter 3, but there we didn't authenticate, and we didn't get into the details of networking or REST. In this section, we'll uncover the details as we perform two distinct tasks: authenticate and retrieve a Google ClientLogin token and retrieve the Google Contacts data for a specified user. Keep in mind that as we work with the GData APIs in any capacity, we'll be using a REST-style API.

The main REST concepts are that you specify resources in a URI form and you use different protocol methods to perform different actions. The *Atom Publishing Protocol* (AtomPub) defines a REST-style protocol, and the GData APIs are an implementation of AtomPub (with some Google extensions). As we noted earlier, the entire Intent approach of the Android platform is a lot like REST. A URI such as content://contacts/1 is in the REST style. It includes a path that identifies the type of data and a particular resource (contact number 1).

That URI doesn't say what to do with contact 1, though. In REST terms, that's where the method of the protocol comes into the picture. For HTTP purposes, REST uses various methods to perform different tasks: POST (create, update, or in special cases, delete), GET (read), PUT (create, replace), and DELETE (delete). True HTTP REST implementations use all the HTTP method types and resources to construct APIs.

In the real world, you'll find few true REST implementations. It's much more common to see a REST-style API. This kind of API doesn't typically use the HTTP DELETE method (many servers, proxies, and so on, have trouble with DELETE) and overloads the more common GET and POST methods with different URLs for different tasks (by encoding a bit about what's to be done in the URL, or as a header or parameter, rather than relying strictly on the method). In fact, though many people refer to the GData APIs as REST, they're technically only REST-like, not true REST. That's not necessarily a bad thing; the idea is ease of use of the API rather than pattern purity. All in all, REST is a popular architecture or style because it's simple, yet powerful.

The following listing is an example that focuses on the network aspects of authentication with GData to obtain a ClientLogin token and use that token with a subsequent REST-style request to obtain Contacts data by including an email address as a resource.

Listing 6.11 Using the Google Contacts AtomPub API with authentication

```
public class GoogleClientLogin extends Activity {
    private static final String URL_GET_GTOKEN =
      "https://www.google.com/accounts/ClientLogin";
    private static final String URL_GET_CONTACTS_PREFIX =
      "http://www.google.com/m8/feeds/contacts/";
    private static final String URL_GET_CONTACTS_SUFFIX = "/full";
    private static final String GTOKEN_AUTH_HEADER_NAME = "Authorization";
    private static final String GTOKEN_AUTH_HEADER_VALUE_PREFIX =
      "GoogleLogin auth=";
    private static final String PARAM_ACCOUNT_TYPE = "accountType";
    private static final String PARAM_ACCOUNT_TYPE_VALUE =
      "HOSTED_OR_GOOGLE";
    private static final String PARAM_EMAIL = "Email";
    private static final String PARAM_PASSWD = "Passwd";
    private static final String PARAM_SERVICE = "service";
    private static final String PARAM_SERVICE_VALUE = "cp";
    private static final String PARAM_SOURCE = "source";
    private static final String PARAM_SOURCE_VALUE =
      "manning-unlockingAndroid-1.0";
    private String tokenValue;
    . . . View member declarations omitted for brevity
    private final Handler tokenHandler = new Handler() {
```

1 Create Handler token request

```

        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            String authToken = bundleResult;
            authToken = authToken.substring(authToken.indexOf("Auth=")
              + 5, authToken.length()).trim();
            tokenValue = authToken;
```

2 Set tokenValue

```

            GtokenText.setText(authToken);
        }
    };
    private final Handler contactsHandler =
      new Handler() {

        public void handleMessage(final Message msg) {
            progressDialog.dismiss();
            String bundleResult = msg.getData().getString("RESPONSE");
            output.setText(bundleResult);
        }
    };
    . . . onCreate and onPause omitted for brevity
    private void getToken(String email, String pass) {
```

3 Implement getToken

```

        final ResponseHandler<String> responseHandler =
          HTTPRequestHelper.getResponseHandlerInstance(
      this.tokenHandler);
    this.progressDialog = ProgressDialog.show(this,
      "working . . .", "getting Google ClientLogin token");
    new Thread() {
      public void run() {
        HashMap<String, String> params =
         new HashMap<String, String>();
        params.put(GoogleClientLogin.PARAM_ACCOUNT_TYPE,
         GoogleClientLogin.PARAM_ACCOUNT_TYPE_VALUE);
        params.put(GoogleClientLogin.PARAM_EMAIL, email);
        params.put(GoogleClientLogin.PARAM_PASSWD, pass);
        params.put(GoogleClientLogin.PARAM_SERVICE,

              GoogleClientLogin.PARAM_SERVICE_VALUE);
        params.put(GoogleClientLogin.PARAM_SOURCE,

        GoogleClientLogin.PARAM_SOURCE_VALUE);
        HTTPRequestHelper helper =
         new HTTPRequestHelper(responseHandler);
        helper.performPost(HTTPRequestHelper.MIME_FORM_ENCODED,
          GoogleClientLogin.URL_GET_GTOKEN,
           null, null, null, params);
      }
    }.start();
  }
   private void getContacts(final String email,final String token) {

      final ResponseHandler<String> responseHandler =
       HTTPRequestHelper.getResponseHandlerInstance(
        this.contactsHandler);
    this.progressDialog = ProgressDialog.show(this,
     "working . . .", "getting Google Contacts");
    new Thread() {
      public void run() {
        HashMap<String, String> headers =
         new HashMap<String, String>();
        headers.put(GoogleClientLogin.GTOKEN_AUTH_HEADER_NAME,
         GoogleClientLogin.GTOKEN_AUTH_HEADER_VALUE_PREFIX
          + token);

        String encEmail = email;
        try {
          encEmail = URLEncoder.encode(encEmail,
           "UTF-8");

        } catch (UnsupportedEncodingException e) {
          // log and or handle
        }
        String url =
         GoogleClientLogin.URL_GET_CONTACTS_PREFIX + encEmail
           + GoogleClientLogin.URL_GET_CONTACTS_SUFFIX;
        HTTPRequestHelper helper = new
         HTTPRequestHelper(responseHandler);
        helper.performGet(url, null, null, headers);
      }
    }.start();
  }
}
```

4 Required parameters for ClientLogin

5 Perform POST to get token

6 Implement getContacts

7 Add token as header

8 Encode email address in URL

9 Make GET request for Contacts

After a host of constants that represent various String values we'll use with the GData services, we have several Handler instances in this class, beginning with a tokenHandler ❶. This handler updates a UI TextView when it receives a message, like similar examples you saw previously, and updates a non-UI member tokenValue variable that other portions of our code will use ❷. The next Handler we have is the contactsHandler that will be used to update the UI after the contacts request.

Beyond the handlers, we have the getToken method ❸. This method includes all the required parameters for obtaining a ClientLogin token from the GData servers (http://code.google.com/apis/gdata/auth.html) ❹. After the setup to obtain the token, we make a POST request via the request helper ❺.

After the token details are taken care of, we have the getContacts method ❻. This method uses the token obtained via the previous method as a header ❼. After you have the token, you can cache it and use it with all subsequent requests; you don't need to obtain the token every time. Next, we encode the email address portion of the Contacts API URL ❽, and we make a GET request for the data—again using the HttpRequestHelper ❾.

With this approach, we're making several network calls (one as HTTPS to get the token and another as HTTP to get data) using our previously defined helper class. When the results are returned from the GData API, we parse the XML block and update the UI.

GData ClientLogin and CAPTCHA

Though we included a working ClientLogin example in listing 6.11, we also skipped over an important part—CAPTCHA. Google might optionally require a CAPTCHA with the ClientLogin approach. To fully support ClientLogin, you need to handle that response and display the CAPTCHA to the user, then resend a token request with the CAPTCHA value that the user entered. For details, see the GData documentation.

Now that we've explored some REST-style networking, the last thing we need to discuss with regard to HTTP and Android is SOAP. This topic comes up frequently in discussions of networking mobile devices, but sometimes the forest gets in the way of the trees in terms of framing the real question.

6.5.3 *To SOAP or not to SOAP, that is the question*

SOAP is a powerful protocol that has many uses. We would be remiss if we didn't at least mention that though it's possible to use SOAP on a small, embedded device such as a smartphone, regardless of the platform, it's not recommended. The question within the limited resources environment Android inhabits is really more one of *should* it be done rather than *can* it be done.

Some experienced developers, who might have been using SOAP for years on other devices, might disagree. The things that make SOAP great are its support for strong types (via XML Schema), its support for transactions, its security and encryption, its support for message orchestration and choreography, and all the related WS-* standards. These things are invaluable in many server-oriented computing environments, whether or not they involve the enterprise. They also add a great deal of overhead, especially on a small, embedded device. In fact, in many situations where people use SOAP on embedded devices, they often don't bother with the advanced features—and they use plain XML with the overhead of an envelope at the end of the day anyway. On an embedded device, you often get better performance, and a simpler design, by using a REST- or POX-style architecture and avoiding the overhead of SOAP.

Even with the increased overhead, it makes sense in some situations to investigate using SOAP directly with Android. When you need to talk to existing SOAP services that you have no control over, SOAP might make sense. Also, if you already have J2ME clients for existing SOAP services, you might be able to port those in a limited set of cases. Both these approaches make it easier only on you, the developer; they have either no effect or a negative one in terms of performance on the user. Even when you're working with existing SOAP services, remember that you can often write a POX- or REST-style proxy for SOAP services on the server side and call that from Android, rather than use SOAP directly from Android.

If you feel like SOAP is still the right choice, you can use one of several ports of the kSOAP toolkit (http://ksoap2.sourceforge.net/), which is specially designed for SOAP on an embedded Java device. Keep in mind that even the kSOAP documentation states, "SOAP introduces some significant overhead for web services that may be problematic for mobile devices. If you have full control over the client and the server, a REST-based architecture may be more adequate." In addition, you might be able to write your own parser for simple SOAP services that don't use fancy SOAP features and just use a POX approach that includes the SOAP XML portions you require (you can always roll your own, even with SOAP).

All in all, to our minds the answer to the question is to not use SOAP on Android, even though you can. Our discussion of SOAP, even though we don't advocate it, rounds out our more general web services discussion, and that wraps up our networking coverage.

6.6 Summary

In this chapter, we started with a brief background of basic networking concepts, from nodes and addresses to layers and protocols. With that general background in place, we covered details about how to obtain network status information and showed several different ways to work with the IP networking capabilities of the platform.

In terms of networking, we looked at using basic sockets and the `java.net` package. Then we also examined the included Apache HttpClient API. HTTP is one of the most common—and most important—networking resources available to the Android platform. Using `HttpClient`, we covered a lot of territory in terms of different request types, parameters, headers, authentication, and more. Beyond basic HTTP, we also explored POX and REST, and we discussed a bit of SOAP—all of which use HTTP as the transport mechanism.

Now that we've covered a good deal of the networking possibilities, and hopefully given you at least a glint of an idea of what you can do with server-side APIs and integration with Android, we're going to turn to another important part of the Android world—telephony.

7 Telephony

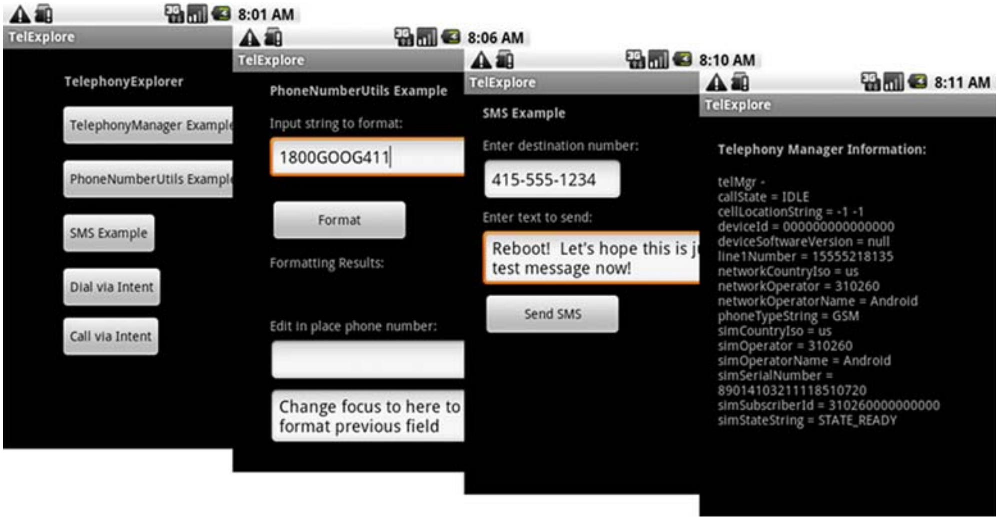
This chapter covers

- Making and receiving phone calls
- Capturing call-related events
- Obtaining phone and service information
- Using SMS

People use Android devices to surf the web, download and store data, access networks, find location information, and use many types of applications. Android can even make phone calls.

Android phones support dialing numbers, receiving calls, sending and receiving text and multimedia messages, and other related telephony services. In contrast to other smartphone platforms, all these items are accessible to developers through simple-to-use APIs and built-in applications. You can easily leverage Android's telephony support into your own applications.

In this chapter, we'll discuss telephony in general and cover terms related to mobile devices. We'll move on to basic Android telephony packages, which handle calls using built-in `Intent` actions, and more advanced operations via the `TelephonyManager` and `PhoneStateListener` classes. The `Intent` actions can initiate basic phone calls in your applications. `TelephonyManager` doesn't make phone calls directly but is used to retrieve all kinds of telephony-related data, such as the



1800GOOG411

415-555-1234

Reboot! Let's hope this is j test message now!

```
Telephony Manager Information:

telMgr -
callState = IDLE
cellLocationString = -1 -1
deviceId = 000000000000000
deviceSoftwareVersion = null
line1Number = 15555218135
networkCountryIso = us
networkOperator = 310260
networkOperatorName = Android
phoneTypeString = GSM
simCountryIso = us
simOperator = 310260
simOperatorName = Android
simSerialNumber =
89014103211118510720
simSubscriberId = 310260000000000
simStateString = STATE_READY
```

Figure 7.1 TelephonyExplorer main screen, along with the related activities the sample application performs

state of the voice network, the device's own phone number, and other details. `TelephonyManager` supports adding a `PhoneStateListener`, which can alert you when call or phone network states change.

After covering basic telephony APIs, we'll move on to sending and receiving SMS messages. Android provides APIs that allow you to send SMS messages and be notified when SMS messages are received. We'll also touch on emulator features that allow you to test your app by simulating incoming phone calls or messages.

Once again, a sample application will carry us through the concepts related to the material in this chapter. You'll build a sample TelephonyExplorer application to demonstrate dialing the phone, obtaining phone and service state information, adding listeners to the phone state, and working with SMS. Your TelephonyExplorer application will have several basic screens, as shown in figure 7.1.

TelephonyExplorer exercises the telephony-related APIs while remaining simple and uncluttered. Before we start to build TelephonyExplorer, let's first define telephony itself.

7.1 Exploring telephony background and terms

Whether you're a new or an experienced mobile developer, it's important to clarify terms and set out some background for discussing telephony.

First, *telephony* is a general term that refers to electrical voice communications over telephone networks. Our scope is, of course, the mobile telephone networks that Android devices[1] participate in, specifically the Global System for Mobile Communications (GSM) and Code Division Multiple Access (CDMA) networks.

[1] For a breakdown of all Android devices released in 2008-2010, go here: http://www.androphones.com/all-android-phones.php.

GSM and CDMA are cellular telephone networks. Devices communicate over radio waves and specified frequencies using cell towers. The standards must define a few important things, such as identities for devices and cells, along with all the rules for making communications possible.

7.1.1 Understanding GSM

We won't delve into the underlying details of the networks, but it's important to know some key facts. GSM is based on Time Division Multiple Access (TDMA), a technology that slices time into smaller chunks to allow multiple callers to share the same frequency range. GSM was the first network that the Android stack supported for voice calls; it's ubiquitous in Europe and very common in North America. GSM devices use Subscriber Identity Module (SIM) cards to store important network and user settings.

A SIM card is a small, removable, secure smart card. Every device that operates on a GSM network has specific unique identifiers, which are stored on the SIM card or on the device itself:

- *Integrated Circuit Card Identifier* (ICCID)—Identifies a SIM card; also known as a SIM Serial Number, or SSN.
- *International Mobile Equipment Identity* (IMEI)—Identifies a physical device. The IMEI number is usually printed underneath the battery.
- *International Mobile Subscriber Identity* (IMSI)—Identifies a subscriber (and the network that subscriber is on).
- *Location Area Identity* (LAI)—Identifies the region within a provider network that's occupied by the device.
- *Authentication key* (Ki)—A 128-bit key used to authenticate a SIM card on a provider network.

GSM uses these identification numbers and keys to validate and authenticate a SIM card, the device holding it, and the subscriber on the network and across networks.

Along with storing unique identifiers and authentication keys, SIM cards often store user contacts and SMS messages. Users can easily move their SIM card to a new device and carry along contact and message data. Currently, the Android platform handles the SIM interaction, and developers can get read-only access via the telephony APIs.

7.1.2 Understanding CDMA

The primary rival to GSM technology is CDMA, which uses a different underlying technology that's based on using different encodings to allow multiple callers to share the same frequency range. CDMA is widespread in the Unites States and common in some Asian countries.

Unlike GSM phones, CDMA devices don't have a SIM card or other removable module. Instead, certain identifiers are burned into the device, and the carrier must maintain the link between each device and its subscriber. CDMA devices have a separate set of unique identifiers:

- *Mobile Equipment Identifier (MEID)*—Identifies a physical device. This number is usually printed under the battery and is available from within device menus. It corresponds to GSM's IMEI.
- *Electronic Serial Number (ESN)*—The predecessor to the MEID, this number is shorter and identifies a physical device.
- *Pseudo Electronic Serial Number (pESN)*—A hardware identifier, derived from the MEID, that's compatible with the older ESN standard. The ESN supply was exhausted several years ago, so pESNs provide a bridge for legacy applications built around ESN. A pESN always starts with 0x80 in hex format or 128 in decimal format.

Unlike GSM phones, which allow users to switch devices by swapping out SIM cards, CDMA phones require you to contact your carrier if you want to transfer an account to a new device. This process is often called an *ESN swap* or *ESN change*. Some carriers make this easy, and others make it difficult. If you'll be working on CDMA devices, learning how to do this with your carrier can save you thousands of dollars in subscriber fees.

> **NOTE** A few devices, sometimes called *world phones*, support both CDMA and GSM. These devices often have two separate radios and an optional SIM card. Currently, such devices operate only on one network or the other at any given time. Additionally, these devices are often restricted to using only particular carriers or technologies in particular countries. You generally don't need to do anything special to support these devices, but be aware that certain phones might appear to change their network technology from time to time.

Fortunately, few applications need to deal with the arcana of GSM and CDMA technology. In most cases, you only need to know that your program is running on a device that in turn is running on a mobile network. You can leverage that network to make calls and inspect the device to find unique identifiers. You can locate this sort of information by using the `TelephonyManager` class.

7.2 Accessing telephony information

Android provides an informative manager class that supplies information about many telephony-related details on the device. Using `TelephonyManager`, you can access phone properties and obtain phone network state information.

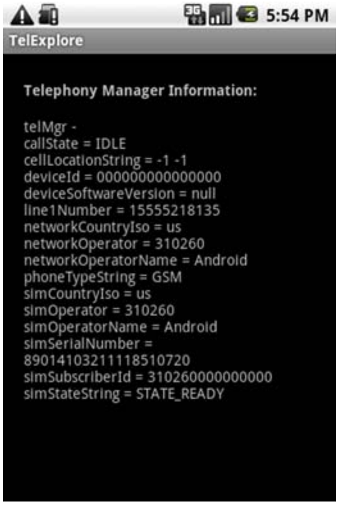
> **NOTE** Starting with version 2.1 of the Android OS, devices no longer need to support telephony features. Expect more and more non-phone devices to reach the market, such as set-top boxes and auto devices. If you want to reach the largest possible market with your app, you should leverage telephony features but fail gracefully if they're not available. If your application makes sense only when running on a phone, go ahead and use any phone features you require.

You can attach a PhoneStateListener event listener to the phone by using the manager. Attaching a PhoneStateListener makes your applications aware of when the phone gains and loses service, and when calls start, continue, or end.

Next, we'll examine several parts of the TelephonyExplorer example application to look at both these classes. We'll start by obtaining a TelephonyManager instance and using it to query useful telephony information.

7.2.1 Retrieving telephony properties

The android.telephony package contains the TelephonyManager class, which provides details about the phone status. Let's retrieve and display a small subset of that information to demonstrate the approach. First, you'll build an Activity that displays a simple screen showing some of the information you can obtain via TelephonyManager, as shown in figure 7.2.



```
Telephony Manager Information:

telMgr -
callState = IDLE
cellLocationString = -1 -1
deviceId = 000000000000000
deviceSoftwareVersion = null
line1Number = 15555218135
networkCountryIso = us
networkOperator = 310260
networkOperatorName = Android
phoneTypeString = GSM
simCountryIso = us
simOperator = 310260
simOperatorName = Android
simSerialNumber =
89014103211118510720
simSubscriberId = 310260000000000
simStateString = STATE_READY
```

Figure 7.2 Displaying device and phone network meta-information obtained from TelephonyManager

The TelephonyManager class is the information hub for telephony-related data in Android. The following listing demonstrates how you obtain a reference to this class and use it to retrieve data.

Listing 7.1 Obtaining a TelephonyManager reference and using it to retrieve data

```
// . . . start of class omitted for brevity
    final TelephonyManager telMgr =
      (TelephonyManager) this.getSystemService(
        Context.TELEPHONY_SERVICE);
// . . . onCreate method and others omitted for brevity
    public String getTelephonyOverview(
      TelephonyManager telMgr) {
        String callStateString = "NA";
        int callState = telMgr.getCallState();
        switch (callState) {
        case TelephonyManager.CALL_STATE_IDLE:
            callStateString = "IDLE";
            break;
        case TelephonyManager.CALL_STATE_OFFHOOK:
            callStateString = "OFFHOOK";
            break;
        case TelephonyManager.CALL_STATE_RINGING:
            callStateString = "RINGING";
            break;
        }

        GsmCellLocation cellLocation =
          (GsmCellLocation) telMgr.getCellLocation();
        String cellLocationString =
```

1 Get TelephonyManager from Context

2 Implement information helper method

3 Obtain call state information

```
    cellLocation.getLac() + " " + cellLocation.getCid();
  String deviceId = telMgr.getDeviceId();                   ❹ Get device
  String deviceSoftwareVersion =                              information
   telMgr.getDeviceSoftwareVersion();
  String line1Number = telMgr.getLine1Number();
  String networkCountryIso = telMgr.getNetworkCountryIso();
  String networkOperator = telMgr.getNetworkOperator();
  String networkOperatorName = telMgr.getNetworkOperatorName();

  String phoneTypeString = "NA";
  int phoneType = telMgr.getPhoneType();
  switch (phoneType) {
  case TelephonyManager.PHONE_TYPE_GSM:
    phoneTypeString = "GSM";
    break;
  case TelephonyManager.PHONE_TYPE_CDMA:
    phoneTypeString = "CDMA";
    break;
  case TelephonyManager.PHONE_TYPE_NONE:
    phoneTypeString = "NONE";
    break;
  }

  String simCountryIso = telMgr.getSimCountryIso();
  String simOperator = telMgr.getSimOperator();
  String simOperatorName = telMgr.getSimOperatorName();
  String simSerialNumber = telMgr.getSimSerialNumber();
  String simSubscriberId = telMgr.getSubscriberId();
  String simStateString = "NA";
  int simState = telMgr.getSimState();
  switch (simState) {
  case TelephonyManager.SIM_STATE_ABSENT:
    simStateString = "ABSENT";
    break;
  case TelephonyManager.SIM_STATE_NETWORK_LOCKED:
    simStateString = "NETWORK_LOCKED";
    break;
  // . . . other SIM states omitted for brevity
  }

  StringBuilder sb = new StringBuilder();
  sb.append("telMgr - ");
  sb.append("  \ncallState = " + callStateString);
  // . . . remainder of appends omitted for brevity
  return sb.toString();
}
```

We use the current Context, through the getSystemService method with a constant, to obtain an instance of the TelephonyManager class ❶. After you have the manager, you can use it as needed. In this case, we create a helper method to get data from the manager and return it as a String that we later display on the screen ❷.

The manager allows you to access phone state data, such as whether a call is in progress ❸, the device ID and software version ❹, the phone number registered to the current user/SIM, and other SIM details, such as the subscriber ID (IMSI) and the current SIM state. TelephonyManager offers even more properties; see the Javadocs for complete details.

> **NOTE** Methods generally return null if they don't apply to a particular device; for example, getSimOperatorName returns null for CDMA phones. If you want to know in advance what type of device you're working with, try using the method getPhoneType.

For this class to work, you must set the READ_PHONE_STATE permission in the manifest. Without it, security exceptions will be thrown when you try to read data from the manager. Phone-related permissions are consolidated in table 7.1.

In addition to providing telephony-related information, including metadata about the device, network, and subscriber, TelephonyManager allows you to attach a PhoneStateListener, which we'll describe in the next section.

7.2.2 Obtaining phone state information

A phone can be in any one of several conditions. The primary phone states include *idle* (waiting), in a call, or initiating a call. When you're building applications on a mobile device, sometimes you not only need to know the current phone state, but you also want to know when the state changes.

In these cases, you can attach a listener to the phone and subscribe to receive notifications of published changes. With Android, you use a PhoneStateListener, which attaches to the phone through TelephonyManager. The following listing demonstrates a sample usage of both these classes.

Listing 7.2 Attaching a PhoneStateListener via the TelephonyManager

```
@Override
public void onStart() {
    super.onStart();
    final TelephonyManager telMgr =
         (TelephonyManager) this.getSystemService(
             Context.TELEPHONY_SERVICE);
    PhoneStateListener phoneStateListener =
      new PhoneStateListener() {

        public void onCallStateChanged(
          int state, String incomingNumber) {
            telMgrOutput.setText(getTelephonyOverview(telMgr));
        }
    };
    telMgr.listen(phoneStateListener,
      PhoneStateListener.LISTEN_CALL_STATE);
    String telephonyOverview = this.getTelephonyOverview(telMgr);
    this.telMgrOutput.setText(telephonyOverview);
}
```

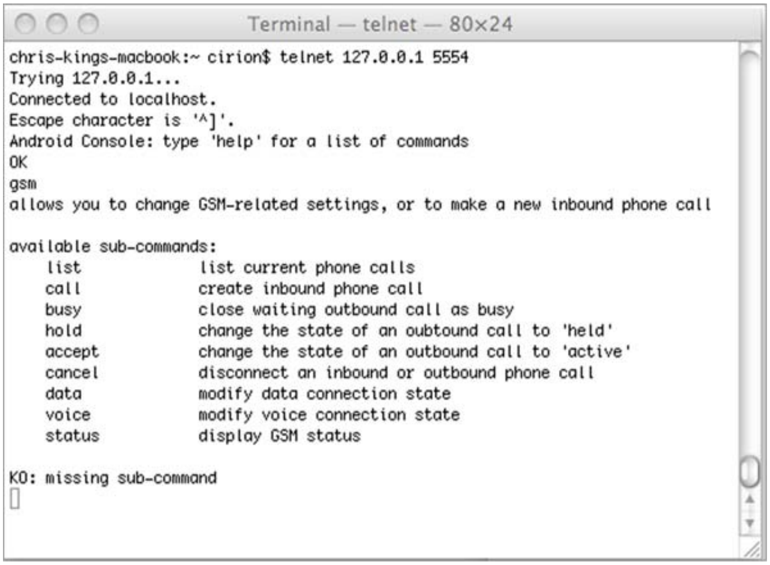
To start working with a PhoneStateListener, you need to acquire an instance of TelephonyManager. PhoneStateListener itself is an interface, so you need to create an implementation, including the required onCallStateChanged method. When you have a valid PhoneStateListener instance, you attach it by assigning it to the manager with the listen method.

Listing 7.2 shows how to listen for any PhoneStateListener.LISTEN_CALL_STATE change in the phone state. This constant value comes from a list of available states that are in PhoneStateListener class. You can use a single value when assigning a listener with the listen method, as demonstrated in listing 7.2, or you can combine multiple values to listen for multiple states.

If a call state change does occur, it triggers the action defined in the onCallStateChanged method of your PhoneStateListener. In this example, we reset the details on the screen using the getTelephonyOverview method from listing 7.1. You can filter this method further, based on the passed-in int state.

To see the values in this example change while you're working with the emulator, you can use the SDK tools to send incoming calls or text messages and change the state of the voice connection. You can access these options from the DDMS perspective in Eclipse. Additionally, the emulator includes a mock GSM modem that you can manipulate using the gsm command from the console. Figure 7.3 shows an example session from the console that demonstrates using the gsm command. For complete details, see the emulator telephony documentation at http://code.google.com/android/reference/emulator.html#telephony.

Now that we've covered the major elements of telephony, let's start exploring basic uses of the telephony APIs and other related facilities. We'll intercept calls, leverage telephony utility classes, and make calls from within applications.



```
chris-kings-macbook:~ cirion$ telnet 127.0.0.1 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 oubtound 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

KO: missing sub-command
```

Figure 7.3 An Android console session demonstrating the gsm command and available subcommands

7.3 Interacting with the phone

In regular development, you'll often want to use your Android device as a phone. You might dial outbound calls through simple built-in intents, or intercept calls to modify them in some way. In this section, we'll cover these basic tasks and examine some of the phone-number utilities Android provides for you.

One of the more common things you'll do with Android telephony support doesn't even require using the telephony APIs directly: making calls using built-in intents.

7.3.1 Using intents to make calls

As we demonstrated in chapter 4, to invoke the built-in dialer and make a call all you need to use is the `Intent.ACTION_CALL` action and the `tel: Uri`. This approach invokes the dialer application, populates the dialer with the provided telephone number (taken from the `Uri`), and initiates the call.

Alternatively, you can invoke the dialer application with the `Intent.ACTION_DIAL` action, which also populates the dialer with the supplied phone number but stops short of initiating the call. The following listing demonstrates both techniques using their respective actions.

Listing 7.3 Using `Intent` actions to dial and call using the built-in dialer application

```
dialintent = (Button) findViewById(R.id.dialintent_button);
   dialintent.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                Intent intent =
                  new Intent(Intent.DIAL_ACTION,
                    Uri.parse("tel:" + NUMBER));
                startActivity(intent);
            }
        });
    callintent = (Button) findViewById(R.id.callintent_button);
    callintent.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                Intent intent =
                  new Intent(Intent.CALL_ACTION,
                    Uri.parse("tel:" + NUMBER));
                startActivity(intent);
            }
        });
```

By now you should feel quite comfortable using intents in the Android platform. In this listing, we again take advantage of Android's loose coupling, in this case to make outgoing calls to specified numbers. First, you set the action you want to take place, either populating the dialer with `ACTION_DIAL` or populating the dialer *and* initiating a call with `ACTION_CALL`. In either case, you also need to specify the telephone number you want to use with the `Intent Uri`.

Dialing calls also requires the proper permissions, which your application manifest includes in order to access and modify the phone state, dial the phone, or intercept

Table 7.1 Phone-related manifest permissions and their purpose

Phone-related permission	Purpose
`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.MODIFY_PHONE_STATE`	Allow the application to modify the phone state; for example, to turn the radio on or off
`android.permission.PROCESS_OUTGOING_CALLS`	Allow application to receive broadcast for outgoing calls and modify
`android.permission.READ_PHONE_STATE`	Allow application to read the phone state

phone calls (shown in section 7.3.3). Table 7.1 lists the relevant phone-related permissions and their purposes. For more detailed information, see the security section of the Android documentation at http://code.google.com/android/devel/security.html.

Android makes dialing simple with built-in handling via intents and the dialer application. The PhoneNumberUtils class, which you can use to parse and validate phone number strings, helps simplify dialing even more, while keeping numbers human-readable.

7.3.2 Using phone number-related utilities

Applications running on mobile devices that support telephony deal with a lot of String formatting for phone numbers. Fortunately, the Android SDK provides a handy utility class that helps to mitigate the risks associated with this task and standardize the numbers you use—PhoneNumberUtils.

The PhoneNumberUtils class parses String data into phone numbers, transforms alphabetical keypad digits into numbers, and determines other properties of phone numbers. The following listing shows an example of using this class.

Listing 7.4 Working with the PhoneNumberUtils class

```
// Imports omitted for brevity
 private TextView pnOutput;
 private EditText pnInput;
 private EditText pnInPlaceInput;
 private Button pnFormat;
// Other instance variables and methods omitted for brevity
 this.pnFormat.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        String phoneNumber = PhoneNumberUtils.formatNumber(
          pnInput.getText().toString());

        phoneNumber = PhoneNumberUtils.convertKeypadLettersToDigits(
          pnInput.getText().toString());
```

1 Format as phone number

2 Convert alpha characters to digits

```
     StringBuilder result = new StringBuilder();
     result.append(phoneNumber);
     result.append("\nisGlobal - "
          + PhoneNumberUtils.isGlobalPhoneNumber(phoneNumber));
     result.append("\nisEmergency - "
          + PhoneNumberUtils.isEmergencyNumber(phoneNumber));
     result.append("\ncompare to 415-555-1234 - " +
          PhoneNumberUtils.compare(phoneNumber, "415-555-1234"));
        pnOutput.setText(result.toString());
        pnInput.setText("");
  }
});
```

❸ **Compare to another number**

The PhoneNumberUtils class offers several static helper methods for parsing phone numbers, including the useful formatNumber. This method takes a single String as input and uses the default locale settings to return a formatted phone number ❶. Additional methods format a number using a locale you specify, parse different segments of a number, and so on. Parsing a number can be combined with another helpful method, convertKeypadLettersToDigits, to convert any alphabetic keypad letter characters into digits ❷. The conversion method won't work unless it already recognizes the format of a phone number, so you should run the format method first.

Along with these basic methods, you can also check properties of a number string, such as whether the number is global and whether it represents an emergency call. The compare method lets you see whether a given number matches another number ❸, which is useful for user-entered numbers that might include dashes or dots.

> **NOTE** Android defines a global number as any string that contains one or more digits; it can optionally be prefixed with a + symbol, and can optionally contain dots or dashes. Even strings like 3 and +4-2 are considered global numbers. Android makes no guarantee that a phone can even dial such a number; this utility simply provides a basic check for whether something looks like it could be a phone number in some country.

You can also format a phone number with the overloaded formatNumber method. This method is useful for any Editable, such as the common EditText (or TextView). This method updates the provided Editable in-place, as shown in the following listing.

Listing 7.5 Using in-place Editable View formatting via PhoneNumberUtils

```
this.pnInPlaceInput.setOnFocusChangeListener(
  new OnFocusChangeListener() {
      public void onFocusChange(View v, boolean hasFocus) {
          if (v.equals(pnInPlaceInput) && (!hasFocus)) {
           PhoneNumberUtils.formatNumber(
             pnInPlaceInput.getText(),
               PhoneNumberUtils.FORMAT_NANP);
        }
    }
    });
```

The in-place editor can be combined with a dynamic update using various techniques. You can make the update happen automatically when the focus changes from a phone-number field. The in-place edit does not provide the keypad alphabetic character-to-number conversion automatically. To ensure that the conversion occurs, we've implemented an `OnFocusChangeListener`. Inside the `onFocusChange` method, which filters for the correct `View` item, we call the `formatNumber` overload, passing in the respective `Editable` and the formatting style we want to use. NANP stands for North American Numbering Plan, which includes an optional country and area code and a 7-digit local phone number.

> **NOTE** `PhoneNumberUtils` also defines a Japanese formatting plan, and might add others in the future.

Now that you can use the phone number utilities and make calls, we can move on to the more challenging and interesting task of call interception.

7.3.3 *Intercepting outbound calls*

Imagine writing an application that catches outgoing calls and decorates or aborts them, based on certain criteria. The following listing shows how to perform this type of interception.

Listing 7.6 Catching and aborting an outgoing call

```
public class OutgoingCallReceiver extends BroadcastReceiver {
    public static final String ABORT_PHONE_NUMBER = "1231231234";
    @Override                                                     ❶ Override
    public void onReceive(Context context, Intent intent) {         onReceive
        if (intent.getAction().equals(
          Intent.ACTION_NEW_OUTGOING_CALL)) {          ❷ Filter Intent for action
            String phoneNumber =
              intent.getExtras().getString(Intent.EXTRA_PHONE_NUMBER);
            if ((phoneNumber != null)
                 && phoneNumber.equals(
                   OutgoingCallReceiver.ABORT_PHONE_NUMBER)) {
                Toast.makeText(context,
                    "NEW_OUTGOING_CALL intercepted to number "
                      + "123-123-1234 - aborting call",
                     Toast.LENGTH_LONG).show();
                this.abortBroadcast();
            }
        }
    }
}
```

Our interception class starts by extending `BroadcastReceiver`. The new subclass implements the `onReceive` method ❶. Within this method, we filter on the `Intent` action we want ❷, then we get the `Intent` data using the phone number key. If the phone number matches, we send a `Toast` alert to the UI and abort the outgoing call by calling the `abortBroadcast` method.

Beyond dialing out, formatting numbers, and intercepting calls, Android also provides support for sending and receiving SMS. Managing SMS can seem daunting, but provides significant rewards, so we're going to focus on it for the rest of the chapter.

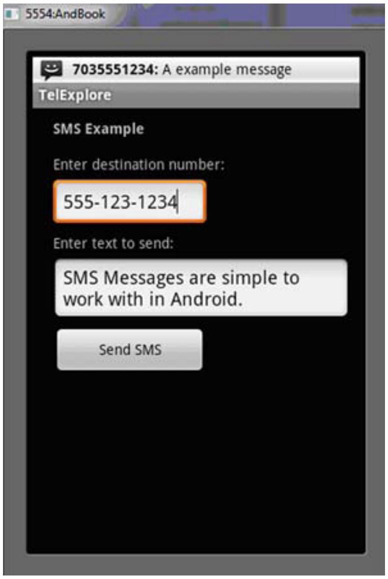
7.4 Working with messaging: SMS

Mobile devices use the Short Message Service (SMS), a hugely popular and important means of communication, to send simple text messages with small amounts of data. Android includes a built-in SMS application that allows users to send, view, and reply to SMS messages. Along with the built-in user-facing apps and the related `ContentProvider` for interacting with the default text-messaging app, the SDK provides APIs for developers to send and receive messages programmatically.

Because Android now supplies an excellent built-in SMS message application, you might wonder why anyone would bother building another one. The Android market sells several superior third-party SMS messaging applications, but SMS can do a lot more than text your contacts. For example, you could build an application that, upon receiving a special SMS, sends back another SMS containing its location information. Due to the nature of SMS, this strategy might succeed, while another approach like trying to get the phone to transmit its location in real time would fail. Alternately, adding SMS as another communications channel can enhance other applications. Best of all, Android makes working with SMS relatively simple and straightforward.

To explore Android's SMS support, you'll create an app that sends and receives SMS messages. The screen in figure 7.4 shows the SMS-related `Activity` you'll build in the TelephonyExplorer application.

To get started working with SMS, you'll first build a class that programmatically sends SMS messages, using the `SmsManager`.



7035551234: A example message

Enter destination number:
555-123-1234

Enter text to send:
SMS Messages are simple to work with in Android.

Figure 7.4 An `Activity` that sends SMS messages

7.4.1 Sending SMS messages

The `android.telephony` package contains the `SmsManager` and `SmsMessage` classes. The `SmsManager` defines many important SMS-related constants, and also provides the `sendDataMessage`, `sendMultipartTextMessage`, and `sendTextMessage` methods.

> **NOTE** Early versions of Android provided access to SMS only through the `android.telephony.gsm` subpackage. Google has deprecated this usage, but if you must target older versions of the OS, look there for SMS-related functions. Of course, such classes work only on GSM-compatible devices.

The following listing shows an example from our TelephonyExplorer application that uses the SMS manager to send a simple text message.

Listing 7.7 Using `SmsManager` to send SMS messages

```
// . . .  start of class omitted for brevity
    private Button smsSend;
    private SmsManager smsManager;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        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();
        final PendingIntent sentIntent =
          PendingIntent.getActivity(
            this, 0, new Intent(this,
              SmsSendCheck.class), 0);

        this.smsSend.setOnClickListener(new OnClickListener() {
            public void onClick(View v) {
                String dest = smsInputDest.getText().toString();
                if (PhoneNumberUtils.
                     isWellFormedSmsAddress(dest)) {
                       smsManager.sendTextMessage(
                         smsInputDest.getText().toString, null,
                         smsInputText.getText().toString(),
                         sentIntent, null);
                     Toast.makeText(SmsExample.this,
                       "SMS message sent",
                        Toast.LENGTH_LONG).show();
                } else {
                   Toast.makeText(SmsExample.this,
                    "SMS destination invalid - try again",
                     Toast.LENGTH_LONG).show();
                }
            }
        });
    }
```

❶ Get SmsManager handle

❷ Create PendingIntent for post action

❸ Check that destination is valid

Before doing anything with SMS messages, we must obtain an instance of the `SmsManager` with the static `getDefault` method ❶. The manager will also send the message later. Before we can send the message, we need to create a `PendingIntent` to provide to the send method.

A `PendingIntent` can specify an `Activity`, `Broadcast`, or `Service` that it requires. In our case, we use the `getActivity` method, which requests an `Activity`, and then we specify the context, request code (not used for this case), the `Intent` to execute, and additional flags ❷. The flags indicate whether the system should create a new instance of the referenced `Activity` (or `Broadcast` or `Service`), if one doesn't already exist.

What is a PendingIntent?

A `PendingIntent` specifies an action to take in the future. It lets you pass a future `Intent` to another application and allow that application to execute that `Intent` as if it had the same permissions as your application, whether or not your application is still around when the `Intent` is eventually invoked. A `PendingIntent` provides a means for applications to work, even after their process exits. It's important to note that even after the application that created the `PendingIntent` has been killed, that `Intent` can still run.

Next, we check that the destination address is valid for SMS ❸, and we send the message using the manager's `sendTextMessage` method.

This `send` method takes several parameters. The following snippet shows the signature of this method:

```
sendDataMessage(String destinationAddress, String scAddress,
  short destinationPort, byte[] data, PendingIntent sentIntent,
  PendingIntent deliveryIntent)
```

The method requires the following parameters:

- `destinationAddress`—The phone number to receive the message.
- `scAddress`—The messaging center address on the network; you should almost always leave this as `null`, which uses the default.
- `destinationPort`—The port number for the recipient handset.
- `data`—The payload of the message.
- `sentIntent`—The `PendingIntent` instance that's fired when the message is successfully sent.
- `deliveryIntent`—The `PendingIntent` instance that's fired when the message is successfully received.

> **NOTE** GSM phones generally support receiving SMS messages to a particular port, but CDMA phones generally don't. Historically, port-directed SMS messages have allowed text messages to be delivered to a particular application. Modern phones support better solutions; in particular, if you can use a server for your application, consider using Android Cloud to Device Messaging (C2DM)[2] for Android phones with software version 2.2 or later.

Much like the phone permissions listed in table 7.1, SMS-related tasks also require manifest permissions. SMS permissions are shown in table 7.2.

The AndroidManifest.xml file for the TelephonyExplorer application contains these permissions.

[2] Read Tim Bray's detailed article for more about C2DM: http://android-developers.blogspot.com/2010/05/android-cloud-to-device-messaging.html.

Table 7.2 SMS-related manifest permissions and their purpose

Phone-related permission	Purpose
`android.permission.READ_SMS`	Allow application to read SMS messages
`android.permission.RECEIVE_SMS`	Allow application to monitor incoming SMS messages
`android.permission.SEND_SMS`	Allow application to send SMS messages
`android.permission.WRITE_SMS`	Write SMS messages to the built-in SMS provider (not related to sending messages directly)

```
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<uses-permission android:name="android.permission.READ_SMS" />
<uses-permission android:name="android.permission.WRITE_SMS" />
<uses-permission android:name="android.permission.SEND_SMS" />
```

Along with sending text and data messages via `SmsManager`, you can create an SMS `BroadcastReceiver` to receive incoming SMS messages.

7.4.2 Receiving SMS messages

You can receive an SMS message programmatically by registering for the appropriate broadcast. To demonstrate how to receive SMS messages in this way with our TelephonyExplorer application, we'll implement a receiver, as shown in the following listing.

Listing 7.8 Creating an SMS-related `BroadcastReceiver`

```
public class SmsReceiver extends BroadcastReceiver {
    private static final String SMS_REC_ACTION =
      "android.provider.Telephony.SMS_RECEIVED";

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().
             equals(SmsReceiver.SMS_REC_ACTION)) {
            StringBuilder sb = new StringBuilder();
            Bundle bundle = intent.getExtras();
            if (bundle != null) {
                Object[] pdus = (Object[])
                  bundle.get("pdus");
                for (Object pdu : pdus) {
                    SmsMessage smsMessage =
                         SmsMessage.createFromPdu
                         ((byte[]) pdu);
                    sb.append("body - " + smsMessage.
                      getDisplayMessageBody());
                }
            }
            Toast.makeText(context, "SMS RECEIVED - "
              + sb.toString(), Toast.LENGTH_LONG).show();
        }
    }
}
```

1 Filter for action in receiver

2 Get pdus from Intent Bundle

3 Create SmsMessage from pdus

To react to an incoming SMS message, we again create a custom `BroadcastReceiver` by extending that class. Our receiver defines a local constant for the `Intent` action it wants to catch, in this case, `android.provider.Telephony.SMS_RECEIVED`.

Next, we filter for the action we want on the `onReceive` method ❶, and we get the SMS data from the `Intent` extras `Bundle` using the key `pdus` ❷. The `Bundle` is a hash that contains Android data types.

What's a PDU?

PDU, or protocol data unit, refers to one method of sending information along cellular networks. SMS messaging, as described in the 3rd Generation Partnership Project (3GPP) Specification, supports two different ways of sending and receiving messages. The first is text mode, which some phones don't support. Text mode encodes message content as a simple bit stream. The other is PDU mode, which not only contains the SMS message, but also metadata about the SMS message, such as text encoding, the sender, SMS service center address, and much more. To access this metadata, mobile SMS applications almost always use PDUs to encode the contents of a SMS message. For more information about PDUs and the metadata they provide, refer to the specification titled "Technical Realization of the Short Message Service (SMS)" which you can find at http://www.3gpp.org/ftp/Specs/html-info/23040.htm. This document, part of the 3GPP TS 23.040 Specification, is extremely technical but will help you with developing more sophisticated SMS applications.

For every `pdu Object` that we receive, we need to construct an `SmsMessage` by casting the data to a byte array ❸. After this conversion, we can use the methods in that class, such as `getDisplayMessageBody`.

> **NOTE** If you run the example shown in listing 7.8, you'll see that even though the receiver does properly report the message, the message still arrives in the user's inbox. Some applications might process specific messages themselves and prevent the user from ever seeing them; for example, you might implement a play-by-SMS chess program that uses text messages to report the other players' moves. To consume the incoming SMS message, call `abortBroadcast` from within your `onReceive` method. Note that your receiver must have a priority level higher than that of the inbox. Also, certain versions of the Android OS don't honor this request, so test on your target devices if this behavior is important to your app.

Congratulations! Now that you've learned how to send SMS messages programmatically, set permissions appropriately, and you can receive and work with incoming SMS messages, you can incorporate useful SMS features into your application.

7.5 Summary

Our trip through the Android telephony-related APIs covered several important topics. After a brief overview of some telephony terms, we examined Android-specific APIs.

You accessed telephony information with the TelephonyManager, including device and SIM card data and phone state. From there, we addressed hooking in a PhoneStateListener to react to phone state changes.

Besides retrieving data, you also learned how to dial the phone using built-in intents and actions, intercept outgoing phone calls, and format numbers with the PhoneNumberUtils class. After we covered standard voice usages, we looked at how to send and receive SMS messages using the SmsManager and SmsMessage classes.

In the next chapter, we'll turn to the specifics of interacting with notifications and alerts on the Android platform. We'll also revisit SMS and you'll learn how to notify users of events, such as an incoming SMS, by putting messages in the status bar, flashing a light, or even by making the phone vibrate.

8 Notifications and alarms

This chapter covers

- Building an SMS Notification application
- Working with Toasts
- Working with the NotificationManager
- Using Alarms and the AlarmManager
- Setting an Alarm

Today's cell phones are expected to be not only phones but personal assistants, cameras, music and video players, and instant-messaging clients, as well as do 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, whether in response to an SMS, a new voicemail, or an Alarm reminding them of a new appointment.

In this chapter, we're going to look at how to use the Android BroadcastReceiver and the AlarmManager to notify users of these sorts of events. First, we'll look at how to display quick, unobtrusive, and nonpersistent messages called Toasts, based on an event, such as receiving an SMS. Second, we'll talk about how to create persistent messages, LED flashes, phone vibrations, and other events to alert the user. These events are called Notifications. Finally, we'll look at how to trigger events by making Alarm events through the AlarmManager. Before we go too deeply into how notifications work, let's first create a simple example application.

8.1 Introducing Toast

For our example, you'll create a simple Receiver class that listens for an SMS text message. When a message arrives, the Receiver briefly pops up a message, called a Toast, to the user, with the content of the message. A Toast is a simple, nonpersistent message designed to alert the user of an event. Toasts are a great way to let a user know that a call is coming in, an SMS or email has arrived, or some other event has just happened. A Toast is not the same as a message, such as a status bar notification, which persists even when a phone is turned off or until the user selects the notification or the Clear Notification button.

In this example, we'll also use the Android SmsMessage class, but we won't go into depth about how SMS works in Android. For more information about how to work with SMS, see section 7.4 in chapter 7.

8.1.1 Creating an SMS example with a Toast

To look at how you can use a Toast, let's create a simple example. First create a project called SMSNotifyExample in Eclipse. You can use whatever package name you like, but for this chapter we'll use com.msi.manning.chapter8. Now that you've created the project, let's edit AndroidManifest.xml. You'll need to add several permission tags to allow Android to both receive and react to SMS messages. Make sure your AndroidManifest.xml file looks like the one shown in the following listing.

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.SMSNotifyExample2">
    <application android:icon="@drawable/icon">
        <activity android:name=".SMSNotifyExampleActivity"
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">
            <intent-filter>
            <action android:name=
"android.provider.Telephony.SMS_RECEIVED" />
            </intent-filter>
        </receiver>
    </application>
    <uses-permission android:name=
"android.permission.RECEIVE_SMS"></uses-permission>
    <uses-permission android:name=
"android.permission.READ_SMS"></uses-permission>
</manifest>
```

1 Define receiver, SMSNotify, with Intent filter

2 SMSNotifyExample acts as receiver

3 Define user permissions to allow SMS messages

The first thing we need to do is add an intent-filter ❶ to the AndroidManifest.xml file. For the intent-filter, we define SMSNotifyActivity, which is simply our Activity. The next class is SMSNotifyExample ❷, which will act as our receiver. Next, and perhaps more importantly, we need to add user permissions ❸ to allow incoming SMS messages to be received and read. The Android security model default has no permissions associated with applications; applications can essentially do nothing that might harm the device or the data on the device. To provide Android permission, you need to use one or more permissions, which you set in the manifest.xml file. In chapter 9, we'll go into greater detail about Android's security model, but with the plethora of emerging SMS viruses and exploits, Android's security model provides important protection against such attacks.

8.1.2 Receiving an SMS message

Now that you've set up the project, you need to write the code to capture the response when an SMS is received. First, create a simple Activity class called SMSNotify-Activity, as shown in the following listing.

Listing 8.2 SMS Activity for the SMSNotifyExample class

```
public class SMSNotifyExampleActivity extends Activity {

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

As you can see, there's little to this listing, in part because for this first example we're not doing much with the Activity. Later in this chapter, we'll build on this class. For now, let's create our Receiver class (see chapter 5 for more about Intent receivers), which will listen for the SMS message and fire off an action. The following listing shows the code for our SMSNotifyExample class.

Listing 8.3 A sample SMSNotifyExample

```
public class SMSNotifyExample extends BroadcastReceiver {

    private static final String LOG_TAG = "SMSReceiver";
    public static final int NOTIFICATION_ID_RECEIVED = 0x1221;
    static final String ACTION =
"android.provider.Telephony.SMS_RECEIVED";

    @Override
    public void onReceive(Context context, Intent intent) {

        if (intent.getAction().equals(SMSNotifyExample.ACTION)) {
            StringBuilder sb = new StringBuilder();

            Bundle bundle = intent.getExtras();
            if (bundle != null) {
                Object[] pdus = (Object[]) bundle.get("pdus");
```

❶ Build message to share to user

❷ Get PDUs from Intent Bundle

```
                for (Object pdu : pdus){
                SmsMessage messages =
 SmsMessage.createFromPdu((byte[]) pdu);
                sb.append("Received SMS\nFrom: ");
                sb.append(messages.getDisplayOriginatingAddress());
                sb.append("\n----Message----\n");
                sb.append( messages.getDisplayMessageBody());
                }
            }
            Log.i(SMSNotifyExample.LOG_TAG,
 "[SMSApp] onReceiveIntent: " + sb);
            Toast.makeText
(context, sb.toString(), Toast.LENGTH_LONG).show();
        }
    }
```

❸ Create SmsMessage from PDUs

❹ Display Toast

This should be easy to follow. First, we extend the SMSNotifyExample class using BroadcastReceiver, which allows the class to receive Intent classes. Then we create a String to hold the action that will be fired by the system when an SMS is received. After that, we create a simple method to notify the user that an SMS message has been received, and we parse the SMS message to show whom it was from and the content of the message ❶. Next, we need to capture the contents of the SMS message ❷. We do this by extracting the data from the *protocol data unit* or *PDUs*.

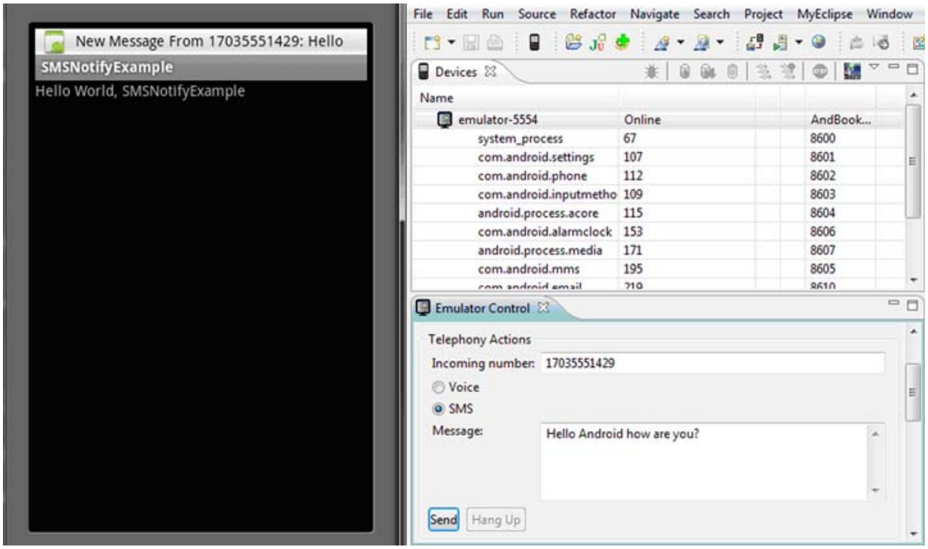
SMS messaging supports two different ways of sending and receiving messages. The one we're using is *text mode*, which happens to be unavailable on some phones and is simply an encoding of the bit stream represented by the PDU mode. The *PDU mode* is generally superior, because a PDU not only contains the SMS message, but also metadata about the SMS message, such as encoding, information about the sender, SMS service center, and much more. PDUs are almost always used in mobile SMS applications as the mechanism for handling SMS messages.

We also use a Bundle, which acts as an Android-specific hash map and accepts only primitives. We can look over the contents of the PDU and build an SmsMessage ❸, as well as extract key pieces from the PDU, such as the SMS sender's phone number, to display to the user in a Toast ❹.

Toast classes are transient little messages—they pop up and provide the user with quick information, without interrupting what the user is doing. In our code, we chain two methods together using the form makeText(Context context, CharSquence text, int duration).show(), where the first method contains a text view 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.

After you've finished 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.

NOTE In *Unlocking Android*, the first edition of this book, the code used the android.telephony.gsm.Smsmessage class for SMS messages. This class is now deprecated and you need to import android.telephony.Smsmessage.



New Message From 17035551429: Hello

SMSNotifyExample

Hello World, SMSNotifyExample

Name		
emulator-5554	Online	AndBook...
system_process	67	8600
com.android.settings	107	8601
com.android.phone	112	8602
com.android.inputmetho	109	8603
android.process.acore	115	8604
com.android.alarmclock	153	8606
android.process.media	171	8607
com.android.mms	195	8605
com.android.email	219	8610

Telephony Actions

Incoming number: 17035551429

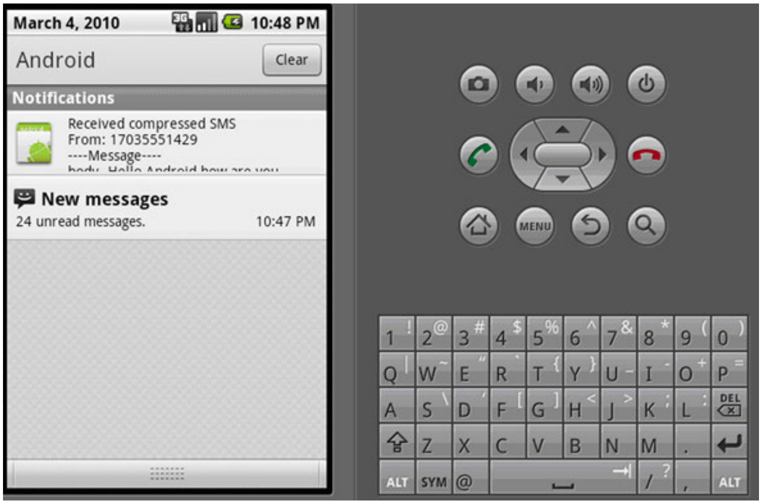
○ Voice

● SMS

Message: Hello Android how are you?

Figure 8.1 **A simple `Toast`, the `SMSNotifyExample`, shown running in the emulator**

To test our application, select the DDMS option in Eclipse. Now, in the Telephony Actions field, type a telephone number, for example, 17035551429. Select SMS, type a message in the Message field, and click Send. Your message should be sent to the emulator, and you should be able to see the emulator responding in the Eclipse console. A message should appear in the Android status bar at the top of the Android screen representation, as shown in figure 8.2.



March 4, 2010 10:48 PM

Android

Notifications

Received compressed SMS
From: 17035551429
----Message----
body: Hello Android how are you

New messages
24 unread messages. 10:47 PM

Figure 8.2 **Example of a `Toast` message being generated from an SMS message**

You've created a simple example and you know how to display a short message upon receiving an SMS and how to use the emulator to create an SMS. Now let's look at how to create a more persistent message that can also be used to set LEDs, play a sound, or something of that nature, to let the user know an event has occurred.

8.2 Introducing notifications

In the previous section, we showed how simple it is to create a quick, unobtrusive message to let the user know an SMS message has arrived. In this section, we're going to look at how to create a persistent notification that not only shows up in the status bar, but stays in a notification area until the user deletes it. To do that, we need to use the class `Notification`, because we want to do something more complex than `Toast` can offer us.

8.2.1 The Notification class

A notification[1] on Android can be many things, ranging from a pop-up message, to a flashing LED, to a vibration, but all these actions start with and are represented by the `Notification` class. The `Notification` class defines how you want to represent a notification to a user. This class has three constructors, one public method, and a number of fields. Table 8.1 summarizes the class.

Table 8.1 `Notification` fields

Access	Type	Method	Description
public	`int`	`audioStreamType`	Stream type to use when playing a sound
public	`int`	`defaults`	Defines which values should be taken from defaults
public	`int`	`deleteIntent`	The `Intent` to execute when user selects Clear All Notifications button
public	`flags`		
public	`int`	`ledARGB`	The color of the LED notification
public	`int`	`ledOffMS`	The number of milliseconds for the LED to be off between flashes
public	`int`	`ledOnMS`	The number of milliseconds for the LED to be on for each flash
public	`Int`	`number`	The number of events represented by this notification
public	`ContentURI`	`sound`	The sound to play

[1] To get your Android notifications on your Mac OS X, Linux, or Windows desktop, follow along here: http://www.readwriteweb.com/archives/how_to_get_android_notifications_on_your_computer_desktop.php.

Table 8.1 `Notification` fields *(continued)*

Access	Type	Method	Description
public	`RemoteViews`	`contentView`	View to display when the `statusBar-Icon` is selected in the status bar
public	`CharSequence`	`statusBarBalloonText`	Text to display when the `statusBar-Icon` is selected in the status bar
public	`PendingIntent`	`contentIntent`	The `Intent` to execute when the icon is clicked
public	`int`	`icon`	The resource ID of a drawable to use as the icon in the status bar
public	`Int`	`iconLevel`	The level of an icon in the status bar
public	`CharSequence`	`tickerText`	Text to scroll across the screen when this item is added to the status bar
public	`long[]`	`vibrate`	The vibration pattern to use
public	`long`	`when`	A timestamp for the notification

As you can see, the `Notification` class has numerous fields; it has to describe every way you can notify a user. Using a `Notification` is as simple as running this code:

```
int icon = R.drawable.icon_from_resources;
CharSequence tickerText = "Hello Android";
long when = System.currentTimeMillis();

CharSequence contentTitle = "New Message";
CharSequence contentText = "Hello Android how are you?";
Intent notificationIntent = new Intent();      // new intent
PendingIntent contentIntent = PendingIntent.getActivity(context, 0,
     notificationIntent, 0);
```

The next two lines initialize the `Notification` using the configurations shown in the previous snippet:

```
Notification notification = new Notification(icon, tickerText, when);
notification.setLatestEventInfo(context,
   contentTitle, contentText, contentIntent);
```

To send the `Notification`, all you have to do is enter the following code:

```
mNotificationManager.notify(NOTIFICATION_ID_RECEIVED, notification);
```

where the `notify` method wakes up a thread that performs the notification task you have defined.

8.2.2 Notifying a user of an SMS

Now let's take our previous example and edit it to change it from a `Toast` notification to a notification in the status bar. Before we do that, we'll make the application more interesting by adding icons to our resources directory. For this example, we're going to use the chat.png icon and the incoming.png icon. You can find these files in the downloaded code for this book, or you can get them from http://www.manning.com/ableson/. Simply drop them in the res/drawable directory to have Eclipse automatically register them for you in the R class.

Now let's edit the code. First, we'll edit the SMSNotifyActivity class so that when the Activity is called, it can find the Notification passed to it from the NotificationManager. After the Activity has run, SMSNotifyActivity can cancel it. The following listing provides the code you need for the new SMSNotifyActivity class.

Listing 8.4 A sample SMSNotifyActivity

```
public class SMSNotifyActivity extends Activity {
@Override
public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        NotificationManager nm = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);

                nm.cancel(R.string.app_name);


    }
}
```

As you can see, all we did was use the NotificationManager to look up the Notification, and then we used the cancel()method to cancel it. We could do more, such as set up a custom view, but for now we'll leave it as is.

Next, we need to edit the SMSNotifyExample to remove the Toast Notification and support a Notification to the status bar. The following listing shows the edits we need to make.

Listing 8.5 Updated SMSNotifyExample.java

```
   private static final String LOG_TAG = "SMSReceiver";
    public static final int NOTIFICATION_ID_RECEIVED = 0x1221;
    static final String ACTION = "android.provider.Telephony.SMS_RECEIVED";

    public void onReceive(Context context, Intent intent) {

       NotificationManager mNotificationManager =
 (NotificationManager) context.getSystemService           ❶ Get reference to
(Context.NOTIFICATION_SERVICE);                               NotificationManager
        if (intent.getAction().equals(SMSNotifyExample.ACTION)) {

            StringBuilder sb = new StringBuilder();
            String from = new String();
            String body = new String();

            Bundle bundle = intent.getExtras();
            if (bundle != null) {
                Object[] pdus = (Object[]) bundle.get("pdus");
                for (Object pdu : pdus){
                SmsMessage messages =
 SmsMessage.createFromPdu((byte[]) pdu);
                sb.append("Received compressed SMS\nFrom: ");
                sb.append(messages.getDisplayOriginatingAddress());
                from = messages.getDisplayOriginatingAddress();
                sb.append("\n----Message----\n");
                sb.append("body -" + messages.getDisplayMessageBody());
                body= messages.getDisplayMessageBody();

          Log.i(SMSNotifyExample.LOG_TAG,
 "[SMSApp] onReceiveIntent: " + sb);
            abortBroadcast();
                }}

                int icon = R.drawable.chat;                           2 Instantiate
                CharSequence tickerText =                               Notification
 "New Message From " + from + ": " + body;
                long when = System.currentTimeMillis();

                Notification notification =
 new Notification(icon, tickerText, when);
                CharSequence contentTitle = "New SMS Message";     3 Define
                CharSequence contentText = sb.toString();            message
                Intent notificationIntent = new Intent();            when
                PendingIntent contentIntent =                        notification
 PendingIntent.getActivity(context, 0, notificationIntent, 0);      bar
                                                                     expands
                notification.setLatestEventInfo
(context, contentTitle, contentText, contentIntent);
                mNotificationManager.notify
(NOTIFICATION_ID_RECEIVED, notification);                 <- 4 Send notification

}
```

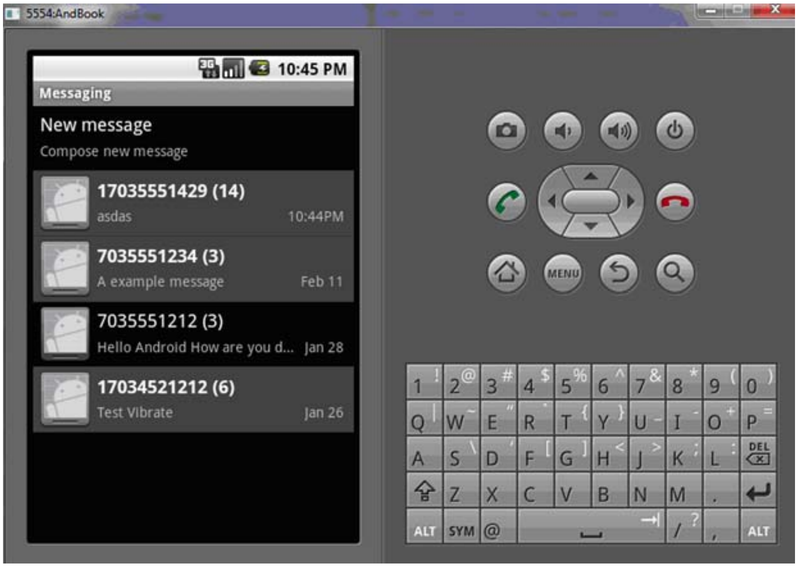
The first thing you'll notice is the addition of the NotificationManager 1, which handles all notifications. The second major change to the code is the addition of the required fields that notifications need, such as icon, when, and tickerText 2.

Note the variable tickerMessage. The tickerMessage will hold the contents of the SMS message that we want to scroll in the notification bar. Though it's possible to style the notification message in a variety of ways, the notification bar in general supplies little space and is best used for short messages.

Next, we instantiate the Notification with the required fields. After that, we create an Intent for the Notification's expanded message 3. Notifications generally have two parts—a message that appears in the notification bar, then a message that appears in the expanded view of the Notifications drop-down view when you open the notification bar. The message for the expanded Notifications window is provided via the method setLatestEventInfo, like so:

```
Public void setLatestEventInfo(Context context,
 CharSequence contentTitle, CharSequence contentT006xt,
 PendingIntent contentIntent);
```

Finally, we use the notify() method 4 from the NotificationManager to broadcast our Notification to the application.



5554:AndBook

10:45 PM

Messaging

New message
Compose new message

17035551429 (14)
asdas 10:44PM

7035551234 (3)
A example message Feb 11

7035551212 (3)
Hello Android How are you d... Jan 28

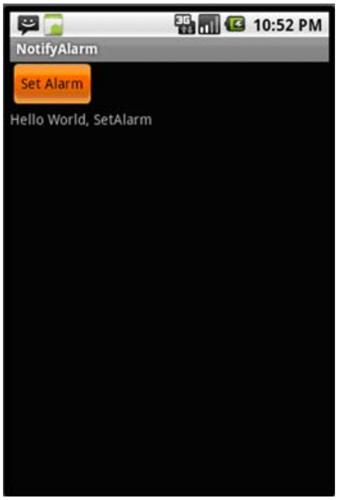
17034521212 (6)
Test Vibrate Jan 26

Figure 8.3 Using the Android DDMS to send an SMS message to the application

Now if you run the application and then open the DDMS and pass an SMS message as you did earlier, you should see the new `Notification` in the status bar. The message displays each line for a short interval until the message is fully displayed. You should also see a new icon pop up in the status bar that indicates a new SMS message, as shown in figure 8.3.

When you've sent the message, click the New Messages icon; a bar should drop down from it. Click the bar and drag it down to the bottom of the screen. The default view of the SMS inbox for Android opens, as shown in figure 8.4.

You could do a lot more with this demo, such as creating a better UI[2] or making the SMS inbox more feature-rich. You could even make the application play a sound when a message arrives. Even so, in this example, we've looked at everything you need to know to start working with notifications. In the next section, we're going to look at `Notification`'s close relative, the `Alarm`.



10:52 PM

NotifyAlarm

Set Alarm

Hello World, SetAlarm

Figure 8.4 The expanded SMS inbox displaying the `contentIntent` and `appIntent`

[2] Here is a great article about improving Android app quality and creating better UI experiences: http://android-developers.blogspot.com/2010/10/improving-app-quality.html.

8.3 Introducing Alarms

In Android, Alarms allow you to schedule your application to run at some point in the future. Alarms can be used for a wide range of applications, from notifying a user of an appointment to something more sophisticated, such as having an application start, check for software updates, and then shut down. An Alarm works by registering an Intent with the Alarm; at the scheduled time, the Alarm broadcasts the Intent. Android automatically starts the targeted application, even if the Android handset is asleep.

Android manages all alarms somewhat like it manages the NotificationManager—via an AlarmManager class. The AlarmManager has four methods: cancel, set, setRepeating, and setTimeZone, as shown in table 8.2.

Table 8.2 AlarmManager public methods

Returns	Method and description
void	cancel(PendingIntent intent) Remove alarms with matching Intent
void	set(int type, long triggerAtTime, PendingIntent operation) Set an Alarm
void	setRepeating(int type, long triggerAtTime, long interval, PendingIntent operation) Set a repeating Alarm
void	setTimeZone(String TimeZone) 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 easy, like most things in Android. In the next example, you'll create a simple application that sets an Alarm when a button is pressed. When the Alarm is triggered, it will pass back a simple Toast to inform us that the Alarm has been fired.

8.3.1 Creating a simple alarm example

In this next example, you're going to create an Android project called SimpleAlarm that has the package name com.msi.manning.chapter8.simpleAlarm, the application name SimpleAlarm, and the Activity name GenerateAlarm. In this project, we'll use another open source icon, which you can find at http://www.manning.com/ableson/ 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.

The next thing to do is edit the AndroidManifest.xml to have a receiver (we'll create that soon) called AlarmReceiver, as shown in the following listing.

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" />
    </application>
</manifest>
```

Now we edit the string.xml file in the values directory and add two new strings:

```
<string name="set_alarm_text">Set Alarm</string>
<string name="alarm_message">Alarm Fired</string>
```

We'll use this as the value of the button in our layout. Next, we need to add a new button to our layout, so edit the main.xml file to add a new button, like this:

```
<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>
```

We're ready to create a new class that will act as the Receiver for the Notification the Alarm will generate. In this case, we're going to be generating a Toast-style Notification to let the user know that the Alarm has been triggered. Now, create a new class as shown in the following listing. This class waits for the Alarm to broadcast to the AlarmReceiver and then generates a Toast.

Listing 8.7 AlarmReceiver.java

```
public class AlarmReceiver extends BroadcastReceiver  {
    public void onReceiveIntent(
        Context context, Intent intent) {
        Toast.makeText
(context, R.string.app_name, Toast.LENGTH_SHORT).show();
    }
    @Override
    public void onReceive(Context context, Intent intent) {
    }
}
```

Broadcast Toast when Intent is received

Next, we need to edit the SimpleAlarm class to create a button widget (as we 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`. The following listing shows what the finished class should look like.

Listing 8.8 SimpleAlarm.java

```
public class GenerateAlarm extends Activity
 {
    Toast mToast;
    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        Button button = (Button)findViewById(R.id.set_alarm_button);
        button.setOnClickListener(this.mOneShotListener);
    }
    private OnClickListener mOneShotListener = new OnClickListener()
 {
        public void onClick(View v)                                        ❶
 {
            Intent intent =
 new Intent(GenerateAlarm.this, AlarmReceiver.class);
            PendingIntent appIntent =                            ❷ Create Intent
            PendingIntent.getBroadcast(GenerateAlarm.this,         to fire when
0, intent, 0);                                                     Alarm goes off
            Calendar calendar = Calendar.getInstance();
            calendar.setTimeInMillis(System.currentTimeMillis());
            calendar.add(Calendar.SECOND, 30);
            AlarmManager am =                                 ❸ Create
 (AlarmManager)getSystemService(ALARM_SERVICE);                 AlarmManager
            am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
            appIntent);
            if (GenerateAlarm.this.mToast != null) {
                GenerateAlarm.this.mToast.cancel();
            }
            GenerateAlarm.this.mToast = Toast.makeText(GenerateAlarm.this,
            R.string.alarm_message, Toast.LENGTH_LONG);
            GenerateAlarm.this.mToast.show();
        }
    };
}
```

As you can see, this class is pretty simple. We create a `Button` to trigger our `Alarm` ❶. Next, we create an inner class for our `mOneShotListener`. Then, we create the `Intent` to be triggered when the `Alarm` actually goes off ❷. In the next section of code, we use the `Calendar` class to help us calculate the number of milliseconds from the time the button is pressed, which we'll use to set the `Alarm`.

Now we've done everything necessary to create and set the `Alarm`. We create the `AlarmManager` ❸ and then call its `set()` method to set the `Alarm`. To see a little more detail of what's going on in the application, take a look at these lines of code:

```
AlarmManager am = (AlarmManager)getSystemService(ALARM_SERVICE);
       am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(), intent);
```

These lines are 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 AlarmManager currently supports four Alarm types, as shown in table 8.3.

Table 8.3 **AlarmManager Alarm types**

Type	Description
ELAPSED_REALTIME	Alarm time in 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

You can use multiple types of alarms, depending on your requirements. The RTC_WAKEUP, for example, sets the Alarm time in milliseconds; when the Alarm goes off, it'll wake up the device from sleep mode for you, as opposed to RTC, which won't.

The next parameter we pass to the method is the amount of time, in milliseconds, that we want to elapse, after which we want the alarm to be triggered. The following snippet shows how to set this sequence of events:

```
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. Now, if you build the application and run it in the emulator, you should see something like the screen shown in figure 8.5.

Clicking the Set Alarm button will set the alarm; after 30 seconds, you should see something like figure 8.6, displaying the Toast message.

8.3.2 Using notifications with Alarms

Creating an Alarm is pretty easy in Android, but what might make more sense would be for that Alarm to trigger a Notification in the status bar. To do that, you need to add a NotificationManager and generate a Notification. We've created a new method to add to listing 8.8 called showNotification, which takes four parameters and creates our Notification:

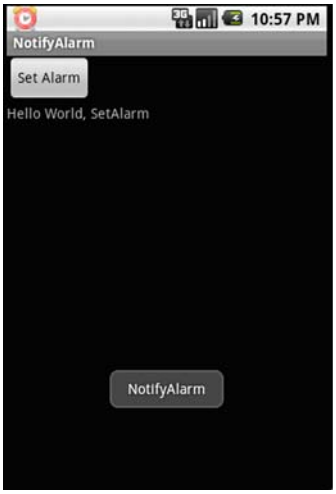


Figure 8.5 Example of the SimpleAlarm application running in the emulator

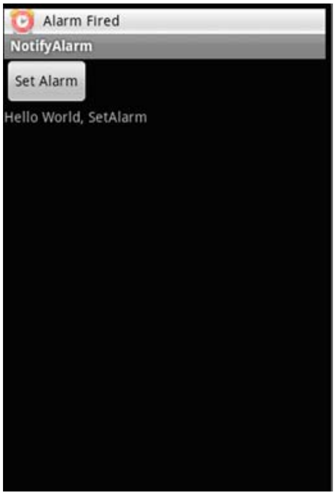


Figure 8.6 After the `Alarm` runs, the application shows a simple `Toast` message.

```
    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  );
    }
```

Much of this code is similar to the `SMSNotifyExample` code. To add it to your `SimpleAlarm`, edit listing 8.8 to look like listing 8.9, where the only other things we've done are to import the `Notification` and `NotificationManager` to the code, add the private variables `nm` and `ApplicationID`, and make a call to `showNotification()`, right after the `Toast`.

Listing 8.9 SetAlarm.java

```
public class SetAlarm extends Activity {
    private NotificationManager nm;
        Toast mToast;
    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        this.nm = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
        Button button = (Button) findViewById(R.id.set_alarm_button);
        button.setOnClickListener(this.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);
        this.nm.notify(this.YOURAPP_NOTIFICATION_ID, notif);
    }
    private OnClickListener mOneShotListener = new OnClickListener() {
        public void onClick(View v) {
            Intent intent = new Intent(SetAlarm.this, AlarmReceiver.class);
            PendingIntent appIntent =
     PendingIntent.getBroadcast(SetAlarm.this, 0,
               intent, 0);
            Calendar calendar = Calendar.getInstance();
            calendar.setTimeInMillis(System.currentTimeMillis());
            calendar.add(Calendar.SECOND, 30);
            AlarmManager am = (AlarmManager)
                getSystemService(Context.ALARM_SERVICE);
            am.set(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
                appIntent);
            showNotification(R.drawable.alarm, R.string.alarm_message,
                R.string.alarm_message, false);
        }
    };
  }
}
```

If you run the code and click Set Alarm, you should see the Alarm Notification in the status bar. You could easily edit this code to take in parameters for time and date, have it show different Intents when the icons are clicked, and so on.

As you can see from this example, Android alarms and the AlarmManager are straightforward, and you should be able to easily integrate them into your applications.

8.4 Summary

In this chapter, we've looked at three separate but related items: Toast, Notification, and Alarm. You learned that for simple, nonpersistent messages, the Toast class provides an easy and convenient way to alert the user. We also looked at how to use the

NotificationManager to generate simple to relatively complex notifications. Then you used the Notification class to present a Notification to the user by building a simple example that displays a message in the status bar, vibrates a phone, or even flashes an LED when an SMS messages arrives in the inbox.

We also looked at how to set an Alarm to cause an application to start or take some action in the future, including waking the system from sleep mode. Finally, we talked about how to trigger a Notification from an Alarm. Though the code presented in these simple examples gives you a taste of what can be done with notifications and alarms, both have broad applications limited only by your imagination.

Now that you have an understanding of how to work with the Notification and Alarm classes, we're going to move on a discussion of graphics and animation. In chapter 9, you'll learn the basic methods of generating graphics in Android, how to create simple animations, and even how to work with OpenGL to generate stunning 3D graphics.

9 Graphics and animation

This chapter covers

- Drawing graphics in Android
- Applying the basics of OpenGL for embedded systems (ES)
- Animating with Android

One of the main features of Android that you should've picked up on by now is how much easier it is to develop Android applications than it is to use other mobile application platforms. This ease of use is especially apparent when you're creating visually appealing UIs and metaphors, but there's a limit to what you can do with typical Android UI elements (such as those we discussed in chapter 3). In this chapter, we'll look at how to create graphics using Android's Graphics API, develop animations, and explore Android's support for the OpenGL standard. (To see examples of what you can do with Android's graphics platform, go to http://www.omnigsoft.com/Android/ADC/readme.html.)

First, we're going to show you how to draw simple shapes using the Android 2D Graphics API, using Java and then XML to describe 2D shapes. Next, we'll look at making simple animations using Java and the Graphics API to move pixels around, and then using XML to perform a frame-by-frame animation. Finally, we'll look at Android's support of the OpenGL ES API, make a simple shape, and then make a more complex, rotating, three-dimensional shape.

If you've ever worked with graphics in Java, you'll likely find the Graphics API and how graphics work in Android familiar. If you've worked with OpenGL, you'll find that Android's implementation of OpenGL ES will often let you port your previous code into Android, with few changes. You must remember though, that cell phones don't have the graphical processing power of a desktop. Regardless of your experience, you'll find the Android Graphics API both powerful and rich, allowing you to accomplish even some of the most complex graphical tasks.

9.1 Drawing graphics in Android

In this section, we'll cover Android's graphical capabilities and show you examples of how to make simple 2D shapes. We will be applying the `android.graphics` package (see http://code.google.com/android/reference/android/graphics/package-summary.html), which provides all the low-level classes you need to create graphics. The graphics package supports such things as bitmaps (which hold pixels), canvas (what your draw calls draw on), primitives (such as rectangles or text), and paint (which you use to add color and styling). Generally, you use Java to call the Graphics API to create complex graphics.

To demonstrate the basics of drawing a shape with Java and the Graphics API, let's look at a simple example in the following listing, where we'll draw a rectangle.

Listing 9.1 Listing 9.1 simlepshape.java

```
       package com.msi.manning.chapter9.SimpleShape;
       public class SimpleShape extends Activity {
    @Override
    protected void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(new SimpleView(this));
    }
    private static class SimpleView extends View {          1 Create View
        private ShapeDrawable mDrawable =
            new ShapeDrawable();                         Create ShapeDrawable
                                                       2 to hold Drawable
        public SimpleView(Context context) {
            super(context);
            setFocusable(true);                            3 Create Rectangle,
            this.mDrawable =                                 assign to mDrawable
                new ShapeDrawable(new RectShape());
            this.mDrawable.getPaint().setColor(0xFFFF0000);
        }
        @Override
        protected void onDraw(Canvas canvas) {
            int x = 10;
            int y = 10;
            int width = 300;
            int height = 50;
            this.mDrawable.setBounds(x, y, x + width, y + height);
            this.mDrawable.draw(canvas);
            y += height + 5;
       }
    }
}
```

Drawing a new shape is simple. First, we need to import the necessary packages ❶, including graphics. Then we import ShapeDrawable, which will support adding shapes to our drawing, and then shapes, which supports several generic shapes, including RectShape, that we'll use. Next, we need to create a view ❷, and then a new ShapeDrawable to add our Drawable to ❸. After we have a ShapeDrawable, we can assign shapes to it. In our code, we use the RectShape, but we could've used OvalShape, PathShape, RectShape, RoundRectShape, or Shape. We then use the onDraw() method to draw the Drawable on the Canvas. Finally, we use the Drawable's setBounds() method to set the boundary (a rectangle) in which we'll draw our rectangle using the draw() method.

When you run listing 9.1, you should see a simple red rectangle like the one shown in figure 9.1.

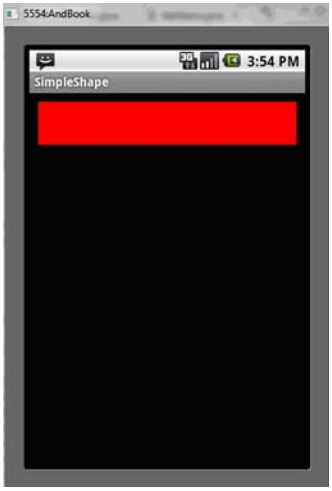


Figure 9.1 A simple red rectangle drawn using Android's Graphics API

Another way to do the same thing is through XML. Android allows you to define shapes to draw in an XML resource file.

9.1.1 Drawing with XML

With Android, you can create simple drawings using an XML file approach. You might want to use XML for several reasons. One basic reason is because it's simple to do. Also, it's worth keeping in mind that graphics described by XML can be programmatically changed later, so XML provides a simple way to do initial design that isn't necessarily static.

To create a drawing with XML, create one or more Drawable objects, which are defined as XML files in your drawable directory, such as res/drawable. The XML you need to create a simple rectangle looks like the code shown in the following listing.

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 is a rectangle, but you can change the shape by using the type tag and selecting the value oval, rectangle, line, or arc. To use your XML shape, you need to reference it in a layout, as shown in the following listing. The layout resides in res/layout.

Listing 9.3 xmllayout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
     <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content">
     <ImageView android:layout_width="fill_parent"
        android:layout_height="50dip"
        android:src="@drawable/simplerectangle" />
     </LinearLayout>
</ScrollView>
```

Now all you need to do is create a simple `Activity`, where you place your UI in a `contentView`, as shown in the following listing.

Listing 9.4 XMLDraw.java

```
public class XMLDraw extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.xmldrawable);
    }
}
```

If you run this code, it'll draw a simple rectangle. You can make more complex drawings or shapes by stacking or ordering your XML drawables, and you can include as many shapes as you want or need, depending on space. Let's explore what multiple shapes might look like next.

9.1.2 *Exploring XML drawable shapes*

One way to draw multiple shapes with XML is to create multiple XML files that represent different shapes. A simple way to do this is to change your xmldrawable.xml file to look like the following listing, which adds a number of shapes and stacks them vertically.

Listing 9.5 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_3" />
    </LinearLayout>
</ScrollView>
```

Finally, you need to add the shapes shown in listings 9.6 through 9.9 into the res/drawable folder.

Listing 9.6 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="#FFFFFFFF"/>
</shape>
```

In the previous listing, we're using an oval. We've added a tag called `padding`, which allows us to define padding or space between the object and other objects in the UI. We're also using the tag called `stroke`, which allows us to define the style of the line that makes up the border of the oval (see the following listing).

Listing 9.7 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="#FFFFFFFF"
        android:dashWidth="1dp" android:dashGap="2dp" />
    <padding android:left="7dp" android:top="7dp"
        android:right="7dp" android:bottom="7dp" />
    <corners android:radius="4dp" />
</shape>
```

With this shape, we're generating another rectangle, but this time (the next listing) we introduce the tag `corners`, which allows us to make rounded corners with the attribute `android:radius`.

Listing 9.8 Shape3.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 the next listing, we create a shape of the type `line` with a `size` tag using the `android:height` attribute, which allows us to describe the number of pixels used on the vertical to size the line.

Listing 9.9 line.xml

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
type="line" >
    <solid android:color="#FFFFFFFF"/>
    <stroke android:width="1dp" android:color="#FFFFFFFF"
        android:dashWidth="1dp" android:dashGap="2dp" />
    <padding android:left="1dp" android:top="25dp"
        android:right="1dp" android:bottom="25dp" />
    <size android:height="23dp" />
</shape>
```

If you run this code, you should see something like figure 9.2.

As you can see, drawing with Android is straightforward, and Android provides the ability for developers to programmatically draw anything they need. In the next section, we're going to look at what you can draw with Android's animation capabilities.

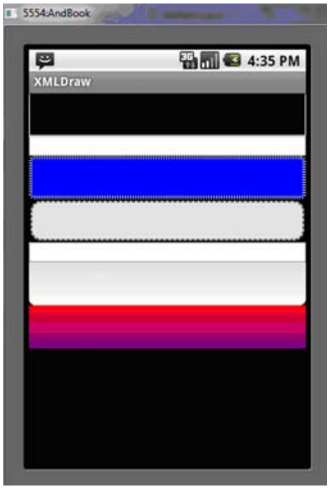


Figure 9.2 **Various shapes drawn using XML**

9.2 Creating animations with Android's Graphics API

If a picture says a thousand words, then an animation must speak volumes. Android supports multiple methods of creating animation, including through XML, as you saw in chapter 3, or via Android's XML frame-by-frame animations using the Android Graphics API, or via Android's support for OpenGL ES. In this section, you're going to create a simple animation of a bouncing ball using Android's frame-by-frame animation.

9.2.1 Android's frame-by-frame animation

Android allows you to create simple animations by showing a set of images one after another to give the illusion of movement, much like stop-motion film. Android sets each frame image as a drawable resource; the images are then shown one after the other in the background of a View. To use this feature, you define a set of resources in a XML file and then call AnimationDrawable start().

To demonstrate this method for creating an animation, you need to download this project from the Google code repository so you can get the images. The images for this exercise are six representations of a ball bouncing. Next, create a project called XMLanimation. Then create a new directory called /anim under the /res resources directory. Place all the images for this example in the /drawable directory. Now create an XML file called Simple_animation.xml that contains the code shown in the following listing.

Listing 9.10 Simple_animation.xml

```
<?xml version="1.0" encoding="utf-8"?>
 <animation-list xmlns:android="http://schemas.android.com/apk/res/android"
 id="selected" android:oneshot="false">
  <item android:drawable="@drawable/ball1" android:duration="50" />
  <item android:drawable="@drawable/ball2" android:duration="50" />
  <item android:drawable="@drawable/ball3" android:duration="50" />
  <item android:drawable="@drawable/ball4" android:duration="50" />
  <item android:drawable="@drawable/ball5" android:duration="50" />
  <item android:drawable="@drawable/ball6" android:duration="50" />
    </animation-list>
```

The XML file defines the list of images to be displayed for the animation. The XML <animation-list> tag contains the tags for two attributes: drawable, which describes the path to the image, and duration, which describes the length of time to show the image, in nanoseconds. Now that you've created the animation XML file, edit the main.xml file to look like the following listing.

Listing 9.11 main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
      <ImageView android:id="@+id/simple_anim"
          android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:gravity="center"
          android:layout_centerHorizontal="true"
           />
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hello World, XMLAnimation"
    />
</LinearLayout>
```

All we've done to the file is added an ImageView tag that sets up the layout for our Image-View. Finally, create the code to run the animation, which is shown in the following listing.

Listing 9.12 xmlanimation.java

```
public class XMLAnimation extends Activity
{
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        ImageView img =
        (ImageView)findViewById(R.id.simple_anim);
        img.setBackground(R.anim.simple_animation);
        MyAnimationRoutine mar =
        new MyAnimationRoutine();
        MyAnimationRoutine2 mar2 =
        new MyAnimationRoutine2();
        Timer t = new Timer(false);
        t.schedule(mar, 100);
        Timer t2 = new Timer(false);
        t2.schedule(mar2, 5000);
    }
    class MyAnimationRoutine extends TimerTask {
        @Override
        public void run() {
            ImageView img = (ImageView) findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
           img.getBackground();
            frameAnimation.start();
        }
    }
    class MyAnimationRoutine2 extends TimerTask {
        @Override
        public void run() {
            ImageView img = (ImageView) findViewById(R.id.simple_anim);
            AnimationDrawable frameAnimation = (AnimationDrawable)
           img.getBackground();
            frameAnimation.stop();
        }
    }
}
```

❶ **Allow wait time before starting Animation**

Listing 9.12 might be slightly confusing because we've used the TimerTask classes. Because we can't control the animation from within the OnCreate method, we need to create two subclasses that call Animation-Drawable's start and stop methods. The first subclass, MyAnimationRoutine, extends TimerTask ❶ and calls the frame-Animation.start() method for the AnimationDrawable bound to the ImageView background. If you run the project now, you should see something like figure 9.3.

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

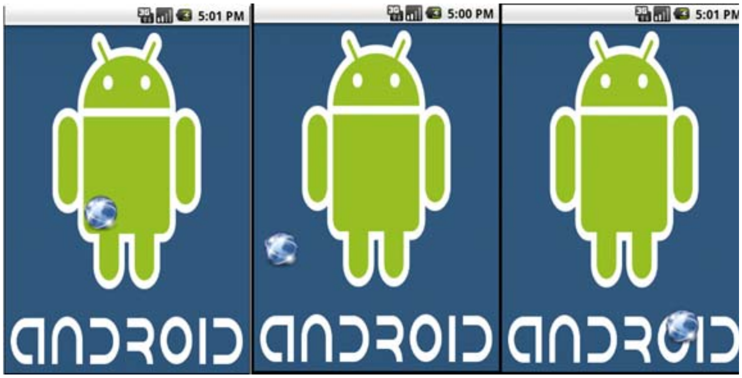


Figure 9.3 **Making a ball bounce using an Android XML animation**

movie, but to create more sophisticated animations programmatically you need to use Android's 2D and 3D graphics abilities. In the next section, we'll do just that.

9.2.2 *Programmatically creating an animation*

In the previous section, we used Android's frame-by-frame animation capabilities to show a series of images in a loop that gives the impression of movement. In this next section, we're going to programmatically animate a globe so that it moves around the screen.

To create this animation, we're going to animate a graphics file (a PNG file) with a ball that appears to be bouncing around inside our Android viewing window. We'll create a `Thread` in which our animation will run, and a `Handler` that'll help communicate messages back to our program that reflect the changes in the state of our animation. We'll use this same approach in section 9.3 when we talk about OpenGL ES. You'll find this approach is useful for creating most complex graphics applications and animations.

CREATING THE PROJECT

In this section, we'll look at a simple animation technique that uses an image bound to a sprite. The image moves that sprite around the screen to give the appearance of a bouncing ball. To get started, create a new project called BouncingBall with a `BounceActivity`. You can copy and paste the code in the following listing for the BounceActivity.java file.

Listing 9.13 BounceActivity.java

```
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();
```

1 Create a unique identifier

2 Create a handler

```
                         break;
               }
               super.handleMessage(msg);
          }
     };
     @Override
     public void onCreate(Bundle icicle) {
          super.onCreate(icicle);
          this.requestWindowFeature(Window.FEATURE_NO_TITLE);
          this.myBounceView = new BounceView(this);            ❸ Create the view
          this.setContentView(this.myBounceView);
          new Thread(new RefreshRunner()).start();
     }
     class RefreshRunner implements Runnable {
          public void run() {                                  ❹ Run the animation
               while (!Thread.currentThread().isInterrupted()) {
                    Message message = new Message();
                    message.what = BounceActivity.GUIUPDATEIDENTIFIER;
                    BounceActivity.this.myGUIUpdateHandler
.sendMessage(message);
                    try {
                         Thread.sleep(100);
                    } catch (InterruptedException e) {
                         Thread.currentThread().interrupt();
                    }
               }
          }
     }
}
```

First, we import the `Handler` and `Message` classes, and then we create a unique identifier to allow us to send a message back to our program to update the view in the main thread. We need to send a message telling the main thread to update the view each time the child thread has finished drawing our ball. Because different messages can be thrown by the system, we need to guarantee the uniqueness of our message to our handler by creating a unique identifier called `GUIUPDATEIDENTIFIER` ❶. Next, we create the `Handler` that'll process our messages to update the main view ❷. A `Handler` allows us to send and process `Message` classes and `Runnable` objects associated with a thread's message queue. Handlers are associated with a single thread and its message queue. We'll use the `Handler` to allow our objects running a thread to communicate changes in state back to the program that spawned them, or vice versa.

> **NOTE** For more information about handling long-running requests in your applications, see http://developer.android.com/reference/android/app/Activity.html.

We set up a `View` ❸ and create the new thread. Finally, we create a `RefreshRunner` inner class implementing `Runnable` that'll run unless something interrupts the thread, at which point a message is sent to the `Handler` to call its `invalidate()` method ❹. The `invalidate` method invalidates the `View` and forces a refresh.

You've got your new project. Now you need to create the code that'll do your animation and create a View.

MAKING ANIMATION HAPPEN

We're going to use an image of a globe, which you can obtain at http://www.manning.com/AndroidinActionSecondEdition. Alternatively, you can use any other PNG file you'd like to use. We also want to have the Android logo as our background, which you can find with the source code downloads. Make sure to drop the images under res/drawable/. Next, create a Java file called BounceView, copy the code from the following listing, and paste it into your editor.

Listing 9.14 BounceView.java

```
public class BounceView extends View {
     protected Drawable mySprite;
     protected Point mySpritePos = new Point(0,0);
     protected enum HorizontalDirection {LEFT, RIGHT} ;
     protected enum VerticalDirection {UP, DOWN} ;
     protected HorizontalDirection myXDirection =
HorizontalDirection.RIGHT;
     protected VerticalDirection myYDirection = VerticalDirection.UP;
     public BounceView(Context context) {
          super(context);
this.setBackground(this.getResources().getDrawable(R.drawable.android));
this.mySprite =
     this.getResources().getDrawable(R.drawable.world);
     }
     @Override
     protected void onDraw(Canvas canvas) {
this.mySprite.setBounds(this.mySpritePos.x,
     this.mySpritePos.y,
     this.mySpritePos.x + 50, this.mySpritePos.y + 50);
          if (mySpritePos.x >= this.getWidth() –
mySprite.getBounds().width()) {
               this.myXDirection = HorizontalDirection.LEFT;
          } else if (mySpritePos.x <= 0) {
               this.myXDirection = HorizontalDirection.RIGHT;
          }
          if (mySpritePos.y >= this.getHeight() –
mySprite.getBounds().height()) {
               this.myYDirection = VerticalDirection.UP;
          } else if (mySpritePos.y <= 0) {
               this.myYDirection = VerticalDirection.DOWN;
          }
          if (this.myXDirection ==
HorizontalDirection.RIGHT) {
               this.mySpritePos.x += 10;
          } else {
               this.mySpritePos.x -= 10;
          }

          if (this.myYDirection ==
                                   VerticalDirection.DOWN) {
               this.mySpritePos.y += 10;
```

1 Get image file and map it to sprite

2 Set the bounds of the globe

3 Move ball left or right, up or down

4 Check if ball is trying to leave screen

```
          } else {
               this.mySpritePos.y -= 10;
          }
          this.mySprite.draw(canvas);
     }
}
```

In this listing, we do all the real work of animating our image. First, we create a `Drawable` to hold our globe image and a `Point` that we use to position and track our globe as we animate it. Next, we create enumerations (`enums`) to hold directional values for horizontal and vertical directions, which we'll use to keep track of the moving globe. Then we map the globe to the `mySprite` variable and set the Android logo as the background for our animation ❶.

Now that we've done the setup work, we create a new `View` and set all the boundaries for the `Drawable` ❷. After that, we create simple conditional logic that detects whether the globe is trying to leave the screen; if it starts to leave the screen, we change its direction ❸. Then we provide simple conditional logic to keep the ball moving in the same direction if it hasn't encountered the bounds of the `View` ❹. Finally, we draw the globe using the `draw` method. If you compile and run the project, you should see the globe bouncing around in front of the Android logo, as shown in figure 9.4.

Though the simple `Animation` that we created is not too exciting, you could—with a little extra work—leverage the key concepts (dealing with boundaries, moving around drawables, detecting changes, dealing with threads, and so on) to create something like the Google Lunar Lander example game or even a simple version of Asteroids. If you want more graphics power and want to easily work with 3D objects to create things such as games or sophisticated animations, read the next section on OpenGL ES.

9.3 Introducing OpenGL for Embedded Systems

One of the most interesting features of the Android platform is its support of *OpenGL for Embedded Systems,*[1] or *OpenGL ES*. OpenGL ES is the embedded systems version of the

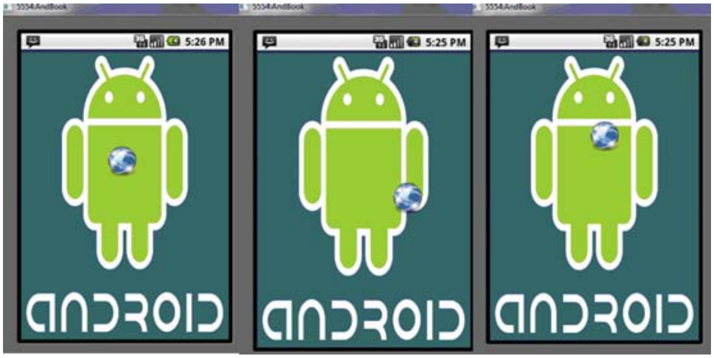


Figure 9.4 A simple animation of a globe bouncing in front of the Android logo

[1] Here is a good series of articles showing how to use OpenGL ES on Android: http://blog.jayway.com/author/pererikbergman.

popular OpenGL standard, which defines a cross-platform and cross-language API for computer graphics. The OpenGL ES API doesn't support the full OpenGL API, and much of the OpenGL API has been stripped out to allow OpenGL ES to run on a variety of mobile phones, PDAs, video game consoles, and other embedded systems. OpenGL ES was originally developed by the Khronos Group, an industry consortium. You can find the most current version of the standard at http://www.khronos.org/opengles/.

OpenGL ES is a fantastic API for 2D and 3D graphics, especially for graphically intensive applications such as games, graphical simulations, visualizations, and all sorts of animations. Because Android also supports 3D hardware acceleration, developers can make graphically intensive applications that target hardware with 3D accelerators.

Android 2.1 supports the Open GL ES 1.0 standard, which is almost equivalent to the OpenGL 1.3 standard. If an application can run on a computer using OpenGL 1.3, it should be possible to run it on Android, but you need to consider the hardware specifications of your Android handset. Though Android offers support for hardware acceleration, some handsets and devices running Android have had performance issues with OpenGL ES in the past. Before you embark on a project using OpenGL, consider the hardware you're targeting and do extensive testing to make sure that you don't overwhelm your hardware with OpenGL graphics.

Because OpenGL and OpenGL ES are such broad topics, with whole books dedicated to them, we'll cover only the basics of working with OpenGL ES and Android. For a much deeper exploration of OpenGL ES, check out the specification and the OpenGL ES tutorial at http://www.zeuscmd.com/tutorials/opengles/index.php. After reading this section on Android support for OpenGL ES, you should have enough information to follow a more in-depth discussion of OpenGL ES, and you should be able to port your code from other languages (such as the tutorial examples) into the Android framework. If you already know OpenGL or OpenGL ES, then the OpenGL commands will be familiar; concentrate on the specifics of working with OpenGL on Android.

NOTE An excellent book on OpenGL and Java 3D programming is *Java 3D Programming* by Daniel Selman, which is available at http://www.manning.com/selman/.

9.3.1 Creating an OpenGL context

Keeping in mind the comments we made in the introduction to this section, let's apply the basics of OpenGL ES to create an `OpenGL-Context` and a `Window` that we can draw in. Much of this task will seem overly complex compared to Android's Graphics API. The good news is that you have to do this setup work only once. That being said, we'll use the following general processes for working with OpenGL ES in Android:

1. Create a custom `View` subclass.
2. Get a `handle` to an `OpenGLContext`, which provides access to Android's OpenGL ES functionality.

3 In the View's onDraw() method, use the handle to the GL object and then use its methods to perform any GL functions.

Following these basic steps, first we'll create a class that uses Android to create a blank surface to draw on. In section 9.3.2, we'll use OpenGL ES commands to draw a square and an animated cube on the surface. To start, open a new project called OpenGLSquare and create an Activity called OpenGLSquare, as shown in the following listing.

Listing 9.15 OpenGLSquare.java

```
public class SquareActivity extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(new DrawingSurfaceView(this));
    }
    class DrawingSurfaceView extends SurfaceView implements
    SurfaceHolder.Callback  {
        public SurfaceHolder mHolder;
        public DrawingThread mThread;

        public DrawingSurfaceView(Context c) {
            super(c);
            init();
        }
        public void init() {
            mHolder = getHolder();
            mHolder.addCallback(this);
            mHolder.setType(SurfaceHolder.SURFACE_TYPE_GPU);
        }
        public void surfaceCreated(SurfaceHolder holder) {
            mThread = new DrawingThread();
            mThread.start();
        }
        public void surfaceDestroyed(SurfaceHolder holder) {
            mThread.waitForExit();
            mThread = null;
        }
        public void surfaceChanged(SurfaceHolder holder,
        int format, int w, int h) {
            mThread.onWindowResize(w, h);
        }
        class DrawingThread extends Thread {

            boolean stop;
            int w;
            int h;
            boolean changed = true;
            DrawingThread() {
                super();
                stop = false;
                w = 0;
                h = 0;
```

1 **Handle all creation and destruction**

2 **Do actual drawing**

3 **Register as callback**

4 **Create thread to do drawing**

```
            }
            @Override
            public void run() {
 EGL10 egl = (EGL10)EGLContext.getEGL();

                EGLDisplay dpy =
                egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);
                int[] version = new int[2];
                egl.eglInitialize(dpy, version);
                int[] configSpec = {

                   EGL10.EGL_RED_SIZE,      5,
                   EGL10.EGL_GREEN_SIZE,    6,
                   EGL10.EGL_BLUE_SIZE,     5,
                   EGL10.EGL_DEPTH_SIZE,   16,
                   EGL10.EGL_NONE
               };
               EGLConfig[] configs = new EGLConfig[1];
               int[] num_config = new int[1];
               egl.eglChooseConfig(dpy, configSpec, configs, 1,
 num_config);
              EGLConfig config = configs[0];
               EGLContext context = egl.eglCreateContext(dpy,
 config, EGL10.EGL_NO_CONTEXT, null);
              EGLSurface surface = null;
              GL10 gl = null;
              while( ! stop ) {

                  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);
```

5 Get EGL Instance

6 Specify configuration to use

7 Obtain reference to OpenGL ES context

8 Do actual drawing

```
               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() {
                        this.stop = true;
                        try {
                            join();
                        } catch (InterruptedException ex) {
                        }
                    }
                    private void drawFrame(GL10 gl) {
                        // do whatever drawing here.
                        }
        }
    }
}
```

Listing 9.15 will generate an empty black screen. Everything in this listing is code you need to draw and manage any OpenGL ES visualization. First, we import all our needed classes. Then we implement an inner class, which will handle everything about managing a surface: creating it, changing it, or deleting it. We extend the class `SurfaceView` and implement the `SurfaceHolder` interface, which allows us to get information back from Android when the surface changes, such as when someone resizes it ❶. With Android, all this has to be done asynchronously; you can't manage surfaces directly.

Next, we create a thread to do the drawing ❷ and create an init method that uses the SurfaceView class's getHolder method to get access to the SurfaceView and add a callback to it via the addCallBack method ❸. Now we can implement surfaceCreated, surfaceChanged, and surfaceDestroyed, which are all methods of the Callback class and are fired on the appropriate condition of change in the Surface's state.

Now that all the Callback methods are implemented, we'll create a thread to do all our drawing ❹. Before we can draw anything though, we need to create an OpenGL ES Context ❺ and create a handler to the surface ❻ so that we can use the OpenGL Context's method to act on the surface via the handle ❼. Now we can finally draw something, although in the drawFrame method ❽ we aren't doing anything.

If you were to run the code right now, all you'd get would be an empty window, but what we've generated so far will appear in some form or another in any OpenGL ES application you make on Android. Typically, you would break up your code so that an Activity class starts your code and another class implements your custom View. Yet another class might implement your SurfaceHolder and Callback and provide all the methods for detecting changes to the surface, as well as the actual drawing of your graphics in a thread. Finally, you might have another class for whatever code represents your graphics.

In the next section, we'll look at how to draw a square on the surface and how to create an animated cube.

9.3.2 Drawing a rectangle with OpenGL ES

In our next example, you'll use OpenGL ES to create a simple drawing, a rectangle, using OpenGL primitives, which are pixels, polygons, and triangles. When you draw the square, you'll use a primitive called the GL_Triangle_Strip, which takes three vertices (the x, y, and z points in an array of vertices) and draws a triangle. The last two vertices become the first two vertices for the next triangle, with the next vertex in the array being the final point. This process repeats for as many vertices as there are in the array, and it generates something like figure 9.5, where two triangles are drawn.

OpenGL supports a small set of primitives, shown in table 9.1, that allow you to build anything using simple geometric shapes, from a rectangle to 3D models of animated characters.

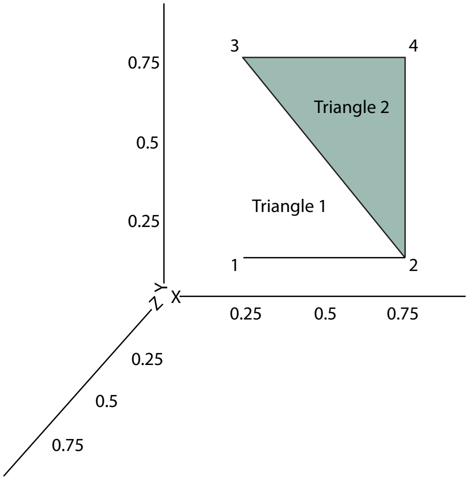


Figure 9.5 How two triangles are drawn from an array of vertices

Table 9.1 OpenGL primitives and their descriptions

Primitive flag	Description
GL_LINE_LOOP	Draws a continuous set of lines. After the first vertex, it draws a line between every successive vertex and the vertex before it. Then it connects the start and end vertices.
GL_LINE_STRIP	Draws a continuous set of lines. After the first vertex, it draws a line between every successive vertex and the vertex before it.
GL_LINES	Draws a line for every pair of vertices given.
GL_POINTS	Places a point at each vertex.
GL_TRIANGLE_FAN	After the first two vertices, every successive vertex uses the previous vertex and the first vertex to draw a triangle. This flag is used to draw cone-like shapes.
GL_TRIANGLE_STRIP	After the first two vertices, every successive vertex uses the previous two vertices to draw a triangle.
GL_TRIANGLES	For every triplet of vertices, it draws a triangle with corners specified by the coordinates of the vertices.

In the next listing, we use an array of vertices to define a square to paint on our surface. To use the code, insert it directly into the code for listing 9.15, immediately below the commented line `// do whatever drawing here`.

Listing 9.16 OpenGLSquare.java

```
gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
    GL10.GL_DEPTH_BUFFER_BIT);
float[] square = new float[] {
    0.25f, 0.25f, 0.0f,
    0.75f, 0.25f, 0.0f,
    0.25f, 0.75f, 0.0f,
    0.75f, 0.75f, 0.0f };
FloatBuffer squareBuff;
ByteBuffer bb =
ByteBuffer.allocateDirect(square.length*4);
    bb.order(ByteOrder.nativeOrder());
    squareBuff = bb.asFloatBuffer();
    squareBuff.put(square);
    squareBuff.position(0);
    gl.glMatrixMode(GL10.GL_PROJECTION);
    gl.glLoadIdentity();
    GLU.gluOrtho2D(gl, 0.0f,1.2f,0.0f,1.0f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, squareBuff);
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
    gl.glColor4f(0,1,1,1);
    gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);
```

1 **Create float buffer to hold square**

2 **Set up 2D orthographic viewing region**

3 **Set current vertices for drawing**

This code is dense with OpenGL commands. The first thing we do is clear the screen using glClear, which you want to do before every drawing. Then we build the array that'll represent the set of vertices that make up our square. As we explained before, we'll be using the OpenGL primitive GL_TRANGLE_STRIP to create the rectangle shown in figure 9.5, where the first set of three vertices (points 1, 2, and 3) represent the first triangle. The last vertex represents the third vertex (point 4) in the second triangle, which reuses vertices 2 and 3 from the first triangle as its first two to make the triangle described by points 2, 3, and 4. To put it more succinctly, Open GL takes one triangle and flips it over on its third side (in this case, the hypotenuse). We then create a buffer to hold that same square data ❶. We also tell the system that we'll be using a GL_PROJECTION for our matrix mode, which is a type of matrix transformation that's applied to every point in the matrix stack.

The next things we do are more related to setup. We load the identity matrix and then use the gluOrtho2D(GL10 gl, float left, float right, float bottom, float top) command to set the clipping planes that are mapped to the lower-left and upper-right corners of the window ❷.

Now we're ready to start drawing our image. First, we use the glVertexPointer(int size, int type, int stride, pointer to array) method, which indicates the location of vertices for our triangle strip. The method has four attributes: size, type, stride, and pointer. The size attribute specifies the number of coordinates per vertex (for example, a 2D shape might ignore the z axis and use only two coordinates per vertex), type defines the data type to be used (GL_BYTE, GL_SHORT, GL_FLOAT, and so on) ❸, stride specifies the offset between consecutive vertices (how many unused values exist between the end of the current vertex and the beginning of the next), and pointer is a reference to the array. Though most drawing in OpenGL ES is performed by using various forms of arrays such as the vertex array, they're all disabled by default to save system resources. To enable them, we use the OpenGL command glEnableClientState(array type), which accepts an array type; in our case the type is GL_VERTEX_ARRAY.

Finally, we use the glDrawArrays function to render our arrays into the OpenGL primitives and create our simple drawing. The glDrawArrays(mode, first, count) function has three attributes: mode indicates which primitive to render, such as GL_TRIANGLE_STRIP; first is the starting index of the array, which we set to 0 because we want it to render all the vertices in the array; and count specifies the number of indices to be rendered, which for us is 4.

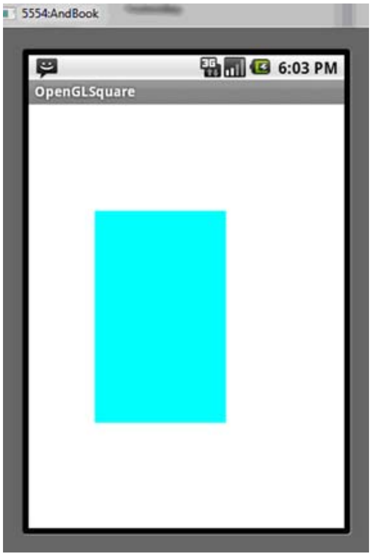


Figure 9.6 A simple rectangle drawn on our surface using OpenGL ES

Now if you run the code, you should see a simple blue rectangle on a white surface, like the one in figure 9.6. It isn't particularly exciting, but you would need most of the code you used for this example for any OpenGL project.

There you have it—your first graphic in OpenGL ES. Now we're going to do something way more interesting. In our next example, you're going to create a 3D cube with different colors on each side, then rotate it in space.

9.3.3 Three-dimensional shapes and surfaces with OpenGL ES

In this section, we're going to use much of the code from the previous example, but we're going to extend it to create a 3D cube that rotates. We'll examine how to introduce perspective to your graphics to give the illusion of depth.

Depth works in OpenGL by using a *depth buffer*, which contains a depth value for every pixel, in the range 0 to 1. The value represents the perceived distance between objects and your viewpoint; when two objects' depth values are compared, the value closer to 0 will appear in front on the screen. To make use of depth in our program, we need to first enable the depth buffer by passing `GL_DEPTH_TEST` to the `glEnable` method. Next, we need to use `glDepthFunc` to define how values are compared. For our example, we're going to use `GL_LEQUAL`, defined in table 9.2, which tells the system to show objects with a lower depth value in front of other objects.

Table 9.2 Flags for determining how values in the depth buffer are compared

Flag	Description
`GL_ALWAYS`	Always passes
`GL_EQUAL`	Passes if the incoming depth value is equal to the stored value
`GL_GEQUAL`	Passes if the incoming depth value is greater than or equal to the stored value
`GL_GREATER`	Passes if the incoming depth value is greater than the stored value
`GL_LEQUAL`	Passes if the incoming depth value is less than or equal to the stored value
`GL_LESS`	Passes if the incoming depth value is less than the stored value
`GL_NEVER`	Never passes
`GL_NOTEQUAL`	Passes if the incoming depth value is not equal to the stored value

When you draw a primitive, the depth test occurs. If the value passes the test, the incoming color value replaces the current one.

The default value is `GL_LESS`. You want the value to pass the test if the values are equal as well. Objects with the same z value will display, depending on the order in which they were drawn. We pass `GL_LEQUAL` to the function.

One important part of maintaining the illusion of depth is providing perspective. In OpenGL, a typical perspective is represented by a viewpoint with near and far clipping planes and top, bottom, left, and right planes, where objects that are closer to the far plane appear smaller, as in figure 9.7.

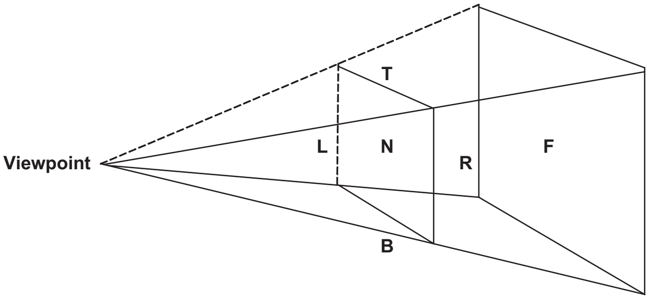


Figure 9.7 In OpenGL, a perspective is made up of a viewpoint and near (N), far (F), left (L), right (R), top (T), and bottom (B) clipping planes.

OpenGL ES provides a function called `gluPerspective(GL10 gl, float fovy, float aspect, float zNear, float zFar)` with five parameters (see table 9.3) that lets you easily create perspective.

Table 9.3 Parameters for the `gluPerspective` function

Parameter	Description
`aspect`	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).
`fovy`	Field of view angle in the y direction, in degrees,
`gl`	GL10 interface.
`zFar`	The distance from the viewer to the far clipping plane. This value is always positive.
`zNear`	The distance from the viewer to the near clipping plane. This value is always positive.

To demonstrate depth and perspective, you're going to create a project called OpenGLCube and copy and paste the code from listing 9.15 into the `OpenGLCubeActivity`.

Now add two new variables to your code, shown in the following listing, right at the beginning of the `DrawSurfaceView` inner class.

Listing 9.17 OpenGLCubeActivity.java

```
class DrawingSurfaceView extends SurfaceView implements
SurfaceHolder.Callback  {
    public SurfaceHolder mHolder;
    float xrot = 0.0f;
    float yrot = 0.0f;
```

We're going to use `xrot` and `yrot` variables later in our code to govern the rotation of our cube.

Next, right before the method, add a new method called `makeFloatBuffer`, as in the following listing.

Listing 9.18 OpenGLCubeActivity.java

```
protected FloatBuffer makeFloatBuffer(float[] arr) {
            ByteBuffer bb = ByteBuffer.allocateDirect(arr.length*4);
            bb.order(ByteOrder.nativeOrder());
            FloatBuffer fb = bb.asFloatBuffer();
            fb.put(arr);
            fb.position(0);
            return fb;
}
```

This float buffer is the same as the one in listing 9.16, but we've 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 the following listing into the `drawFrame` method. Note you'll also need to update your `drawFrame` call in the following way:

```
drawFrame(gl, w, h);
```

Listing 9.19 OpenGLCubeActivity.java

```
        private void drawFrame(GL10 gl, int w1, int h1) {
            float mycube[] = {
                // FRONT
                -0.5f, -0.5f,  0.5f,
                0.5f, -0.5f,  0.5f,
                -0.5f,  0.5f,  0.5f,
                0.5f,  0.5f,  0.5f,
                // BACK
                -0.5f, -0.5f, -0.5f,
                -0.5f,  0.5f, -0.5f,
                0.5f, -0.5f, -0.5f,
                 0.5f,  0.5f, -0.5f,
                // LEFT
                -0.5f, -0.5f,  0.5f,
                -0.5f,  0.5f,  0.5f,
                -0.5f, -0.5f, -0.5f,
                -0.5f,  0.5f, -0.5f,
                // RIGHT
                0.5f, -0.5f, -0.5f,
                0.5f,  0.5f, -0.5f,
                0.5f, -0.5f,  0.5f,
                0.5f,  0.5f,  0.5f,
                // TOP
                -0.5f,  0.5f,  0.5f,
                0.5f,  0.5f,  0.5f,
                -0.5f,  0.5f, -0.5f,
                0.5f,  0.5f, -0.5f,
                // BOTTOM
                -0.5f, -0.5f,  0.5f,
                -0.5f, -0.5f, -0.5f,
                0.5f, -0.5f,  0.5f,
                0.5f, -0.5f, -0.5f,
            };
            FloatBuffer cubeBuff;

            cubeBuff = makeFloatBuffer(mycube);

            gl.glEnable(GL10.GL_DEPTH_TEST);
            gl.glEnable(GL10.GL_CULL_FACE);
            gl.glDepthFunc(GL10.GL_LEQUAL);
            gl.glClearDepthf(1.0f);
            gl.glClear(GL10.GL_COLOR_BUFFER_BIT |
GL10.GL_DEPTH_BUFFER_BIT);
            gl.glMatrixMode(GL10.GL_PROJECTION);
            gl.glLoadIdentity();
            gl.glViewport(0,0,w,h);
            GLU.gluPerspective(gl, 45.0f,
((float)w)/h, 1f, 100f);
            gl.glMatrixMode(GL10.GL_MODELVIEW);
            gl.glLoadIdentity();
            GLU.gluLookAt(gl, 0, 0, 3, 0, 0, 0, 0, 1, 0);
            gl.glShadeModel(GL10.GL_SMOOTH);
            gl.glVertexPointer(3, GL10.GL_FLOAT, 0, cubeBuff);
            gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
            gl.glRotatef(xrot, 1, 0, 0);
            gl.glRotatef(yrot, 0, 1, 0);
            gl.glColor4f(1.0f, 0, 0, 1.0f);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 4);

            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 4, 4);
            gl.glColor4f(0, 1.0f, 0, 1.0f);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 8, 4);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 12, 4);
            gl.glColor4f(0, 0, 1.0f, 1.0f);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 16, 4);
            gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 20, 4);
            xrot += 1.0f;
            yrot += 0.5f;
```

❶ Create float buffer for vertices

❷ Enable depth test

❸ Define perspective

❹ Draw six sides in three colors

❺ Increment x and y rotations

This listing doesn't contain much new code. First, we describe the vertices for a cube, which is built in the same way as our simple rectangle in listing 9.16 (using triangles). Next, we set up the float buffer for our vertices ❶ and enable the depth function ❷ and perspective function ❸ to provide a sense of depth. Note that with our `gluPerspective` we passed `45.0f` (45 degrees) to give a more natural viewpoint.

Next, we use the `GLU.gluLookAt(GL10 gl, float eyeX, float eyeY, float eyeZ, float centerX, float centerY, float centerZ, float upX, float upY, float upZ)` function to move the position of our view without having to modify the projection matrix directly. When we've established our view position, we turn on smooth shading for the model and rotate the cube around the x and y axes ❺. Then we draw the cube sides and increment the rotation so that on the next iteration of draw, the cube is drawn at a slightly different angle ❹. If you run the code, you should now see a rotating 3D cube like the one shown in figure 9.8.

NOTE You can try experimenting with the `fovy` value to see how changing the angle affects the display of the cube.

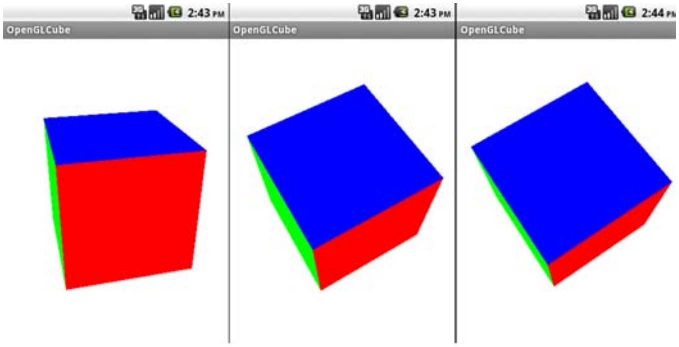


Figure 9.8 **A 3D cube rotating in space**

You've done a lot in this section, starting with creating a simple OpenGL ES context in which you can develop your OpenGL ES applications. Next, you learned how to build shapes using OpenGL ES by creating multiple triangles. Then, you learned how to realize this in three dimensions while incorporating it into a simple example. You accomplished much of this without diving deep into OpenGL ES, which is definitely nontrivial, but the good news is if you're serious about doing 3D graphics on Android, it's definitely possible. Because Android provides excellent support for OpenGL ES, you can find plenty of tutorials and references on the internet or at your local bookstore.

9.4 Summary

In this chapter, we've lightly touched on a number of topics related to Android's powerful graphics features. First, we looked at how you can use both Java and XML with the Android Graphics API to describe simple shapes. Next, we looked at how you can use Android's frame-by-frame XML to create an animation. You also learned how to use more standard pixel manipulation to provide the illusion of movement through Java and the Graphics API. Finally, we delved lightly into Android's support of OpenGL ES. We looked at how to create an OpenGL context, and then built a shape in that context. Finally, you built a 3D animated cube.

Graphics and visualizations are large and complex topics, easily filling a whole book. Yet, because Android uses open and well-defined standards and supports an excellent API for graphics, it should be easy for you to use Android's documentation, API, and other resources, such as Manning's *Java 3D Programming* by Daniel Selman, to develop anything from a new drawing program to complex games.

In the next chapter, we'll move from graphics to working with multiple media. We'll explore working with audio and video to lay the groundwork for making rich multimedia applications.

10 Multimedia

This chapter covers

- Playing audio and video
- Controlling the camera
- Recording audio
- Recording video

Today, people use cell phones for almost everything but phone calls, from chatting to surfing the web to listening to music and even to watching live streaming TV. Nowadays, a cell phone needs to support multimedia to be considered a usable device. In this chapter, we're going to look at how you can use Android to play audio files, watch video, take pictures, and even record sound.

Android supports multimedia by using the open source multimedia system called *OpenCORE* from PacketVideo Corporation. OpenCORE provides the foundation for Android's media services, which Android wraps in an easy-to-use API. In addition to the OpenCORE framework, the Android platform is migrating to a Google-written multimedia framework named Stagefright. Both frameworks are provided in version 2.2 (Froyo) and in subsequent versions of the SDK. It's anticipated that most, if not all, of the multimedia functionality will be handled by the Stagefright code base.

In this chapter, we'll look at OpenCORE's multimedia 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 begin, let's look at OpenCORE's multimedia architecture.

10.1 Introduction to multimedia and OpenCORE

Because the foundation of Android's multimedia platform is PacketVideo's OpenCORE, we're going to review OpenCORE's architecture and services. OpenCORE is a Java open source, multimedia platform that supports:

- Interfaces for third-party and hardware media codecs, input and output devices, and content policies
- Media playback, streaming, downloading, and progressive playback, including 3rd Generation Partnership Program (3GPP), Moving Picture Experts Group 4 (MPEG-4), Advanced Audio Coding (AAC), and Moving Picture Experts Group Audio Layer 3(MP3) containers
- Video and image encoders and decoders, including MPEG-4, International Telecommunication Union H.263 video standard (H.263), Advanced Video Coding (AVC H.264), and the Joint Photographic Experts Group (JPEG)
- Speech codecs, including Adaptive Multi-Rate audio codecs AMR-NB and AMR-WB
- Audio codecs, including MP3, AAC, and AAC+
- Media recording, including 3GPP, MPEG-4, and JPEG
- Video telephony based on the 3GPP video conferencing standard 324-M
- 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, shown in figure 10.1.

> **NOTE** As of Android 2.2, Google added a new multimedia platform called Stagefright with the intention of replacing OpenCORE. As for Android 2.3, Stagefright fully replaces OpenCORE with additional features, such as support for multiple cameras, progressive streaming over HTTP, and the like. For the most part, Android 2.3 and Stagefright focus on increasing performance and making the underlying multimedia system easier to use and develop for. If you plan to target Android 2.3 for your multimedia application development, be sure to check against the Google list of support media file types at http://developer.android.com/guide/appendix/media-formats.html.

As you can see from figure 10.1, OpenCORE's architecture has excellent support for multimedia and numerous codecs. In the next section, we're going to dive right in and use the Android API to play audio files.

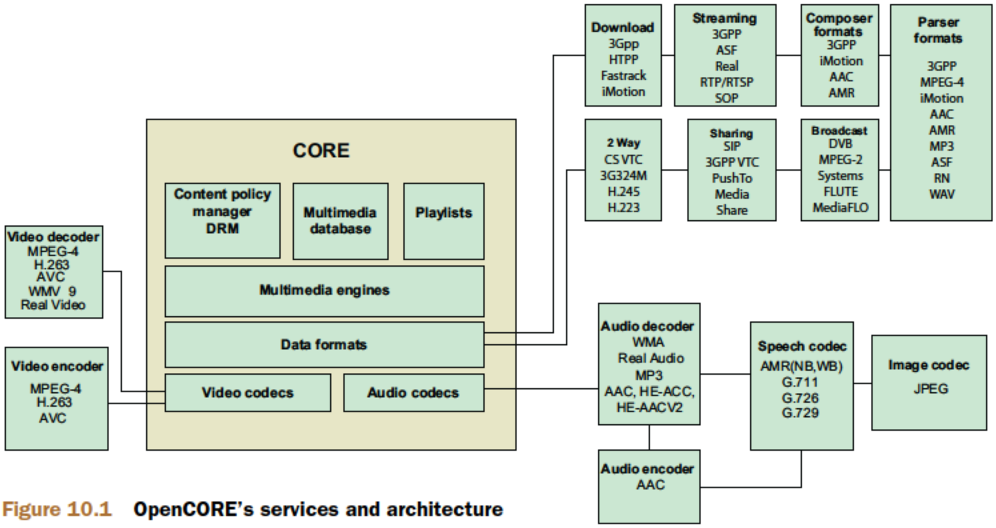


Figure 10.1 OpenCORE's services and architecture

NOTE The current Android SDK doesn't support video recording via the API. Video recording is still possible, but it's specific to the phone vendor.

10.2 Playing audio

Probably the most basic need for multimedia on a cell phone is the ability to play audio files, whether new ringtones, MP3s, or quick audio notes. Android's Media Player is easy to use. At a high level, all you need to do to play back an MP3 file is follow these steps:

1. Put the MP3 in the res/raw directory in a project (note that you can also use a URI to access files on the network or via the internet).
2. Create a new instance of the `MediaPlayer` and reference your MP3 by calling `MediaPlayer.create()`.
3. Call the `MediaPlayer` methods `prepare()` and `start()`.

Let's work through an example to demonstrate how simple this task is. First, create a new project called MediaPlayer Example, with an `Activity` called `MediaPlayerActivity`. Now, create a new folder under res/ called raw. We'll store our MP3s in this folder. For this example, we'll use a ringtone for the game Halo 3, which you can retrieve from `MediaPlayer.create`. Download the Halo 3 theme song and any other MP3s you fancy, and put them in the raw directory. Next, create a simple `Button` for the music player, as shown in the following listing.

Listing 10.1 main.xml for MediaPlayer Example

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Simple Media Player"
    />
<Button android:id="@+id/playsong"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Halo 3 Theme Song"
    />
</LinearLayout>
```

We need to fill out our `MediaPlayerActivity` class, as shown in the following listing.

Listing 10.2 MediaPlayerActivity.java

```
public class MediaPlayerActivity extends Activity {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.main);
        Button mybutton = (Button) findViewById(R.id.playsong);
        mybutton.setOnClickListener(new Button.OnClickListener() {
               public void onClick(View v) {
                   MediaPlayer mp =
 MediaPlayer.create(MediaPlayerActivity.this,
 R.raw.halotheme);
                  mp.start();
                  mp.setOnCompletionListener(new OnCompletionListener(){
                         public void onCompletion(MediaPlayer arg0) {

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

❶ Set view and button to play MP3

❷ Get context and play MP3

As you can see, playing back an MP3 is easy. In listing 10.2, all we did was use the view that we created in listing 10.1 and map the button, `playsong`, to `mybutton`, which we then bound to the `setOnClickListener()` ❶. Inside the listener, we created the `MediaPlayer` instance ❷ using the `create(Context context, int resourceid)` method, which takes our context and a resource ID for our MP3. Finally, we set the `setOnCompletionListener`, which will perform some task on completion. For the moment, we do nothing, but you might want to change a button's state or provide a notification to a user that the song is over, or ask if the user would like to play another song. If you want to do any of these things, you'd use this method.

Now if you compile the application and run it, you should see something like figure 10.2. Click the button, and you should hear the Halo 3 song played back 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.

Now that we've looked at how to play an audio file, let's see how you can play a video file.

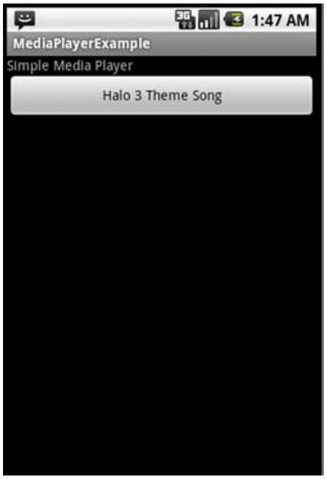


Figure 10.2 Simple media player example

10.3 Playing video

Playing a video is slightly more complicated than playing audio with the MediaPlayer API, in part because you have to provide a view surface for your video to play on. Android has a `VideoView` widget that handles that task for you; you can use it in any layout manager. Android also provides a number of display options, including scaling and tinting. So let's get started with playing video by creating a new project called Simple Video Player. Next, create a layout, as shown in the following listing.

> **NOTE** Currently the emulator has some issues playing video content on certain computers and operating systems. Don't be surprised if your audio or video playback is choppy.

Listing 10.3 main.xml—UI for Simple Video Player

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent"
>
    <VideoView android:id="@+id/video"
    android:layout_width="320px"
    android:layout_height="240px"
    />/
</LinearLayout>
```

❶ Add VideoView widget

All we've done in this listing is add the `VideoView` widget ❶. The `VideoView` provides a UI widget with stop, play, advance, rewind, and other buttons, making it unnecessary to add your own. Next, we need to write a class to play the video, which is shown in the following listing.

Listing 10.4 SimpleVideo.java

```
    public class SimpleVideo extends Activity {

    private VideoView myVideo;
    private MediaController mc;
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        getWindow().setFormat(PixelFormat.TRANSLUCENT);
        setContentView(R.layout.main);
        this.myVideo = (VideoView) findViewById(R.id.video);
        this.myVideo.setVideoPath("sdcard/test.mp4");
        this.mc = new MediaController(this);
        this.mc.setMediaPlayer(this.myVideo);
        this.myVideo.setMediaController(this.mc);
        this.myVideo.requestFocus();
    }
}
```

❶ **Create translucent window**

In this listing, we first created a translucent window, which is necessary for our `SurfaceView` ❶. Next, we reference the `VideoView` as a container for playing the video, and use its `setVideoPath()` to have it look at an SD card for our test MP4. Finally, we set up the `MediaController` and use the `setMediaController()` to perform a callback to the `VideoView` to notify it when our video is finished playing.

Before we can run this application, we need to set up an `sdcard` in the emulator (see chapter 5 for details on the SD card). First, create a new SD card image:

```
mksdcard 512M mysdcard
```

Hit Return. A 512 MB FAT32 image named mysdcard has now been created for you to load into the emulator. Load the SD card into the emulator like this:

```
emulator -sdcard mysdcard
```

Now push the file test.mp4 to the disk image. After you've pushed the file to the image, you can launch the SimpleVideo application by going to your IDE and running the project while the emulator is already running. You should now see something like figure 10.3.

As you can see, the `VideoView` and `MediaPlayer` classes simplify working with video files. Something you'll need to pay attention to when working with video files is that the emulator and physical devices will react differently with very large media files.

Now that you've seen how simple it is to play media using Android's MediaPlayer API, let's look at how you can use a phone's built-in camera or microphone to capture images or audio.

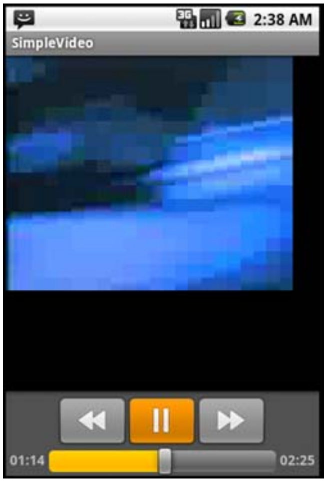


Figure 10.3 Playing an MP4 video in the Android emulator

10.4 Capturing media

Using your cell phone to take pictures, record memos, film short videos, and so on, are features that are expected of any such device. In this section, we're going to not only look at how to capture media from the microphone and camera, but also how to write these files to the simulated SD card image you created in the previous section.

To get started, let's examine how to use the Android Camera class to capture images and save them to a file.

10.4.1 Understanding the camera

An important feature of modern cell phones is their ability to take pictures or video using a built-in camera. Some phones even support using the camera's microphone to capture audio. Android, of course, supports all three features and provides a variety of ways to interact with the camera. In this section, we're going to look at how to interact with the camera and take photographs. In the next section, you'll use the camera to take video and save it to an SD card.

You'll be creating a new project called SimpleCamera to demonstrate how to connect to a phone's camera to capture images. For this project, we'll be using the Camera class (http://code.google.com/android/reference/android/hardware/Camera.html) to tie the emulator's (or phone's) camera to a View. Most of the code that we've created for this project deals with showing the input from the camera, but the main work for taking a picture is done by a single method called take-Picture(Camera.ShutterCallback shutter, Camera.PictureCallback raw, Camera.PictureCallback jpeg), which has three callbacks that allow you to control how a picture is taken.

Before we get any further into the Camera class and how to use the camera, let's create the project. We'll be creating two classes; because the main class is long, we'll break it into two sections. For the first section, look at the following listing, CameraExample.java.

> **NOTE** The Android emulator doesn't allow you to connect to camera devices, such as a webcam, on your computer; all your pictures will display a chessboard like the one shown in figure 10.4. You can 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 website, where he has an open source project for obtaining live images from a webcam: http://www.tomgibara.com/android/camera-source. It's possible that in later versions of the SDK, the emulator will support connections to cameras on the hardware the emulator is running on.

Listing 10.5 CameraExample.java

```
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 icicle)
    {
        super.onCreate(icicle);
        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);
    }
    @Override
    public boolean onCreateOptionsMenu(android.view.Menu menu) {
        MenuItem item =
menu.add(0, 0, 0, "View Photos?");
        item.setOnMenuItemClickListener(new
            MenuItem.OnMenuItemClickListener() {
            public boolean onMenuItemClick(MenuItem item) {
                Intent intent = new Intent(Intent.ACTION_VIEW,
                    SimpleCamera.this.targetResource);
                startActivity(intent);
                return true;
            }
        });
        return true;
    }
    @Override
    protected void onRestoreInstanceState(Bundle savedInstanceState)
 {
        super.onRestoreInstanceState(savedInstanceState);
    }
    Camera.PictureCallback mPictureCallbackRaw = new
        Camera.PictureCallback() {
        public void onPictureTaken(byte[] data, Camera c) {
            SimpleCamera.this.camera.startPreview();
        }
    };
    Camera.ShutterCallback mShutterCallback = new Camera.ShutterCallback()
{
        Public void onShutter() {}
        }
    };
```

❶ Create menu to Android's Photo Gallery

❷ Create PictureCallback

❸ Create ShutterCallback

This listing is straightforward. First, we set variables for managing a `surfaceView` and then set up the `View`. Next, we create a simple menu and menu option that will float over our surface when the user clicks the Menu button on the phone while the application is running ❶. Doing so will open Android's picture browser and let the user view the photos on the camera. Next, we create the first `PictureCallback`, which is called when a picture is first taken ❷. This first callback captures the `PictureCallback`'s only method, `onPictureTaken(byte[] data, Camera camera)`, to grab the raw image data directly from the camera. Next, we create a `ShutterCallback`, which can be used with its method, `onShutter()`, to play a sound; here we don't call the method ❸. We'll continue with the CameraExample.java in the next listing.

Listing 10.6 CameraExample.java continued

```
@Override
    public boolean onKeyDown(int keyCode, KeyEvent event) {

        ImageCaptureCallback camDemo = null;
        if(keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
            try {
                String filename = this.timeStampFormat.format(new Date());
                ContentValues values = new ContentValues();
                values.put(MediaColumns.TITLE, filename);
                values.put(ImageColumns.DESCRIPTION,
                    "Image from Android Emulator");
                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) {
            this.camera.takePicture(this.mShutterCallback,
                this.mPictureCallbackRaw, this.camDemo);
            return true;
        }
        return false;
    }
    @Override
    protected void onResume()
 {
        Log.e(getClass().getSimpleName(), "onResume");
        super.onResume();
    }
    @Override
    protected void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
    }
    @Override
    protected void onStop()
    {
       super.onStop();
    }
    public void surfaceChanged(SurfaceHolder holder,
int format, int w, int h)
 {
        if (this.isPreviewRunning) {
            this.camera.stopPreview();
        }
        Camera.Parameters p = this.camera.getParameters();
        p.setPreviewSize(w, h);
        this.camera.setParameters(p);
```

1 Create method to detect key events

2 If center key pressed, write file to sdcard

3 If center key depressed, take picture

```
        try{
            this.camera.setPreviewDisplay(holder);
        }catch(IOexcetion e{
            e.printStackTrace();
        }
        this.camera.startPreview();
        this.isPreviewRunning = true;
    }
    public void surfaceCreated(SurfaceHolder holder)
 {
       this.camera = Camera.open();
    }
    public void surfaceDestroyed(SurfaceHolder holder) {
        this.camera.stopPreview();
        this.isPreviewRunning = false;
        this.camera.release();
    }
}
```

This listing is more complicated than listing 10.5, although a large amount of the code is about managing the surface for the camera preview. The first line is the start of an implementation of the method onKeyDown ❶, which checks to see whether the center key on the dpad was pressed. If it was, we set up the creation of a file, and by using the ImageCaptureCallback, which we'll define in listing 10.7, we create an Outputstream to write our image data to ❷, including not only the image but the filename and other metadata. Next, we call the method takePicture() and pass to it the three callbacks mShutterCallback, mPictureCallbackRaw, and camDemo. mPictureCallbackRaw is our raw image and camDemo writes the image to a file on the SD card ❸, as you can see in the following listing.

Listing 10.7 ImageCaptureCallback.java

```
public class ImageCaptureCallback implements PictureCallback  {
    private OutputStream filoutputStream;
    public ImageCaptureCallback(OutputStream filoutputStream) {
        this.filoutputStream = filoutputStream;
    }
    public void onPictureTaken(byte[] data, Camera camera) {
        try {
            this.filoutputStream.write(data);
            this.filoutputStream.flush();
            this.filoutputStream.close();
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Here, the class implements the PictureCallback interface and provides two methods. The constructor creates a stream to write data to, and the second method, onPictureTaken, takes binary data and writes to the SD card as a JPEG.

If you build this project and start the emulator running using the SD card image you created earlier in this chapter, you should see something like figure 10.4 when you start the SimpleCamera application from the Android menu. If you look at figure 10.4, you'll notice an odd black-and-white checked background with a bouncing gray box. The Android emulator generates this test pattern to simulate an image feed because the emulator isn't pulling a live feed from the camera.

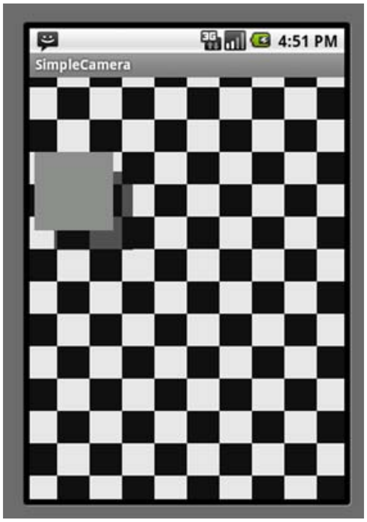
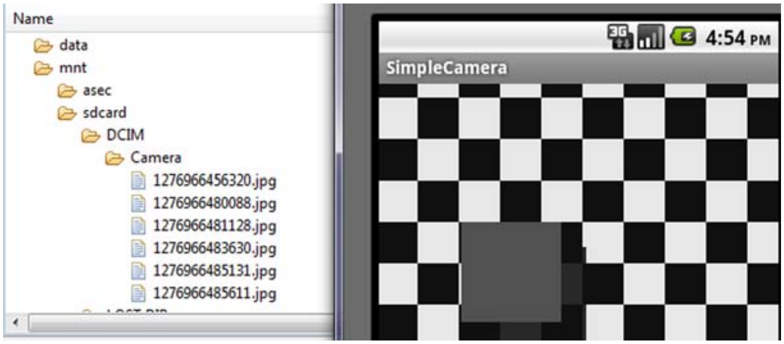


Figure 10.4 Test pattern coming from the emulator camera and displayed in the SimpleCamera application

Now if you click the center button on the dpad in the emulator, the application will take a picture. To see the picture, click the Menu button; a menu appears on the camera view window with a single option, View Pictures. If you select View Pictures, you're taken to the Android picture explorer, and you should see Android's image placeholders representing the number of camera captures. You can also see the JPEG files that were written to the SD card by opening the DDMS in Eclipse and navigating to mnt > sdcard > DCIM > Camera. You can see an example in figure 10.5.

As you can see, working with the camera in Android isn't particularly complicated. To see how a real camera will behave, you'll have to test on a real handset until the emulator provides a simple way to connect to a camera on your computer. This workaround shouldn't stop you from developing your camera applications. A wealth of Android applications already makes sophisticated use of the camera, ranging from games to an application that uses a picture of your face to unlock your phone.

Now that you've seen how the `Camera` class works in Android, let's look at how to capture or record audio from a camera's microphone. In the next section, we'll explore the `MediaRecorder` class and you'll write recordings to an SD card.



```
Name
  data
  mnt
    asec
    sdcard
      DCIM
        Camera
          1276966456320.jpg
          1276966480088.jpg
          1276966481128.jpg
          1276966483630.jpg
          1276966485131.jpg
          1276966485611.jpg
```

Figure 10.5 The Android emulator shows placeholder images for each photo taken.

10.4.2 Capturing audio

Now we'll look at using the onboard microphone to record audio. In this section, we're going to use the Android MediaRecorder example from Google Android Developers list, which you can find at http://code.google.com/p/unlocking-android/. The code shown in this section has been updated slightly.

> **NOTE** Audio capture requires a physical device running Android because it's not currently supported in the Android emulator.

In general, recording audio or video follows the same process in Android:

1. Create an instance of `android.media.MediaRecorder`
2. Create an instance of `andriod.content.ContentValues`, and add properties such as `TITLE`, `TIMESTAMP`, and the all-important `MIME_TYPE`
3. Create a file path for the data to go to using `android.content.ContentResolver`
4. To set a preview display on a view surface, use `MediaRecorder.setPreviewDisplay()`
5. Set the source for audio, using `MediaRecorder.setAudioSource()`
6. Set output file format, using `MediaRecorder.setOutputFormat()`
7. Set your encoding for audio, using `MediaRecorder.setAudioEncoder()`
8. Use `prepare()` and `start()` to prepare and start your recordings
9. Use `stop()` and `release()` to gracefully stop and clean up your recording process

Though recording media isn't especially complex, you can see that it's more complex than playing it. To understand how to use the `MediaRecorder` class, we'll look at an application. Create a new application called SoundRecordingDemo. Next, you need to edit the AndroidManifest.xml file and add the following line:

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

This code will allow the application to record the audio files and play them. Next, create the class shown in the following listing.

Listing 10.8 SoundRecordingdemo.java

```
public class SoundRecordingDemo extends Activity  {
    MediaRecorder mRecorder;
    File mSampleFile = null;
    static final String SAMPLE_PREFIX = "recording";
    static final String SAMPLE_EXTENSION = ".mp3";
    private static final String TAG="SoundRecordingDemo";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        this.mRecorder = new MediaRecorder();
        Button startRecording = (Button)findViewById(R.id.startrecording);
        Button stopRecording = (Button)findViewById(R.id.stoprecording);
        startRecording.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v) {
                startRecording();
            }
        });
        stopRecording.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v) {
                stopRecording();
                addToDB();
            }
        });
    }
    protected void addToDB() {
        ContentValues values = new ContentValues(3);
        long current = System.currentTimeMillis();
        values.put(MediaColumns.TITLE, "test_audio");
        values.put(MediaColumns.DATE_ADDED, (int) (current / 1000));
        values.put(MediaColumns.MIME_TYPE, "audio/mp3");
        values.put(MediaColumns.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));
    }
    protected void startRecording() {
        this.mRecorder = new MediaRecorder();
        this.mRecorder.setAudioSource(MediaRecorder.AudioSource.MIC);
        this.mRecorder.setOutputFormat
(MediaRecorder.OutputFormat.THREE_GPP);
        this.mRecorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
        this.mRecorder.setOutputFile(this.mSampleFile.getAbsolutePath());
        try{this.mRecorder.prepare();
        } catch (IllegalStateException e1) {
                    e1.printStackTrace();
           } catch (IOException e1 {
                    e1.printStackTrace();
           }
        this.mRecorder.start();
        if (this.mSampleFile == null) {
            File sampleDir = Environment.getExternalStorageDirectory();
            try {
                 this.mSampleFile = File.createTempFile(
                     SoundRecordingDemo.SAMPLE_PREFIX,
                     SoundRecordingDemo.SAMPLE_EXTENSION, sampleDir);
            } catch (IOException e) {
                Log.e(TAG,"sdcard access error");
                return;
            }
        }
    }
    protected void stopRecording() {
        this.mRecorder.stop();
        this.mRecorder.release();
    }
}
```

1 Set metadata for audio

2 Notify music player new audio file created

3 Start recording file

4 Stop recording and release MediaRecorder

In this listing, the first part of the code is 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 ❶. Next, we call the Intent ACTION_MEDIA_SCANNER_SCAN_FILE to notify applications such as Android's Music Player that a new audio file has been created ❷. Calling this Intent allows us to use the Music Player to look for new files in a playlist and play the files.

Now that we've finished the addToDB method, we create the startRecording method, which creates a new MediaRecorder ❸. As in the steps in the beginning of this section, we set an audio source, which is the microphone, set an output format as THREE_GPP, set the audio encoder type to AMR_NB, and then set the output file path to write the file. Next, we use the methods prepare() and start() to enable audio recording.

Finally, we create the stopRecording() method to stop the MediaRecorder from saving audio ❹ by using the methods stop() and release(). If you build this application and run the emulator with the SD card image from the previous section, you should be able to launch the application from Eclipse and press the Start Recording button. After a few seconds, press the Stop Recording button and open the DDMS; you should be able to navigate to the sdcard folder and see your recordings, as shown in figure 10.6.

If music is playing on your computer's audio system, the Android emulator will pick it up and record it directly from the audio buffer (it's not recording from a microphone). You can then easily test whether it recorded sound by opening the Android Music Player and selecting Playlists > Recently Added. It should play your recorded file, and you should be able to hear anything that was playing on your computer at the time.

As of version 1.5, Android supported the recording of video, although many developers found it difficult and some vendors implemented their own customer solutions to support video recording. With the release of 2.0, 2.1, and 2.2, video has become far easier to work with, both for playing as well as recording. You'll see how much easier in the next section about using the MediaRecorder class to write a simple application for recording video.

10.5 Recording video

Video recording on Android is no more difficult than recording audio, with the exception that you have a few different

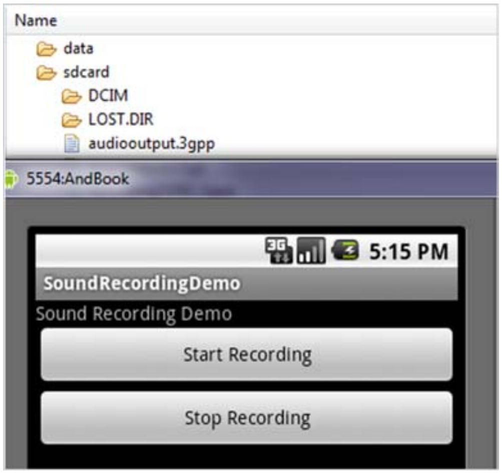


Figure 10.6 **An example of audio files being saved to the SD card image in the emulator**

fields. But there's one important difference—unlike with recording audio data, Android requires you to first preview a video feed before you can record it by passing it a surface object much like we did with the camera application earlier in this chapter. It's worth repeating this point because when Android started supporting video recording, many developers found themselves unable to record video: You must always provide a surface object. This might be awkward for some applications, but it's currently required in Android up to 2.2. Also, like recording audio, you have to provide several permissions to Android so you can record video. The new one is `RECORD_VIDEO`, which lets you use the camera to record video. The other permissions are `CAMERA`, `RECORD_AUDIO`, and `WRITE_EXTERNAL_ STORAGE`, as shown in the following listing. So go ahead and set up a new project called VideoCam and use the permissions in this AndroidManifest.xml.

Listing 10.9 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.msi.manning.chapter10.VideoCam"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
 android:label="@string/app_name">
        <activity android:name=".VideoCam"
 android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name=
"android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
     <uses-permission android:name="android.permission.CAMERA">
</uses-permission>
       <uses-permission android:name=
"android.permission.RECORD_AUDIO"></uses-permission>
       <uses-permission android:name=
"android.permission.RECORD_VIDEO"></uses-permission>
       <uses-permission android:name=
"android.permission.WRITE_EXTERNAL_STORAGE" />
       <uses-feature android:name="android.hardware.camera" />
</manifest>
```

Now that you've defined the manifest, you need to create a simple layout that has a preview area and some buttons to start, stop, pause, and play your video recording. The layout is shown in the following listing:

Listing 10.10 maim.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!-- This file is /res/layout/main.xml -->
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">

    <RelativeLayout android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:id="@+id/relativeVideoLayoutView"
            android:layout_centerInParent="true">

            <VideoView android:id="@+id/videoView"
     android:layout_width="176px"
                   android:layout_height="144px"
                   android:layout_centerInParent="true"/>

    </RelativeLayout>

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_centerHorizontal="true"
        android:layout_below="@+id/relativeVideoLayoutView">
        <ImageButton android:id="@+id/playRecordingBtn"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:background="@drawable/play"
            />

    <ImageButton android:id="@+id/bgnBtn"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/record"
        android:enabled="false"
        />
    </LinearLayout>
</RelativeLayout>
```

NOTE You'll need to download this code from http://code.google.com/p/unlocking-android/ to get the open source icons that we use for the buttons, or you can use your own.

Video recording follows a set of steps that are similar to those for audio recording:

1. Create an instance of `android.media.MediaRecorder`.
2. Set up a `VideoView`.
3. To set a preview display on a view surface, use `MediaRecorder.setPreviewDisplay()`.
4. Set the source for audio, using `MediaRecorder.setAudioSource()`.
5. Set the source for video, using `MediaRecorder.setVideoSource()`.
6. Set your encoding for audio, using `MediaRecorder.setAudioEncoder()`.
7. Set your encoding for video, using `MediaRecorder.setVideoEncoder()`.
8. Set output file format using `MediaRecorder.setOutputFormat()`.
9. Set the video size using `setVideoSize()`. (At the time this book was written, there was a bug in `setVideoSize` that limited it to 320 by 240.)

10 Set the video frame rate, using `setVideoFrameRate`.

11 Use `prepare()` and `start()` to prepare and start your recordings.

12 Use `stop()` and `release()` to gracefully stop and clean up your recording process.

As you can see, using video is very similar to using audio. So let's go ahead and finish our example by using the code in the following listing.

Listing 10.11 VideoCam.java

```
VideoCam.java
public class VideoCam extends Activity implements SurfaceHolder.Callback {

        private MediaRecorder recorder = null;
        private static final String OUTPUT_FILE =
 "/sdcard/uatestvideo.mp4";
        private static final String TAG = "RecordVideo";
        private VideoView videoView = null;
        private ImageButton startBtn = null;
        private ImageButton playRecordingBtn = null;
        private Boolean playing = false;
        private Boolean recording = false;

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

        startBtn = (ImageButton) findViewById(R.id.bgnBtn);

        playRecordingBtn = (ImageButton)
            findViewById(R.id.playRecordingBtn);

        videoView = (VideoView)this.findViewById(R.id.videoView);

        final SurfaceHolder holder = videoView.getHolder();
        holder.addCallback(this);
        holder.setType(SurfaceHolder.SURFACE_TYPE_PUSH_BUFFERS);

        startBtn.setOnClickListener(new OnClickListener() {

            public void onClick(View view) {

                if(!VideoCam.this.recording & !VideoCam.this.playing)
                {
                    try
                    {
                        beginRecording(holder);
                        playing=false;
                        recording=true;
                        startBtn.setBackgroundResource(R.drawable.stop);
                    } catch (Exception e) {
                        Log.e(TAG, e.toString());
                        e.printStackTrace();
                    }
                }
                else if(VideoCam.this.recording)
                {
                    try
                    {
                        stopRecording();
                        playing = false;
                        recording= false;
                        startBtn.setBackgroundResource(R.drawable.play);
                    }catch (Exception e) {
                        Log.e(TAG, e.toString());
                        e.printStackTrace();
                    }
                }
            }
        });

        playRecordingBtn.setOnClickListener(new OnClickListener() {

            public void onClick(View view)
            {

                if(!VideoCam.this.playing & !VideoCam.this.recording)
                {
                    try
                    {
                        playRecording();
                        VideoCam.this.playing=true;
                        VideoCam.this.recording=false;
                        playRecordingBtn.setBackgroundResource

(R.drawable.stop);
                    } catch (Exception e) {
                        Log.e(TAG, e.toString());
                        e.printStackTrace();
                    }
                }
                else if(VideoCam.this.playing)
                {
                    try
                    {
                        stopPlayingRecording();
                        VideoCam.this.playing = false;
                        VideoCam.this.recording= false;
                    playRecordingBtn.setBackgroundResource

(R.drawable.play);
                    }catch (Exception e) {
                        Log.e(TAG, e.toString());
                        e.printStackTrace();
                    }
                }

            }
        });

    }

    public void surfaceCreated(SurfaceHolder holder) {
        startBtn.setEnabled(true);
```

```
}

public void surfaceDestroyed(SurfaceHolder holder) {
}

public void surfaceChanged(SurfaceHolder holder, int format, int width,
            int height) {
    Log.v(TAG, "Width x Height = " + width + "x" + height);
}

private void playRecording() {
    MediaController mc = new MediaController(this);
    videoView.setMediaController(mc);
    videoView.setVideoPath(OUTPUT_FILE);
    videoView.start();
}

private void stopPlayingRecording() {
    videoView.stopPlayback();
}

private void stopRecording() throws Exception {
    if (recorder != null) {
        recorder.stop();
    }
}

protected void onDestroy() {
    super.onDestroy();
    if (recorder != null) {
        recorder.release();
    }
}

private void beginRecording(SurfaceHolder holder) throws Exception {
    if(recorder!=null)
    {
        recorder.stop();
        recorder.release();
    }

    File outFile = new File(OUTPUT_FILE);
    if(outFile.exists())
    {
        outFile.delete();
    }

    try {
        recorder = new MediaRecorder();
        recorder.setVideoSource(MediaRecorder.VideoSource.CAMERA);
        recorder.setAudioSource(MediaRecorder.AudioSource.MIC);
        recorder.setOutputFormat(MediaRecorder.OutputFormat.MPEG_4);
        recorder.setVideoSize(320, 240);
        recorder.setVideoFrameRate(15);
        recorder.setVideoEncoder(MediaRecorder.VideoEncoder.MPEG_4_SP);
        recorder.setAudioEncoder(MediaRecorder.AudioEncoder.AMR_NB);
        recorder.setMaxDuration(20000);
        recorder.setPreviewDisplay(holder.getSurface());
        recorder.setOutputFile(OUTPUT_FILE);
```

```
            recorder.prepare();
            recorder.start();
        }
        catch(Exception e) {
            Log.e(TAG, e.toString());
            e.printStackTrace();
        }
    }
}
```

Because much of this code is similar to other code in this chapter, we won't describe everything that's happening. If you look quickly at the code in this listing, you'll note that it's relatively simple. The first thing we do in the code, besides setting some fields, is set up our surface to support the camera preview, much like we did in the simple camera application earlier in this chapter. The next part of the code that's important is the `beginRecording` method. First, this method checks to make sure that everything is ready to record a video file by making sure that the camera is free, and that it can record the output file. Then, it closely follows the preceding processes to set up the camera for recording before calling `prepare()` and then `start()`.

Unfortunately, as we noted with the camera project, there's no easy way to test your application in the emulator. For this example, we've pushed the application to a cell phone to test the camera, and used the DDMS to note the file that was recorded and to play it back. You can see an example of the output, captured with the DDMS, from an HTC Hero in figure 10.7.

Without a device to test on, you'll have difficulties debugging your video applications. If you decide to develop a video application, we strongly suggest that you not only obtain an Android device to test on, but that you test every physical device that you hope your application will run on. Although developing Android applications that record data from sensors can be difficult to work with on the emulator, they're relatively straightforward to code, but you need to use a physical Android device to test.

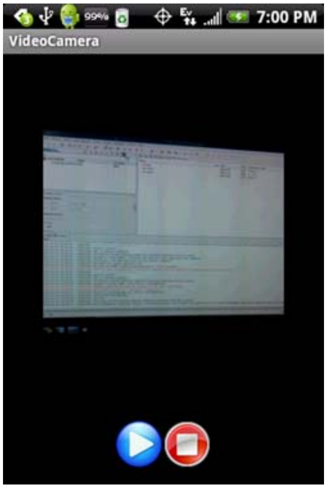


Figure 10.7 **Photograph of VideoCam application running on an HTC Hero 2.**

10.6 Summary

In this chapter, we looked at how the Android SDK uses multimedia and how you can play, save, and record video and sound. We also looked at various features the Android MediaPlayer offers the developer, from a built-in video player to wide support for formats, encodings, and standards.

We talked about how to interact with other hardware devices attached to the phone, such as a microphone and camera. You used the SDK to create an SD card image for the emulator to simulate SD cards, and you used the MediaRecorder application to record audio and save it to the SD card.

The most consistent characteristic of multimedia programming with Android is that things are changing and maturing! Multimedia support is moving from OpenCORE to Stagefright, with version 2.2 being somewhat of a pivot release where both frameworks share responsibility for delivering multimedia functionality. Writing multimedia applications requires the developer to conduct a bit more work directly on the hardware because the emulated environments don't adequately replicate the hardware capabilities of the handsets. Despite this potential speed-bump in the development process, Android currently offers you everything you need to create rich and compelling media applications. Its focus on supporting industry and open standards guarantees that your applications will have wide support on a variety of phones.

In the next chapter, you'll learn all about how to use Android's location services to interact with GPS and maps. By mixing in what you've learned in this chapter, you'll be able to create your own GPS application that not only provides voice direction, but that even responds to voice commands.

11 Location, location, location

This chapter covers

- Working with `LocationProvider` and `LocationManager`
- Testing location in the emulator
- Receiving location alerts with `LocationListener`
- Drawing with `MapActivity` and `MapView`
- Looking up addresses with the `Geocoder`

Accurate location awareness makes a mobile device more powerful. Combining location awareness with network data can change the world—and Android shines here. Other platforms have gained similar abilities in recent years, but Android excels with its easy-to-use and popular location API framework based on Google Maps.

From direct network queries to triangulation with cell towers and even satellite positioning via GPS, an Android-powered device has access to different types of `LocationProvider` classes that allow access to location data. Various providers supply a mix of location-related metrics, including latitude and longitude, speed, bearing, and altitude.

Developers generally prefer to work with GPS because of its accuracy and power. But some devices may not have a GPS receiver, and even GPS-enabled devices can't access satellite data when inside a large building or otherwise obstructed from receiving the signal. In those instances the Android platform provides a graceful and automatic fallback to query other providers when your first choice fails. You can examine provider availability and hook into one or another using the `LocationManager` class.

Location awareness[1] opens up a new world of possibilities for application development. In this chapter you'll build an application that combines location awareness with data from the U.S. National Oceanic and Atmospheric Administration (NOAA) to produce an interesting and useful mashup.

Specifically you'll connect to the National Data Buoy Center (NDBC) to retrieve data from buoys and ships located around the coastline in North America. Thanks to the NOAA-NDBC system, which polls sensors on buoys and makes that data available in RSS feeds, you can retrieve data for the vicinity, based on the current location, and display condition information such as wind speed, wave height, and temperature. Although we won't cover non-location-related details in this chapter, such as using HTTP to pull the RSS feed data, the full source code for the application is available with the code download for this chapter. Our Wind and Waves application has several main screens, including an Android `MapActivity` with a `MapView`. These components are used for displaying and manipulating map information, as shown in figure 11.1.

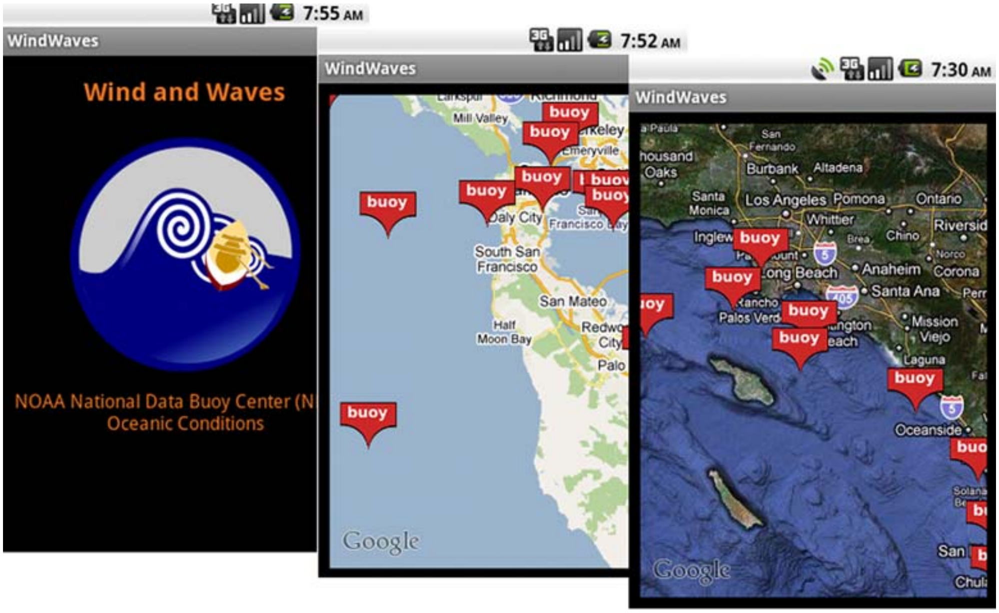


Figure 11.1 **Screens from the Wind and Waves location-aware application**

[1] For more about location, check out *Location-Aware Applications* by Richard Ferraro and Murat Aktihanoglu, to be published by Manning in March 2011: http//www.manning.com/ferraro.

Accessing buoy data, which is important mainly for marine use cases, has a somewhat limited audience. But the principles shown in this app demonstrate the range of Android's location-related capabilities, and should inspire you to develop your own unique application.

In addition to displaying data based on the current location, you'll use this application to create several `LocationListener` instances that receive updates when the user's location changes. When the position changes, the device will inform your application, and you'll update your `MapView` using an `Overlay`—an object that allows you to draw on top of the map.

Beyond the buoy application requirements, you'll also write a few samples for working with the `Geocoder` class. This class allows you to map between a `GeoPoint` (latitude and longitude) and a place (city or postal code) or address. This utility doesn't help much on the high seas but does benefit many other apps.

Before writing the sample apps, you'll start by using the device's built-in mapping application and simulating your position within the Android emulator. This approach will allow you to mock your location for the emulator. After we've covered all of the emulator location-related options, we'll move on to building Wind and Waves.

11.1 Simulating your location within the emulator

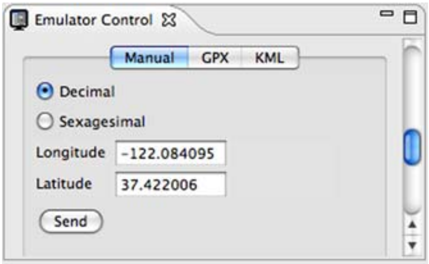
For any location-aware application, you'll start by working with the provided SDK and the emulator. Within the emulator, you'll set and update your current location. From there you'll want to progress to supplying a range of locations and times to simulate movement over a geographic area.

You can accomplish these tasks in several ways, either by using the DDMS tool or by using the command line from the shell. To get started quickly, let's first send in direct coordinates through the DDMS tool.

11.1.1 Sending in your coordinates with the DDMS tool

You can access the DDMS tool in two ways, either launched on its own from the SDK tools subdirectory or as the Emulator Control view within the Eclipse IDE. You need to have Eclipse and the Android Eclipse plug-in to use DDMS within Eclipse; see chapter 2 and appendix A for more details about getting the SDK and plug-in set up.

With the DDMS tool you can send direct latitude and longitude coordinates manually from the Emulator Control > Location Controls form. This is shown in figure 11.2. Note that *Longitude* is the first field, which is the standard around the world, but backward from how latitude and longitude are generally expressed in the United States.



● Decimal

○ Sexagesimal

Longitude -122.084095

Latitude 37.422006

Figure 11.2 **Using the DDMS tool to send direct latitude and longitude coordinates to the emulator as a mock location**

If you launch the built-in Maps application from Android's main menu and send in a location with the DDMS tool, you can then use the menu to select My Location, and the map will animate to the location you've specified—anywhere on Earth.

> **NOTE** Both the Google Maps application and the mapping APIs are part of the optional Google APIs. As such, not all Android phones support these features. Check your target devices to ensure that they provide this support. For development, you'll need to install an Android Virtual Device[2] (AVD) that supports the Google APIs.

Try this a few times to become comfortable with setting locations; for example, send the decimal coordinates in table 11.1 one by one, and in between browse around the map. When you supply coordinates to the emulator, you'll need to use the decimal form.

Although the DDMS tool requires the decimal format, latitude and longitude are more commonly expressed on maps and other tools as degrees, minutes, and seconds. Degrees (°) represent points on the surface of the globe as measured from either the equator (for latitude) or the prime meridian (for longitude). Each degree is further subdivided into 60 smaller sections, called minutes ('), and each minute also has 60 seconds ("). If necessary, seconds can be divided into tenths of a second or smaller fractions.

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	

When representing latitude and longitude on a computer, the degrees are usually converted into decimal form with positive representing north and east and negative representing south and west, as shown in figure 11.3.

If you live in the southern and eastern hemispheres, such as in Buenos Aires, Argentina, which is 34°60' S, 58°40' W in the degree form, the decimal form is negative for both latitude and longitude, -34.60, -58.40. If you haven't used latitude and longitude much, the different forms can be confusing at first, but they quickly become clear.

[2] For more on Android, maps and Android Virtual Devices, try here: http://developer.appcelerator.com/doc/mobile/android-maps.

Once you've mastered setting a fixed position, you can move on to supplying a set of coordinates that the emulator will use to simulate a range of movement.

> **NOTE** You can also send direct coordinates from within the emulator console. If you telnet localhost 5554 (adjust the port where necessary) or adb shell, you'll connect to the default emulator's console. From there you can use the geo fix command to send longitude, latitude, and optional altitude; for example, geo fix -21.55 64.1. Keep in mind that the Android tools require longitude in the first parameter.

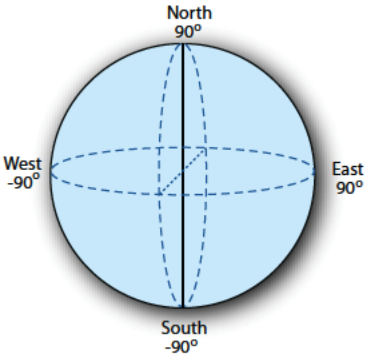


Figure 11.3 Latitude and longitude spherical diagram, showing positive north and east and negative south and west

11.1.2 The GPS Exchange Format

The DDMS tool supports two formats for supplying a range of location data in file form to the emulator. The GPS Exchange Format (GPX) allows a more expressive form when working with Android.

GPX is an XML schema that allows you to store waypoints, tracks, and routes. Many handheld GPS devices support this format. The following listing shows the basics of the format in a portion of a sample GPX file.

Listing 11.1 A sample GPX file

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<gpx xmlns="http://www.topografix.com/GPX/1/1"
  version="1.1"
  creator="Charlie Collins - Hand Rolled"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
    "http://www.topografix.com/GPX/1/1/gpx.xsd">
    <metadata>
        <name>Sample Coastal California Waypoints</name>
        <desc>Test waypoints for use with Android</desc>
        <time>2008-11-25T06:52:56Z</time>
        <bounds minlat="25.00" maxlat="75.00"
          minlon="100.00" maxlon="-150.00" />
    </metadata>
    <wpt lat="41.85" lon="-124.38">
        <ele>0</ele>
        <name>Station 46027</name>
        <desc>Off the coast of Lake Earl</desc>
    </wpt>
    <wpt lat="41.74" lon="-124.18">
        <ele>0</ele>
        <name>Station CECC1</name>
        <desc>Crescent City</desc>
    </wpt>
    <wpt lat="38.95" lon="-123.74">
```

1 Define root gpx element

2 Include metadata stanza

3 Supply waypoint element

```
        <ele>0</ele>
        <name>Station PTAC1</name>
        <desc>Point Arena Lighthouse</desc>
    </wpt>
    . . .  remainder of wpts omitted for brevity
<trk>
     <name>Example Track</name>
        <desc>A fine track with trkpts.</desc>
        <trkseg>
            <trkpt lat="41.85" lon="-124.38">
                <ele>0</ele>
                <time>2008-10-15T06:00:00Z</time>
            </trkpt>
            <trkpt lat="41.74" lon="-124.18">
                <ele>0</ele>
                <time>2008-10-15T06:01:00Z</time>
            </trkpt>
            <trkpt lat="38.95" lon="-123.74">
                <ele>0</ele>
                <time>2008-10-15T06:02:00Z</time>
            </trkpt>
            . . . remainder of trkpts omitted for brevity
        </trkseg>
    </trk>
</gpx>
```

4 Supply track element

5 Use track segment

6 Provide specific point

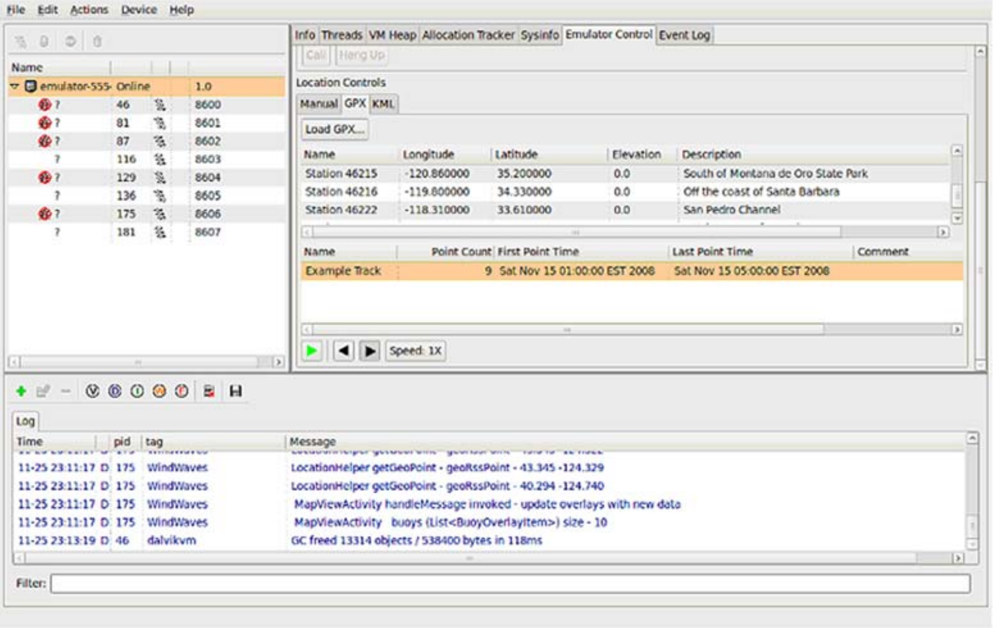
A GPX file requires the correct XML namespace in the root gpx element 1. Within its body, the file includes metadata 2 and individual waypoints 3. *Waypoints* are named locations at a particular latitude and longitude. Along with individual waypoints, a GPX file supports related route information in the form of tracks 4, which can be subdivided further into *track segments* 5. Each track segment is made up of track points. Finally, each track point 6 contains a waypoint with an additional point-in-time property.

When working with a GPX file in the DDMS tool, you can use two different modes, as figure 11.4 reveals. The top half of the GPX box lists individual waypoints; when you click one, that individual location is sent to the emulator. In the bottom half of the GPX box, all the tracks are displayed. Tracks can be "played" forward and backward to simulate movement. As the track reaches each track point, based on the time it defines, it sends those coordinates to the emulator. You can modify the speed for this playback via the Speed button.

GPX is simple and extremely useful when working with mock location information for your Android applications, but it's not the only file format supported. The DDMS tool also supports a format called KML.

11.1.3 The Google Earth Keyhole Markup Language

The second format that the Android DDMS tool supports for sending a range of mock location information to the emulator is the *Keyhole Markup Language (KML)*. KML was originally a proprietary format created by a company named Keyhole. After Google


File Edit Actions Device Help
Info Threads VM Heap Allocation Tracker Sysinfo Emulator Control Event Log
Call Hang Up
Location Controls
Manual GPX KML
Load GPX...
Speed: 1X
Log
Filter:

Name			
emulator-555- Online			1.0
?	46		8600
?	81		8601
?	87		8602
?	116		8603
?	129		8604
?	136		8605
?	175		8606
?	181		8607

Name	Longitude	Latitude	Elevation	Description
Station 46215	-120.860000	35.200000	0.0	South of Montana de Oro State Park
Station 46216	-119.800000	34.330000	0.0	Off the coast of Santa Barbara
Station 46222	-118.310000	33.610000	0.0	San Pedro Channel

Name	Point Count	First Point Time	Last Point Time	Comment
Example Track	9	Sat Nov 15 01:00:00 EST 2008	Sat Nov 15 05:00:00 EST 2008	

```
Time              pid  tag         Message
11-25 23:11:17 D  175  WindWaves   LocationHelper getGeoPoint - geoRssPoint - 43.345 -124.329
11-25 23:11:17 D  175  WindWaves   LocationHelper getGeoPoint - geoRssPoint - 40.294 -124.740
11-25 23:11:17 D  175  WindWaves    MapViewActivity handleMessage invoked - update overlays with new data
11-25 23:11:17 D  175  WindWaves    MapViewActivity   buoys (List<BuoyOverlayItem>) size - 10
11-25 23:13:19 D  46   dalvikvm    GC freed 13314 objects / 538400 bytes in 118ms
```

Figure 11.4 **Using the DDMS tool with a GPX file to send mock location information**

acquired Keyhole, it submitted KML to the Open Geospatial Consortium (OGC), which accepted KML as an international standard.

OGC KML pursues the following goal:

> *That there be one international standard language for expressing geographic annotation and visualization on existing or future web-based online and mobile maps (2d) and earth browsers (3d).*

The following listing shows a sample KML file for sending location data to the Android Emulator. This file uses the same coastal location data as you saw with the previous GPX example.

Listing 11.2 A sample KML file

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://earth.google.com/kml/2.2">
    <Placemark>
        <name>Station 46027</name>
        <description>Off the coast of Lake Earl</description>
        <Point>
            <coordinates>-124.38,41.85,0</coordinates>
        </Point>
    </Placemark>
    <Placemark>
        <name>Station 46020</name>
        <description>Outside the Golden Gate</description>
        <Point>
            <coordinates>-122.83,37.75,0</coordinates>
```

1 Capture information with Placemark

Provide Point

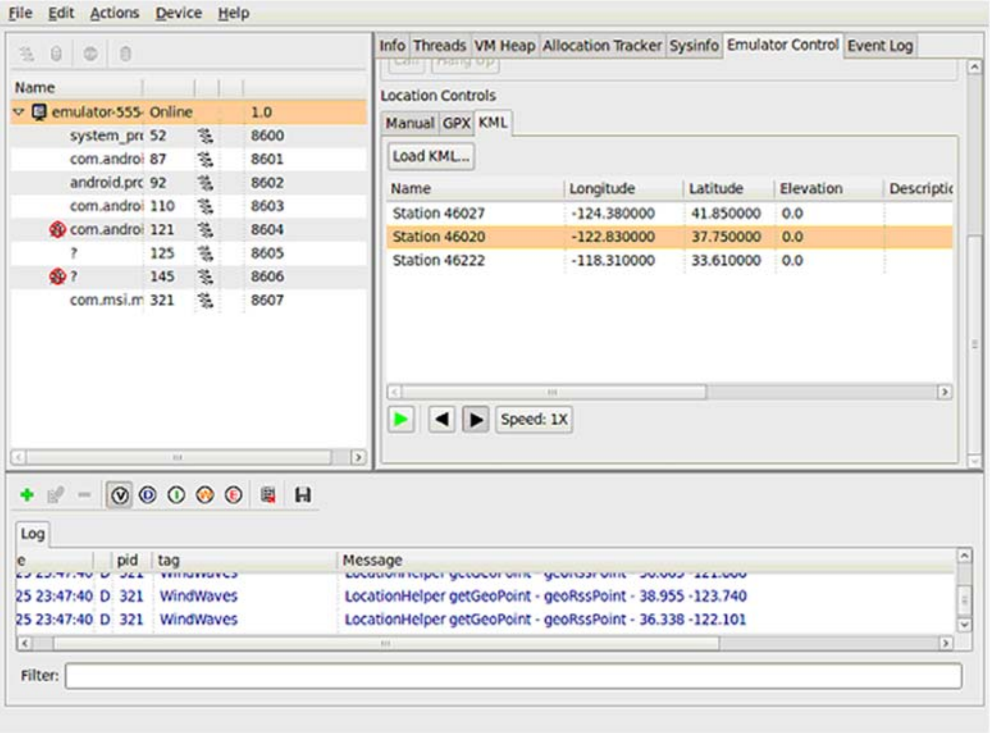
Supply coordinates for Point 3

```
        </Point>
    </Placemark>
    <Placemark>
        <name>Station 46222</name>
        <description>San Pedro Channel</description>
        <Point>
            <coordinates>-118.31,33.61,0</coordinates>
           </Point>
    </Placemark>
</kml>
```

KML uses a `kml` root element requiring the correct namespace declaration. KML supports many more elements and attributes than the DDMS tool handles. DDMS only checks your KML files for `Placemark` elements ❶, which contain `Point` child elements ❷, which in turn supply `coordinates` ❸.

Figure 11.5 shows an example of using a KML file with the DDMS tool.

KML[3] is flexible and expressive, but it has drawbacks when used with the Android Emulator. As we've noted, the DDMS parser looks for the `coordinate` elements in the file and sends the latitude, longitude, and elevation for each in a sequence, one



Name				
emulator-555-	Online			1.0
system_pr	52			8600
com.androi	87			8601
android.prc	92			8602
com.androi	110			8603
com.androi	121			8604
?	125			8605
?	145			8606
com.msi.m	321			8607

Name	Longitude	Latitude	Elevation	Descriptic
Station 46027	-124.380000	41.850000	0.0	
Station 46020	-122.830000	37.750000	0.0	
Station 46222	-118.310000	33.610000	0.0	

```
e               pid  tag         Message
25 23:47:40  D  321  WindWaves   LocationHelper getGeoPoint - geoRssPoint - 38.955 -123.740
25 23:47:40  D  321  WindWaves   LocationHelper getGeoPoint - geoRssPoint - 36.338 -122.101
```

Figure 11.5 Using the DDMS tool with a KML file to send mock location information

[3] For more details on KML, go to: http://code.google.com/apls/kml/documentation/

Placemark per second. Timing and other advanced features of KML aren't yet supported by DDMS. Because of this we find it more valuable at present to use GPX as a debugging and testing format, because it supports detailed timing.

KML is still important; it's an international standard and will continue to gain traction. Also, KML is an important format for other Google applications, so you may encounter it more frequently in other contexts than GPX. For example, you could create a KML route using Google Earth, and then later use it in your emulator to simulate movement.

Now that you know how to send mock location information to the emulator in various formats, you can step out of the built-in Maps application and start creating your own programs that rely on location.

11.2 Using LocationManager and LocationProvider

When building location-aware applications on the Android platform, you'll most often use several key classes. A LocationProvider provides location data using several metrics, and you can access providers through a LocationManager.

LocationManager allows you to attach a LocationListener that receives updates when the device location changes. LocationManager also can directly fire an Intent based on the proximity to a specified latitude and longitude. You can always retrieve the last-known Location directly from the manager.

The Location class is a Java bean that represents all the location data available from a particular snapshot in time. Depending on the provider used to populate it, a Location may or may not have all the possible data present; for example, it might not include speed or altitude.

To get your Wind and Waves sample application started and to grasp the related concepts, you first need to master the LocationManager.

11.2.1 Accessing location data with LocationManager

LocationManager lets you retrieve location-related data on Android. Before you can check which providers are available or query the last-known Location, you need to acquire the manager from the system service. The following listing demonstrates this task, and includes a portion of the MapViewActivity that will drive our Wind and Waves application.

Listing 11.3 Start of MapViewActivity

```
public class MapViewActivity extends MapActivity {                   Extend
    private static final int MENU_SET_SATELLITE = 1;               ❶ MapActivity
    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;                      ❷ Define
    private LocationManager locationManager;                     LocationManager
    private LocationProvider locationProvider;
    private MapView mapView;
    private ViewGroup zoom;
    private Overlay buoyOverlay;
    private ProgressDialog progressDialog;
    private Drawable defaultMarker;
    private ArrayList<BuoyOverlayItem> buoys;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        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 =
          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);
        this.locationProvider =
            this.locationManager.getProvider(
            LocationManager.GPS_PROVIDER);
        // LocationListeners omitted here for brevity
        GeoPoint lastKnownPoint = this.getLastKnownPoint();
        this.mapController = this.mapView.getController();
        this.mapController.setZoom(10);
        this.mapController.animateTo(lastKnownPoint);
        this.getBuoyData(lastKnownPoint);
    }
    . . . onResume and onPause omitted for brevity
    . . . other portions of MapViewActivity are included
            in later listings in this chapter
    private GeoPoint getLastKnownPoint() {
        GeoPoint lastKnownPoint = null;
        Location lastKnownLocation =

     this.locationManager.getLastKnownLocation(
            LocationManager.GPS_PROVIDER);
        if (lastKnownLocation != null) {
            lastKnownPoint = LocationHelper.getGeoPoint(lastKnownLocation);
        } else {
            lastKnownPoint = LocationHelper.GOLDEN_GATE;
        }
        return lastKnownPoint;
    }
```

3 Define LocationProvider

4 Instantiate LocationManager system service

5 Assign GPS LocationProvider

6 Set up map

7 Get last-known Location

Our custom `MapViewActivity` extends `MapActivity` ❶. We'll focus on the `MapActivity` in more detail in section 11.3, but for now, recognize that this is a special kind of `Activity`. Within the class, you declare member variables for `LocationManager` ❷ and `LocationProvider` ❸.

To acquire the `LocationManager`, you use the `Activity getSystemService (String name)` method ❹. Once you have the `LocationManager`, you assign the `LocationProvider` you want to use with the manager's `getProvider` method ❺. In this case use the GPS provider. We'll talk more about the `LocationProvider` class in the next section.

Once you have the manager and provider in place, you implement the `onCreate` method of your `Activity` to instantiate a `MapController` and set initial state for the screen ❻. Section 11.3 covers `MapController` and the `MapView` it manipulates.

Along with helping you set up the provider you need, `LocationManager` supplies quick access to the last-known `Location` ❼. Use this method if you need a quick fix on the last location, as opposed to the more involved techniques for registering for periodic location updates with a listener; we'll cover that topic in section 11.2.3.

Besides the features shown in this listing, `LocationManager` allows you to directly register for proximity alerts. For example, your app could show a custom message if you pass within a quarter-mile of a store that has a special sale. If you need to fire an `Intent` based on proximity to a defined location, call the `addProximityAlert` method. This method lets you set the target location with latitude and longitude, and also lets you specify a radius and a `PendingIntent`. If the device comes within the range, the `PendingIntent` is fired. To stop receiving these messages, call `removeProximityAlert`.

Getting back to the main purpose for which you'll use the `LocationManager` with Wind and Waves, we'll next look more closely at the GPS `LocationProvider`.

11.2.2 Using a LocationProvider

`LocationProvider` helps define the capabilities of a given provider implementation. Each implementation responsible for returning location information may be available on different devices and in different circumstances.

Available provider implementations depend on the hardware capabilities of the device, such as the presence of a GPS receiver. They also depend on the situation: even if the device has a GPS receiver, can it currently receive data from satellites, or is the user somewhere inaccessible such as an elevator or a tunnel?

At runtime you'll query for the list of providers available and use the most suitable one. You may select multiple providers to fall back on if your first choice isn't available or enabled. Developers generally prefer using the `LocationManager.GPS_PROVIDER` provider, which uses the GPS receiver. You'll use this provider for Wind and Waves because of its accuracy and its support in the emulator. Keep in mind that a real device will normally offer multiple providers, including the `LocationManager.NETWORK_PROVIDER`, which uses cell tower and Wi-Fi access points to determine location data. To piggyback on other applications requesting location, use LocationManager.PASSIVE_PROVIDER.

In listing 11.3 we showed how you can obtain the GPS provider directly using the getProvider(String name) method. Table 11.2 provides alternatives to this approach of directly accessing a particular provider.

Table 11.2 Methods for obtaining a LocationProvider reference

LocationProvider code snippet	Description
`List<String> providers = locationManager.getAllProviders();`	Get all of the providers registered on the device.
`List<String> enabledProviders = locationManager.getAllProviders(true);`	Get all of the currently enabled providers.
`locationProvider = locationManager.getProviders(true).get(0);`	A shortcut to get the first enabled provider, regardless of type.
`locationProvider = locationManager.getBestProvider( myCriteria, true);`	An example of getting a LocationProvider using a particular Criteria argument. You can create a Criteria instance and specify whether bearing, altitude, cost, and other metrics are required.

Different providers may support different location-related metrics and have different costs or capabilities. The Criteria class helps define what each provider instance can handle. Available metrics are latitude and longitude, speed, bearing, altitude, cost, and power requirements.

Remember to set the appropriate Android permissions. Your manifest needs to include location-related permissions for the providers you want to use. The following listing shows the Wind and Waves manifest XML file, which includes both COARSE- and FINE-grained location-related permissions.

Listing 11.4 A manifest file showing COARSE and FINE location-related permissions

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.windwaves">
    <application android:icon="@drawable/wave_45"
        android:label="@string/app_name"
        android:theme="@android:style/Theme.Black”>
        <activity android:name="StartActivity"
          android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
        </activity>
        <activity android:name="MapViewActivity" />
        <activity android:name="BuoyDetailActivity" />
        <uses-library android:name="com.google.android.maps" />
    </application>
    <uses-permission
       android:name=
         "android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission
      android:name=
        "android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission
       android:name="android.permission.INTERNET" />
</manifest>
```

2 **Access GPS provider**

Include both the ACCESS_COARSE_LOCATION 1 and ACCESS_FINE_LOCATION 2 permissions in your manifest. The COARSE permission corresponds to the LocationManager.NETWORK_PROVIDER provider for cell and Wi-Fi based data, and the FINE permission corresponds to the LocationManager.GPS_PROVIDER provider. You don't use the network provider in Wind and Waves, but this permission would allow you to enhance the app to fall back to the network provider if the GPS provider becomes unavailable or disabled.

Once you understand the basics of LocationManager and LocationProvider, you can unleash the real power and register for periodic location updates in your application with the LocationListener class.

11.2.3 Receiving location updates with LocationListener

You can keep abreast of the device location by creating a LocationListener implementation and registering it to receive updates. LocationListener lets you filter for many types of location events based on a flexible and powerful set of properties. You implement the interface and register your instance to receive location data callbacks.

Listing 11.5 demonstrates those principles as you create several LocationListener implementations for the Wind and Waves MapViewActivity and then register those listeners using the LocationManager and LocationProvider. This listing helps complete the initial code from listing 11.3.

Listing 11.5 Creation of LocationListener implementations in MapViewActivity

```
. . . start of class in Listing 11.3
private final LocationListener locationListenerGetBuoyData =
    new LocationListener() {
        public void onLocationChanged(
          final Location loc) {
            int lat = (int) (loc.getLatitude()
              * LocationHelper.MILLION);
            int lon = (int) (loc.getLongitude()
              * LocationHelper.MILLION);
            GeoPoint geoPoint = new GeoPoint(lat, lon);
            getBuoyData(geoPoint);
        }
```

1 **Create anonymous LocationListener**

2 **Implement onLocationChanged**

3 **Get latitude and longitude**

4 **Create GeoPoint**

5 **Update map pins (buoy data)**

```
        public void onProviderDisabled(String s) {
        }
        public void onProviderEnabled(String s) {
        }
        public void onStatusChanged(String s,
            int i, Bundle b) {
        }
    };
private final LocationListener locationListenerRecenterMap =
  new LocationListener() {
        public void onLocationChanged(final Location loc) {
            int lat = (int) (loc.getLatitude()
              * LocationHelper.MILLION);
            int lon = (int) (loc.getLongitude()
              * LocationHelper.MILLION);
            GeoPoint geoPoint = new GeoPoint(lat, lon);
            mapController.animateTo(geoPoint);
        }
        public void onProviderDisabled(String s) {
        }
        public void onProviderEnabled(String s) {
        }
        public void onStatusChanged(String s,
            int i, Bundle b) {
        }
    };
    @Override
    public void onStart() {
        super.onStart();
        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);
            this.locationManager.requestLocationUpdates(
              locationProvider.getName(), 3000, 1000,
                this.locationListenerRecenterMap);
        } else {
            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)
    }
```

6 Move map to new location

Methods intentionally left blank

8 Register locationListener-RecenterMap

You'll usually find it practical to use an anonymous inner class 1 to implement the LocationListener interface. For this MapViewActivity, you create two LocationListener implementations so you can later register them using different settings.

The first listener, locationListenerGetBuoyData, implements the onLocationChanged method ❷. In that method you get the latitude and longitude from the Location sent in the callback ❸. You then use the data to create a GeoPoint ❹ after multiplying the latitude and longitude by 1 million (1e6). You need to multiply by a million because GeoPoint requires microdegrees for coordinates. A separate class, LocationHelper, defines this constant and provides other location utilities; you can view this class in the code download for this chapter.

After you have the data, you update the map ❺ using a helper method that resets a map Overlay; you'll see this method's implementation in the next section. In the second listener, locationListenerRecenterMap, you perform the different task of centering the map ❻.

The need for two separate listeners becomes clear when you see how listeners are registered with the requestLocationUpdates method of the Location-Manager class. You register the first listener, locationListenerGetBuoyData, to fire only when the new device location has moved a long way from the previous one ❼. The defined distance is 185,000 meters. (We chose this number to stay just under 100 nautical miles, which is the radius you'll use to pull buoy data for your map; you don't need to redraw the buoy data on the map if the user moves less than 100 nautical miles.) You register the second listener, locationListenerRecenterMap, to fire more frequently; the map view recenters if the user moves more than 1,000 meters ❽. Using separate listeners like this allows you to fine-tune the event processing, rather than having to build in your own logic to do different things based on different values with one listener.

Keep in mind that your registration of LocationListener instances could become even more robust by implementing the onProviderEnabled and onProviderDisabled methods. Using those methods and different providers, you could provide useful messages to the user and also provide a graceful fallback through a set of providers; for example, if GPS becomes disabled, you could try the network provider instead.

> **NOTE** You should carefully use the time parameter to the requestLocationUpdates method. Requesting location updates too frequently (less than 60,000 ms per the documentation) can wear down the battery and make the application too jittery. In this sample you use an extremely low value (3,000 ms) for debugging purposes. Long-lived or always-running code shouldn't use a value lower than the recommended 60,000 ms in production code.

With LocationManager, LocationProvider, and LocationListener instances in place, we can address the MapActivity and MapView in more detail.

11.3 Working with maps

In the previous sections, you wrote the start of the MapViewActivity for our Wind and Waves application. We covered the supporting classes and showed you how to register to receive location updates. With that structure in place, let's now focus on the actual map details.

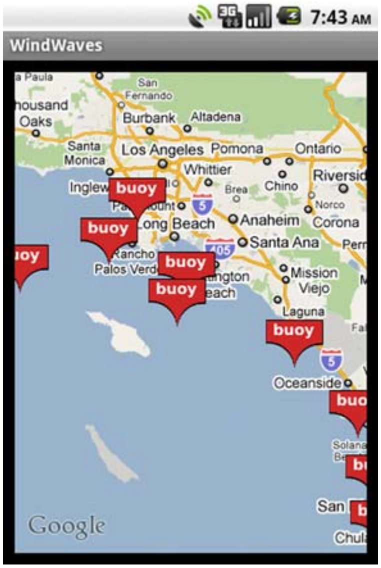


Figure 11.6 **The `MapViewActivity` from the Wind and Waves application shows a `MapActivity` with `MapView`.**

The `MapViewActivity` screen will look like figure 11.6, where several map `Overlay` classes display on top of a `MapView` within a `MapActivity`

To use the `com.google.android.maps` package on the Android platform and support all the features related to a `MapView`, you must use a `MapActivity`.

11.3.1 Extending MapActivity

A `MapActivity` defines a gateway to the Android Google Maps-like API package and other useful map-related utilities. It handles several details behind creating and using a `MapView` so you don't to have to worry about them.

The `MapView`, covered in the next section, offers the most important features. But a `MapActivity` provides essential support for the `MapView`. It manages all the network and filesystem-intensive setup and teardown tasks needed for supporting the map. For example, the `MapActivity` `onResume` method automatically sets up network threads for various map-related tasks and caches map section tile data on the filesystem, and the `onPause` method cleans up these resources. Without this class, all these details would require extra housekeeping that any `Activity` wishing to include a `MapView` would have to repeat each time.

Your code won't do much with `MapActivity`. Extend this class (as in listing 11.3), making sure to use only one instance per process, and include a special manifest element to enable the `com.google.android.maps` package. You may have noticed the `uses-library` element in the Wind and Waves manifest in listing 11.4:

```
<uses-library android:name="com.google.android.maps" />
```

The `com.google.android.maps` package, where `MapActivity`, `MapView`, `MapController`, and other related classes such as `GeoPoint` and `Overlay` reside, isn't a standard package in the Android library. This manifest element pulls in support for the Google `maps` package.

Once you include the `uses-library` element and write a basic `Activity` that extends `MapActivity`, you can start writing the main app features with a `MapView` and related `Overlay` classes.

11.3.2 Using a MapView

Android offers `MapView`[4] as a limited version of the Google Maps API in the form of a `View` for your Android application. A `MapView` displays tiles of a map, which it obtains

[4] Take a look at this MapView tutorial for more information: http://developer.android.com/guide/tutorials/views/hello-mapview.html.

over the network as the map moves and zooms, much like the web version of Google Maps.

Android supports many of the concepts from the standard Google Maps API through the `MapView`. For instance, `MapView` supports a plain map mode, a satellite mode, a street-view mode, and a traffic mode. When you want to write something on top of the map, draw a straight line between two points, drop a "pushpin" marker, or display full-sized images, you use an `Overlay`.

You can see examples of several of these concepts in figure 11.6, which shows `MapViewActivity` screenshots for the Wind and Waves application. Figure 11.7 shows that same `MapViewActivity` again after switching into satellite mode.

You've already seen the `MapView` we'll use for the Wind and Waves application declared and instantiated in listing 11.3. Now we'll discuss using this class inside your `Activity` to control, position, zoom, populate, and overlay your map.

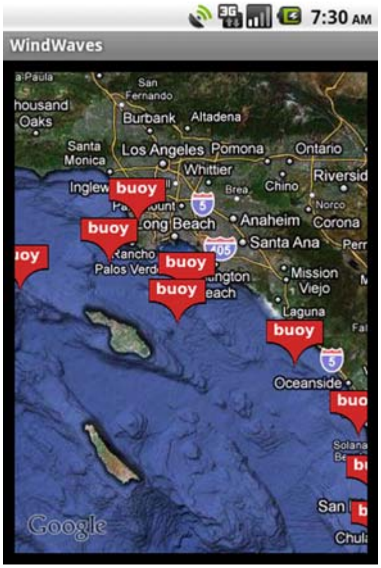


Figure 11.7 The `MapViewActivity` from the Wind and Waves application using satellite mode

Before you can use a map at all, you have to request a Google Maps API key and declare it in your layout file. This listing shows the `MapActivity` layout file you'll use with a special `android:apiKey` attribute.

Listing 11.6 A `MapView` layout file including the Google Maps API key

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:gravity="center_horizontal" android:padding="10px">
    <com.google.android.maps.MapView
      android:id="@+id/map_view"
      android:layout_width="fill_parent"
      android:layout_height="fill_parent"
      android:enabled="true"
      android:clickable="true"
      android:apiKey=
       "051Sygx-ttd-J5GXfsIB-dlpNtggca4I4DMyVqQ" />
</RelativeLayout>
```

❶ Define MapView element in XML

❷ Include apiKey attribute

You can declare a `MapView` in XML just like other `View` components ❶. In order to use the Google Maps network resources, a `MapView` requires an API key ❷. You can obtain a map key from the Google Maps Android key registration web page at http://code.google.com/android/maps-api-signup.html.

Before you register for a key, you need to look up the MD5 fingerprint of the certificate that signs your application. This sounds tricky, but it's really simple. When using the Android Emulator, the SDK always uses a Debug Certificate. To get the MD5 fingerprint for this certificate on Mac and Linux, you can use the following command:

```
cd ~/.android
keytool -list -keystore ./debug.keystore -storepass android -keypass android
```

On Windows, adjust for the user's home directory slash directions, such as

```
cd c:\Users\Chris\.android
keytool -list -keystore debug.keystore -storepass android -keypass android
```

Getting a key for a production application follows the same process, but you need to use the actual certificate your APK file is signed with instead of the debug.keystore file. The Android documentation provides additional information about obtaining a key at http://code.google.com/android/add-ons/google-apis/mapkey.html. For more information about digital signatures, keys, and signing in general, see appendix B.

> **CAUTION** Android requires you to declare the map API key in the layout file. With the key in the layout file, you must remember to update the key between debug and production modes. Additionally, if you debug on different development machines, you must switch keys by hand.

Once you write a MapActivity with a MapView and create your view in the layout file, complete with map API key, you can make full use of the map. Several of the previous listings use the MapView from the Wind and Waves application. In the next listing we repeat a few of the map-related lines of code we've already shown, and add related items to consolidate all the map-related concepts in one listing.

Listing 11.7 Portions of code that demonstrate working with maps

```
. . . from onCreate
mapView = (MapView)
  findViewById(R.id.map_view);
mapView.
  setBuiltInZoomControls(true);
 . . .  from onStart
mapController = mapView.getController();
mapController.setZoom(10);
mapController.
  animateTo(lastKnownPoint);
. . . from onMenuItemSelected
case MapViewActivity.MENU_SET_MAP:
    mapView.setSatellite(false);
    break;
case MapViewActivity.MENU_SET_SATELLITE:
    mapView.setSatellite(true);
    break;
case MapViewActivity.MENU_BUOYS_FROM_MAP_CENTER:
    getBuoyData(mapView.getMapCenter());
    break;
```

1 Inflate MapView from layout

2 Animate to given GeoPoint

3 Set map satellite mode

You declare the MapView in XML and inflate it just like other view components ❶. Because it's a ViewGroup, you can also combine and attach other elements to it. You tell the MapView to display its built-in zoom controls so the user can zoom in and out.

Next you get a MapController from the MapView. The controller allows you to programmatically zoom and move the map. When starting, you use the controller to set the initial zoom level and animate to a specified GeoPoint ❷. When the user selects a view mode from the menu, you set the mode of the map from plain to satellite or back again ❸. Along with manipulating the map itself, you can retrieve data from it, such as the coordinates of the map center.

Besides manipulating the map and getting data from it, you can draw items on top of the map using Overlay instances.

11.3.3 Placing data on a map with an Overlay

The small buoy icons for the Wind and Waves application that we've used in several figures up to this point draw on the screen at specified coordinates using an Overlay.

Overlay describes an item to draw on the map. You can define your own Overlay by extending this class or MyLocationOverlay. The MyLocationOverlay class lets you display a user's current location with a compass, and it has other useful features such as a LocationListener for convenient access to position updates.

Besides showing the user's location, you'll often place multiple marker items on the map. Users generally expect to see markers as pushpins. You'll create buoy markers for the location of every buoy using data you get back from the NDBC feeds. Android provides built-in support for this with the ItemizedOverlay base class and the OverlayItem.

OverlayItem, a simple bean, includes a title, a text snippet, a drawable marker, coordinates defined in a GeoPoint, and a few other properties. The following listing shows the buoy data-related BuoyOverlayItem class for Wind and Waves.

Listing 11.8 The OverlayItem subclass BuoyOverlayItem

```
public class BuoyOverlayItem extends OverlayItem {
    public final GeoPoint point;
    public final BuoyData buoyData;
    public BuoyOverlayItem(GeoPoint point, BuoyData buoyData) {
        super(point, buoyData.title, buoyData.dateString);
        this.point = point;
        this.buoyData = buoyData;
    }
}
```

You extend OverlayItem to include all the necessary properties of an item to draw on the map. In the constructor you call the superclass constructor with the location, title, and a brief snippet, and you assign additional elements your subclass instance variables. In this case you add a BuoyData member, which is another bean with name, water temperature, wave height, and other properties.

After you prepare the individual item class, you need a class that extends `ItemizedOverlay` and uses a `Collection` of the items to display them on the map one by one. The following listing, the `BuoyItemizedOverlay` class, shows how this works.

Listing 11.9 The `BuoyItemizedOverlay` class

```
public class BuoyItemizedOverlay
  extends ItemizedOverlay<BuoyOverlayItem> {
    private final List<BuoyOverlayItem> items;
    private final Context context;
    public BuoyItemizedOverlay(List<BuoyOverlayItem> items,
      Drawable defaultMarker, Context context) {
        super(defaultMarker);
        this.items = items;
        this.context = context;
        this.populate();
    }
    @Override
    public BuoyOverlayItem createItem(int i) {
        return items.get(i);
    }
    @Override
    protected boolean onTap(int i) {
        final BuoyData bd = items.get(i).buoyData;
        LayoutInflater inflater = LayoutInflater.from(context);
        View bView = inflater.inflate(R.layout.buoy_selected, null);
        TextView title = (TextView) bView.findViewById(R.id.buoy_title);
    . . . rest of view inflation omitted for brevity
        new AlertDialog.Builder(context)
            .setView(bView)
            .setPositiveButton("More Detail",
               new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface di, int what) {
                    Intent intent =
                      new Intent(context, BuoyDetailActivity.class);
                    BuoyDetailActivity.buoyData = bd;
                    context.startActivity(intent);
                }
            })
            .setNegativeButton("Cancel",
               new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface di, int what) {
                    di.dismiss();
                }
            })
            .show();
        return true;
    }
    @Override
    public int size() {
        return items.size();
    }
    @Override
    public void draw(Canvas canvas, MapView mapView, boolean b) {
```

1 Extend ItemizedOverlay

2 Include Collection of OverlayItem

3 Provide drawable marker

4 Override createItem

5 Get data to display

6 Override size method

```
        super.draw(canvas, mapView, false);
    }
}
```

Customized drawing ❼

The BuoyItemizedOverlay class extends ItemizedOverlay ❶ and includes a Collection of BuoyOverlayItem elements ❷. In the constructor you pass the Drawable marker to the parent class ❸. This marker draws on the screen in the overlay to represent each point on the map.

ItemizedOverlay takes care of many of the details you'd otherwise have to implement yourself if you made your own Overlay with multiple points drawn on it. This includes drawing items, handling focus, and processing basic events. An ItemizedOverlay will invoke the onCreate method ❹ for every element in the Collection of items it holds. ItemizedOverlay also supports facilities such as onTap ❺, where you can react when the user selects a particular overlay item. In this code you inflate some views and display an AlertDialog with information about the respective buoy when a BuoyOverlayItem is tapped. From the alert, the user can navigate to more detailed information if desired.

The size method tells ItemizedOverlay how many elements it needs to process ❻, and even though you aren't doing anything special with it in this case, there are also methods such as onDraw ❼ that you can customize to draw something beyond the standard pushpin.

When working with a MapView, you create the Overlay instances you need, then add them on top of the map. Wind and Waves uses a separate Thread to retrieve the buoy data in the MapViewActivity. You can view the data-retrieval code in the code download for this chapter. After downloading the buoy data, you send a Message to a Handler that adds the BuoyItemizedOverlay to the MapView. The following listing shows these details.

Listing 11.10 The Handler Wind and Waves uses to add overlays to the MapView

```
private final Handler handler = new Handler() {
    public void handleMessage(final Message msg) {
        progressDialog.dismiss();
        if (mapView.getOverlays().contains(buoyOverlay)) {
            mapView.getOverlays().remove(buoyOverlay);
        }
        buoyOverlay = new BuoyItemizedOverlay(buoys,
          defaultMarker,
        MapViewActivity.this);
        mapView.getOverlays().add(buoyOverlay);
    }
};
```

A MapView contains a Collection of Overlay elements. You use the remove method to clean up any existing BuoyOverlayItem class before you create and add a new one. This way you reset the data instead of adding more items on top of each other.

The built-in Overlay subclasses perfectly handle your requirements. The ItemizedOverlay and OverlayItem classes have allowed you to complete the Wind and Waves application without having to make your own Overlay subclasses directly. If you need to, Android lets you go to that level and implement your own draw, tap, touch, and other methods within your custom Overlay.

With this sample application now complete and providing you with buoy data using a MapActivity and MapView, we need to address one final maps-related concept that you haven't yet encountered—geocoding.

11.4 Converting places and addresses with Geocoder

The Android documentation describes *geocoding* as converting a "street address or other description of a location" into latitude and longitude coordinates. *Reverse geocoding* is the opposite—converting latitude and longitude into an address. To accomplish this, the Geocoder class makes a network call to a web service.

You won't use geocoding in Wind and Waves because the ocean doesn't contain cities, addresses, and so on. Nevertheless, geocoding provides invaluable tools when working with coordinates and maps. To demonstrate the concepts surrounding geocoding, this listing includes a new single Activity application, GeocoderExample.

Listing 11.11 A Geocoder example

```
. . . Class declaration and Instance variables omitted for brevity
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    input = (EditText) findViewById(R.id.input);
    output = (TextView) findViewById(R.id.output);
    button = (Button) findViewById(R.id.geocode_button);
    isAddress = (CheckBox)
      findViewById(R.id.checkbox_address);
    button.setOnClickListener(new OnClickListener() {
        public void onClick(final View v) {
            output.setText(performGeocode(
                       input.getText().toString(),
                       isAddress.isChecked()));
        }
    });
}
private String performGeocode(String in, boolean isAddr) {
        String result = "Unable to Geocode - " + in;
        if (input != null) {
        Geocoder geocoder = new Geocoder(this);
        if (isAddr) {
            try {
                List<Address> addresses =
                  geocoder.getFromLocationName(in, 1);
                if (addresses != null) {
                   result = addresses.get(0).toString();
                }
```

1 Instantiate Geocoder with Context

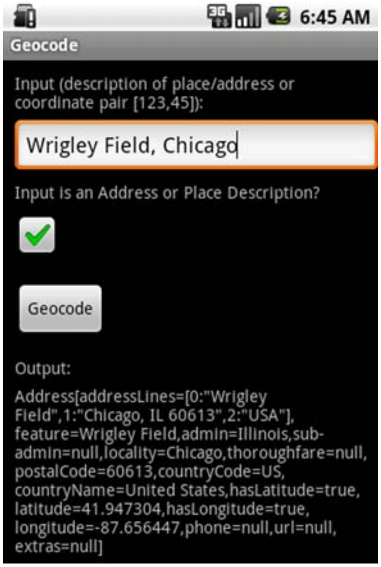
2 Get Address from location name

```
            } catch (IOException e) {
                Log.e("GeocodExample", "Error", e);
            }
        } else {
            try {
                String[] coords = in.split(",");
                if ((coords != null) && (coords.length == 2)) {
                    List<Address> addresses =
                      geocoder.getFromLocation(
                        Double.parseDouble(coords[0]),
                        Double.parseDouble(coords[1]),
                        1);
                    result = addresses.get(0).toString();
                }
            } catch (IOException e) {
                Log.e("GeocodExample", "Error", e);
            }
        }
    }
    return result;
}
```

❸ **Get Address from coordinates**

You create a `Geocoder` by constructing it with the `Context` of your application ❶. You then use a `Geocoder` to either convert `String` instances that represent place names into `Address` objects with the `getFromLocationName` method ❷ or convert latitude and longitude coordinates into `Address` objects with the `getFromLocation` method ❸.

Figure 11.8 shows our GeocoderExample in use. In this case we've converted a `String` describing Wrigley Field in Chicago into an `Address` object containing latitude and longitude coordinates.



Geocode

Input (description of place/address or coordinate pair [123,45]):

Wrigley Field, Chicago

Input is an Address or Place Description?

☑

Output:

Address[addressLines=[0:"Wrigley Field",1:"Chicago, IL 60613",2:"USA"], feature=Wrigley Field,admin=Illinois,sub-admin=null,locality=Chicago,thoroughfare=null, postalCode=60613,countryCode=US, countryName=United States,hasLatitude=true, latitude=41.947304,hasLongitude=true, longitude=-87.656447,phone=null,url=null, extras=null]

Figure 11.8 Geocoder example turning a `String` into an `Address` object that provides latitude and longitude coordinates

`Geocoder` provides many useful features. For instance, if you have data that includes address string portions, or only place descriptions, you can easily convert them into latitude and longitude numbers for use with `GeoPoint` and `Overlay` to place them on the user's map.

> **CAUTION** As of this writing, the AVD for API level 8 (the OS 2.2 emulator) doesn't properly support the geocoder. Attempts to look up an address will result in a "Service not Available" exception. Geocoding does work properly on OS 2.2 devices. To work around this problem during development, you can use API level 7 for building and testing your app on the emulator.

Geocoding concludes our look at the powerful location- and mapping-related components of the Android platform.

11.5 Summary

"Location, location, location," as they say in real estate, could also be the mantra for the future of mobile computing. Android supports readily available location information and includes smart-mapping APIs and other location-related utilities.

In this chapter we explored the location and mapping capabilities of the Android platform. You built an application that acquired a `LocationManager` and `LocationProvider`, to which you attached several `LocationListener` instances. You did this so that you could keep your application informed about the current device location by using updates delivered to your listeners. Along with the `LocationListener`, we also briefly discussed several other ways to get location updates from the Android platform.

After we covered location-awareness basics, we showed you how to add information from a unique data source, the National Data Buoy Center, to provide a draggable, zoomable, interactive map. To build the map you used a `MapActivity`, along with `MapView` and `MapController`. These classes make it fairly easy to set up and display maps. Once you had your `MapView` in place, you created an `ItemizedOverlay` to include points of interest, using individual `OverlayItem` elements. From the individual points, in this case buoys, you linked into another `Activity` class to display more detailed information, thereby demonstrating how to go from the map to any other kind of `Activity` and back.

Our water-based sample application didn't include the important mapping feature of converting from an address into a latitude and longitude and vice versa. To demonstrate this capability, we showed you how to build a separate small sample and discussed usage of the `Geocoder` class.

With our exploration of the mapping capabilities of Android complete, including a fully functional sample application that combines mapping with many other Android tenets we've previously explored, we'll move into a new stage of the book. In the next few chapters, we'll explore complete nontrivial applications that bring together intents, activities, data storage, networking, and more.

Part 3

Android applications

As you learned in part 2, the Android platform is capable of enabling rich applications in many genres and vertical industries. The goal of part 3 is to integrate many of the lessons of part 2 on a larger scale and spur you on to explore the platform in greater depth than simply using the Android SDK.

In chapter 12, we take a detailed look at the requirements of a field service application. We next map those requirements on a practical application that could be adapted for many industries. The application includes multiple UI elements, server communications, and detecting touch-screen events for capturing and uploading a signature.

In chapter 13, we move on to a deeper examination of the Android/Linux relationship by writing native C applications for Android and connecting to Android core libraries such as SQLite and TCP socket communications.

12 Putting Android to work in a field service application

This chapter covers

- Designing a real-world Android application
- Mapping out the application flow
- Writing application source code
- Downloading, data parsing, and signature capture
- Uploading data to a server

Now that we've introduced and examined Android and some of its core technologies, it's time to put together a more comprehensive application. Exercising APIs can be informative, educational, and even fun for a while, but at some point a platform must demonstrate its worth via an application that can be used outside of the ivory tower—and that's what this chapter is all about. In this chapter, we systematically design, code, and test an Android application to aid a team of field service technicians in performing their job. The application syncs XML data with an internet-hosted server, presents data to the user via intuitive user interfaces, links to Google Maps, and concludes by collecting customer signatures via Android's touch screen. Many of the APIs introduced earlier are exercised here, demonstrating the power and versatility of the Android platform.

In addition to an in-depth Android application, this chapter's sample application works with a custom website application that manages data for use by a mobile worker. This server-side code is presented briefly toward the end of the chapter. All of the source code for the server-side application is available for download from the book's companion website.

If this example is going to represent a useful real-world application, we need to put some flesh on it. Beyond helping you understand the application, this definition process will get you thinking about the kinds of impact a mobile application can have on our economy. This chapter's sample application is called a *field service application*. A pretty generic name perhaps, but it'll prove to be an ample vehicle for demonstrating key elements required in mobile applications, as well as 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 that makes its services available to a number of contracted customers. One day our technician, who we'll call a *mobile worker*, is replacing a hard drive in the computer at the local fast-food restaurant, and the next day he may be installing a memory upgrade in a piece of pick-and-place machinery at a telephone system manufacturer. If you've ever had a piece of equipment serviced at your home or office and thought the technician's uniform didn't really match the job he was doing, you've experienced this kind of service arrangement. This kind of technician is often referred to as *hands and feet*. He has basic mechanical or computer skills and is able to follow directions reliably, often guided by the manufacturer of the equipment being serviced at the time. Thanks to workers like these, companies can extend their reach to a much broader geography than internal staffing levels would ever allow. For example, a small manufacturer of retail music-sampling equipment might contract with such a firm to provide tech support to retail locations across the country.

Because of our hypothetical technician's varied schedule and lack of experience on a particular piece of equipment, it's important to equip him with as much relevant and timely information as possible. But he can't be burdened with thick reference manuals or specialized tools. So, with a toolbox containing a few hand tools and of course an Android-equipped device, our fearless hero is counting on us to provide an application that enables him to do his job. And remember, this is the person who restores the ice cream machine to operation at the local Dairy Barn, or perhaps fixes the farm equipment's computer controller so the cows get milked on time. You never know where a computer will be found in today's world!

If built well, this application can enable the efficient delivery of service to customers in many industries, where we live, work, and play. Let's get started and see what this application must be able to accomplish and how Android steps up to the task.

12.1 Designing a real-world Android application

We've established that our mobile worker will be carrying two things: a set of hand tools and an Android device. Fortunately, in this book we're not concerned with the applicability of the hand tools in his toolbox, leaving us free to focus on the capabilities and features of a field service application running on the Android platform. In this section, we define the basic and high-level application requirements.

12.1.1 Core requirements of the application

Before diving into the bits and bytes of data requirements and application features, it's helpful to enumerate some basic requirements and assumptions about our field service application. Here are a few items that come to mind for such an application:

- The mobile worker is dispatched by a home office/dispatching authority, which takes care of prioritizing and distributing job orders to the appropriate technician.
- The mobile worker is carrying an Android device, which has full data service—a device capable of browsing rich web content. The application needs to access the internet for data transfer as well.
- The home office dispatch system and the mobile worker share data via a wireless internet connection on an Android device; a laptop computer isn't necessary or even desired.
- A business requirement is the proof of completion of work, most readily accomplished with the capture of a customer's signature. Of course, an electronic signature is preferred.
- The home office wants to receive job completion information as soon as possible, as this accelerates the invoicing process, which improves cash flow.
- The mobile worker is also eager to perform as many jobs as possible, because he's paid by the job, not by the hour, so getting access to new job information as quickly as possible is a benefit to the mobile worker.
- The mobile worker needs information resources in the field and can use as much information as possible about the problem he's being asked to resolve. The mobile worker may have to place orders for replacement parts while in the field.
- The mobile worker will require navigation assistance, as he's likely covering a rather large geographic area.
- The mobile worker needs an intuitive application—one that's simple to use with a minimum number of requirements.

There are likely additional requirements for such an application, but this list is adequate for our purposes. One of the most glaring omissions from our list is security.

Security in this kind of an application comes down to two fundamental aspects. The first is physical security of the Android device. Our assumption is that the device itself is locked and only the authorized worker is using it. A bit naïve perhaps, but there are more important topics we need to cover in this chapter. If this bothers you, just assume there's a sign-in screen with a password field that pops up at the most inconvenient times, forcing you to tap in your password on a small keypad. Feel better now? The second security topic is the secure transmission of data between the

Android device and the dispatcher. This is most readily accomplished through the use of a *Secure Sockets Layer (SSL)* connection whenever required.

The next step in defining this application is to examine the data flows and discuss the kind of information that must be captured to satisfy the functional requirements.

12.1.2 Managing the data

Throughout this chapter, the term *job* refers to a specific task or event that our mobile worker engages in. For example, a request to replace a hard drive in a computer at the bookstore is a job. A request to upgrade the firmware in the fuel-injection system at the refinery is likewise a job. The home office dispatches one or more jobs to the mobile worker on a regular basis. Certain data elements in the job are helpful to the mobile worker to accomplish his goal of completing the job. This information comes from the home office. Where the home office gets this information isn't our concern for this application.

In this chapter's sample application, there are only two pieces of information the mobile worker is responsible for submitting to the dispatcher:

- The mobile worker communicates to the home office that a job has been *closed*, or completed.
- The collection of an electronic signature from the customer, acknowledging that the job has, in fact, been completed.

Figure 12.1 depicts these data flows.

Of course, additional pieces of information exist that may be helpful here, such as the customer's 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. Although important to a real-world application, these pieces of information are extraneous to the goals of this chapter and are left as an exercise for you to extend the application for your own learning and use.

The next objective is to determine how data is stored and transported.

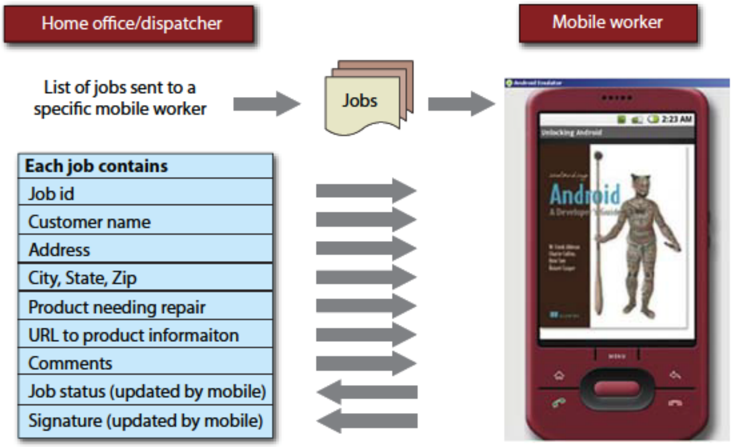


Figure 12.1 **Data flows between the home office and a mobile worker**

12.1.3 Application architecture and integration

Now that you know which entities are responsible for the relevant data elements, and in which direction they flow, let's look at how the data is stored and exchanged. You'll be deep into code before too long, but for now we'll focus on the available options and continue to examine things from a requirements perspective, building to a proposed architecture.

At the home office, the dispatcher must manage data for multiple mobile workers. The best tool for this purpose is a relational database. The options here are numerous, but we'll make the simple decision to use MySQL, a popular open source database. Not only are there multiple mobile workers, but the organization we're building this application for is quite spread out, with employees in multiple markets and time zones. Because of the nature of the dispatching team, it's been decided to host the MySQL database in a data center, where it's accessed via a browser-based application. For this sample application, the dispatcher system is supersimple and written in PHP.[1]

Data storage requirements on the mobile device are modest. At any point, a given mobile worker may have only a half-dozen or so assigned jobs. Jobs may be assigned at any time, so the mobile worker is encouraged to refresh the list of jobs periodically. Although you learned about how to use SQLite in chapter 5, we have little need for sharing data between multiple applications and don't need to build a `ContentProvider`, so we've decided to use an XML file stored on the filesystem to serve as a persistent store of our assigned job list.

The field service application uses HTTP to exchange data with the home office. Again, we use PHP to build the transactions for exchanging data. Though more complex and sophisticated protocols can be employed, such as SOAP, this application simply requests an XML file of assigned jobs and submits an image file representing the captured signature. In fact, SOAP is simple in name only and should be avoided. A better solution that's coming on strong in the mobile and web space is the JSON format. This architecture is depicted in figure 12.2.

The last item to discuss before diving into the code is configuration. Every mobile worker needs to be identified uniquely. This way, the field service application can retrieve the correct job list, and the dispatchers can assign jobs to workers in the field. Similarly, the mobile application may need to communicate with different servers, depending on locale. A mobile worker in the United States might use a server located in Chicago, but a worker in the United Kingdom may need to use a server in Cambridge. Because of these requirements, we've decided that both the mobile worker's identifier and the server address need to be readily accessed within the application. Remember, these fields would likely be secured in a deployed application, but for our purposes they're easy to access and not secured.

[1] See Manning's *PHP in Action* for details on PHP development: http://www.manning.com/reiersol/.

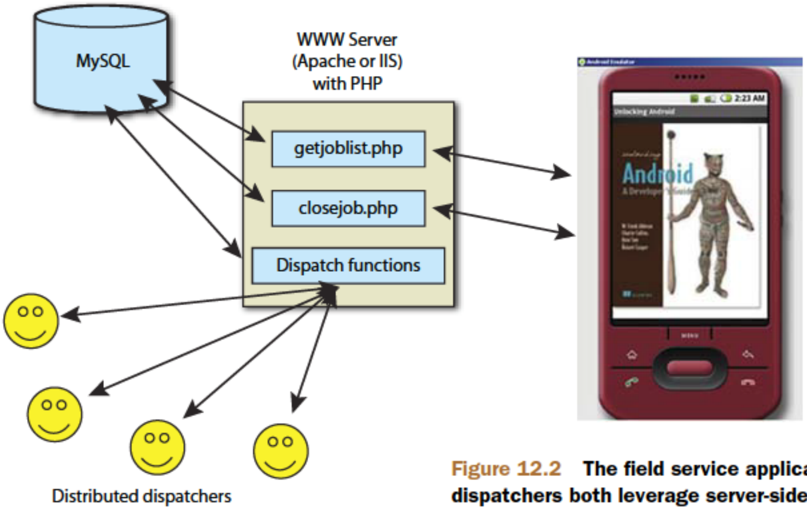


Figure 12.2 **The field service application and dispatchers both leverage server-side transactions.**

We've identified the functional requirements, defined the data elements necessary to satisfy those objectives, and selected the preferred deployment platform. The next section examines the high-level solution to the application's requirements.

12.2 Mapping out the application flow

Have you ever downloaded an application's source code, excited to get access to all that code, but found it overwhelming to navigate? You want to make your own changes, to put your own spin on the code, but you unzip the file into all the various subdirectories, and you don't know where to start. Before we jump directly into examining the source code, we need to pay attention to the architecture, in particular the flow from one screen to the next.

12.2.1 Mapping out the field service application

In this section we'll examine the application flow to better understand the relation among the application's functionality, the UI, and the classes used to deliver this functionality. Doing this process up-front helps ensure that the application delivers the needed functionality and assists in defining which classes we require when it comes time to start coding (which is soon)! Figure 12.3 shows the relation between the high-level classes in the application, which are implemented as an Android `Activity`, as well as interaction with other services available in Android.

Here's the procession of steps in the application:

1. The application is selected from the application launch screen on the Android device.
2. The application splash screen displays. Why? Some applications require setup time to get data structures initialized. As a practical matter, such time-consuming behavior is discouraged on a mobile device, but it's an important aspect to application design, so we include it in this sample application.

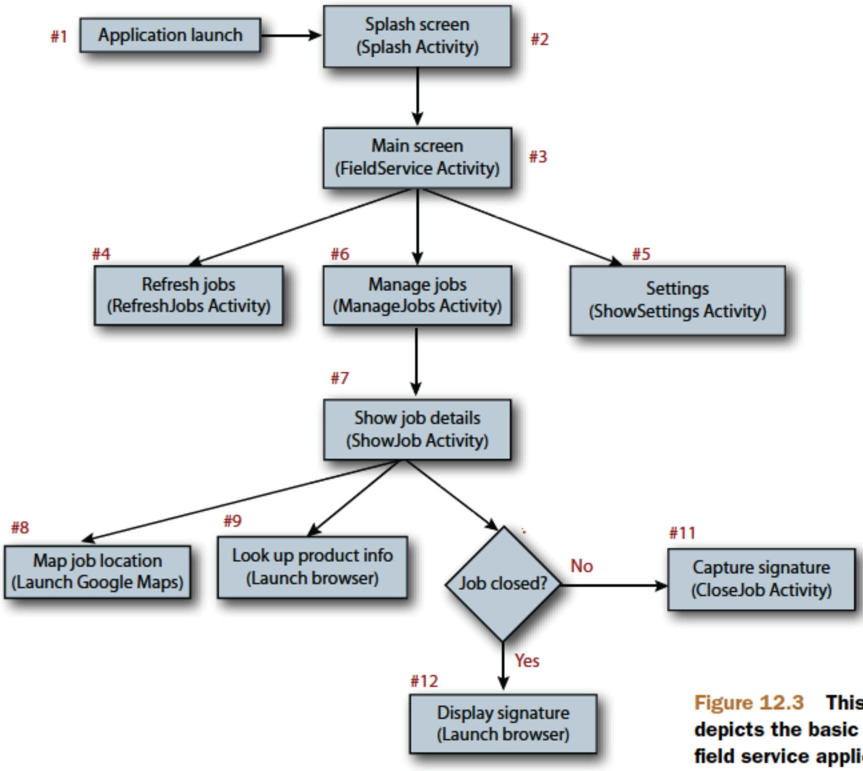

#1
Application launch
Splash screen (Splash Activity)
#2
Main screen (FieldService Activity)
#3
#4
Refresh jobs (RefreshJobs Activity)
#6
Manage jobs (ManageJobs Activity)
#5
Settings (ShowSettings Activity)
#7
Show job details (ShowJob Activity)
#8
Map job location (Launch Google Maps)
#9
Look up product info (Launch browser)
Job closed?
No
#11
Capture signature (CloseJob Activity)
Yes
#12
Display signature (Launch browser)

Figure 12.3 This figure depicts the basic flow of the field service application.

3 The main screen displays the currently configured user and server settings, along with three easy-to-hit-with-your-finger buttons.

4 The Refresh Jobs button initiates a download procedure to fetch the currently available jobs for this mobile worker from the configured server. The download includes a `ProgressDialog`, which we discuss in section 12.3.5.

5 The Settings button brings up a screen that allows you to configure the user and server settings.

6 Selecting Manage Jobs lets our mobile worker review the available jobs assigned to him and proceed with further steps specific to a chosen job.

7 Selecting a job from the list of jobs on the Manage Jobs screen brings up the Show Job Details screen with the specific job information listed. This screen lists the available information about the job and presents three additional buttons.

8 The Map Job Location button initiates a geo query on the device using an `Intent`. The default handler for this `Intent` is the Maps application.

9 Because our mobile worker may not know much about the product he's being asked to service, each job includes a product information URL. Clicking this button brings up the built-in browser and takes the mobile worker to a (hopefully) helpful internet resource. This resource may be an online manual or an instructional video.

10 The behavior of the third button depends on the current status of the job. If the job is still marked OPEN, this button is used to initiate the closeout or completion of this job.

11 When the close procedure is selected, the application presents an empty canvas upon which the customer can take the stylus (assuming a touch screen–capable Android device) and sign that the work is complete. A menu on that screen presents two options: Sign & Close or Cancel. If the user selects Sign & Close option, the application submits the signature as a JPEG image to the server, and the server marks the job as CLOSED. In addition, the local copy of the job is marked as CLOSED. The Cancel button causes the Show Job Details screen to be restored.

12 If the job being viewed has already been closed, the browser window is opened to a page displaying the previously captured signature.

Now that you have a feel for what the requirements are and how you're going to tackle the problem from a functionality and application-flow perspective, let's examine the code that delivers this functionality.

12.2.2 List of source files

The source code for this application consists of 12 Java source files, one of which is the R.java file, which you'll recall is automatically generated based on the resources in the application. This section presents a quick introduction to each of these files. We won't explain any code yet; we want you to know a bit about each file, and then it'll be time to jump into the application, step by step. Table 12.1 lists the source files in the Android field service application.

Table 12.1 Source files used to implement the field service application

Source filename	Description
Splash.java	`Activity` provides splash screen functionality.
ShowSettings.java	`Activity` provides management of username 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 initiates the signature-capture 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 that represents a job. Includes helpful methods used when passing `JobEntry` objects from one `Activity` to another.

Table 12.1 Source files used to implement the field service application *(continued)*

Source filename	Description
JobList.java	Class representing the complete list of `JobEntry` objects. Includes methods for marshaling and unmarshaling to nonvolatile storage.
JobListHandler.java	Class used for parsing XML document containing job data.

The application also relies on `layout` resources to define the visual aspect of the UI. In addition to the `layout` XML files, an image used by the `Splash Activity` is placed in the drawable subfolder of the res folder along with the stock Android icon image. This icon is used for the home application launch screen. Figure 12.4 depicts the resources used in the application.

In an effort to make navigating the code as easy as possible, table 2.2 shows the field service application resource files. Note that each is clearly seen in figure 12.4, which is a screenshot from our project open in Eclipse.

Examining the source files in this application tells us that we have more than one `Activity` in use. To enable navigation between one `Activity` and the next, our application must inform Android of the existence of these `Activity` classes. Recall from chapter 1 that this registration step is accomplished with the AndroidManifest.xml file.

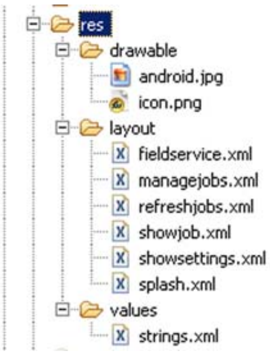


Figure 12.4 Resource files used in the sample application

Table 12.2 Resource files used in the sample application

Filename	Description
android.jpg	Image used in the `Splash Activity`.
icon.jpg	Image used in the application launcher.
fieldservice.xml	Layout for 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.

12.2.3 Field service application's AndroidManifest.xml

Every Android application requires a manifest file to let Android properly "wire things up" when an `Intent` is handled and needs to be dispatched. Take a look at the AndroidManifest.xml file used by our application, shown in the following listing.

Listing 12.1 The field service application's AndroidManifest.xml file

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.msi.manning.UnlockingAndroid">
    <application android:icon="@drawable/icon">
        <activity android:name=".Splash"
                android:label="@string/app_name">
          <intent-filter>
             <action android:name="android.intent.action.MAIN" />
             <category android:name="android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
        <activity android:name=".FieldService" >
        </activity>
        <activity android:name=".RefreshJobs" >
        </activity>
        <activity android:name=".ManageJobs" >
        </activity>
        <activity android:name=".ShowJob" >
        </activity>
        <activity android:name=".CloseJob" >
        </activity>
        <activity android:name=".ShowSettings" >
        </activity>
    </application>
<uses-sdk android:minSdkVersion="6"/>
<uses-permission android:name="android.permission.INTERNET">
    </uses-permission>
</manifest>
```

Entry point, Splash Activity

Intent filter for main launcher visibility

Application's defined Activity list

Required permission for internet access

12.3 Application source code

After a rather long introduction, it's time to look at the source code for the field service application. The approach is to follow the application flow, step by step. Let's start with the splash screen. In this portion of the chapter, we work through each of the application's source files, starting with the splash screen and moving on to each subsequent `Activity` of the application.

12.3.1 Splash Activity

We're all familiar with a splash screen for a software application. It acts like a curtain while important things are taking place behind the scenes. Ordinarily splash screens are visible until the application is ready—this could be a brief amount of time or much longer when a bit of housekeeping is necessary. As a rule, a mobile application should focus on economy and strive to consume as few resources as possible. The splash screen in this sample application is meant to demonstrate how such a feature may be constructed—we don't need one for housekeeping purposes. But that's okay; you can learn in the process. Two code snippets are of interest to us: the implementation of the `Activity` as well as the layout file that defines what the UI looks like. First, examine the layout file in the following listing.

Listing 12.2 splash.xml, defining the layout of the application's splash screen

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

❶ Full screen ImageView

❷ Image reference

The splash.xml layout contains a single `ImageView` ❶, set to fill the entire screen. The source image for this view is defined as the `drawable` resource ❷, named `android`. Note that this is simply the name of the file (minus the file extension) in the drawable folder, as shown earlier.

Now you must use this layout in an `Activity`. Aside from the referencing of an image resource from the layout, this part is not that interesting. Figure 12.5 shows the splash screen running on the Android Emulator.

Of interest to us is the code that creates the splash page functionality, shown in the following listing.

Listing 12.3 Splash.java, which implements the splash screen functionality

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source code
public class Splash extends Activity  {
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.splash);
        Handler x = new Handler();
        x.postDelayed(new SplashHandler(), 2000);
   }
   class SplashHandler implements Runnable {
  public void run() {
        startActivity(
          new Intent(getApplication(),FieldService.class));
        Splash.this.finish();
       }
  }
}
```

❶ Set up main View

❷ Define and set up Handler

❸

❹ Kill splash screen

As with most `Activity` classes in Android, you want to associate the splash layout with this `Activity`'s `View` ❶. A `Handler` is set up ❷, which is used to close down the splash screen after an elapsed period of time. Note that the arguments to the `postDelayed` method are an instance of a class that implements the `Runnable` interface and the desired elapsed time in milliseconds. In this snippet of code, the screen will be shown for 2,000 milliseconds, or 2 seconds. After the indicated amount of time has elapsed, the class `splashhandler` is invoked. The `FieldService Activity` is instantiated with a call to `startActivity` ❸. Note that an `Intent` isn't used here—you explicitly specify which class is going to satisfy your request. Once you've started the next `Activity`, it's time to get rid of your splash screen `Activity` ❹.

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.

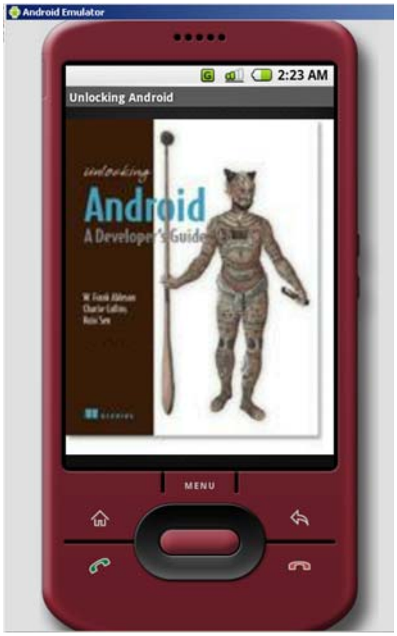


Figure 12.5 **The splash screen**

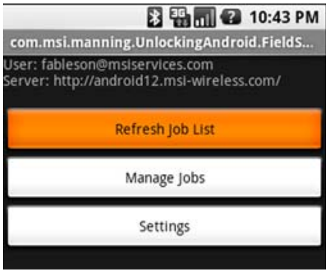
12.3.2 *Preferences used by the FieldService Activity*

The goal of the `FieldService Activity` is to put the functions the mobile worker requires directly in front of him and make sure they're easy to access. A good mobile application is often one that can be used with one hand, such as using the five-way navigation buttons, or in some cases a thumb tapping on a button. In addition, if there's helpful information to display, you shouldn't hide it. It's helpful for our mobile worker to know that he's configured to obtain jobs from a particular server. Figure 12.6 demonstrates the field service application conveying an easy-to-use home screen.

Before reviewing the code in FieldService.java, let's take a break to discuss how the user and server settings are managed. This is important because these settings are used throughout the application, and as shown in the fieldservice.xml layout file, we need to access those values to display to our mobile worker on the home screen.

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 nonvolatile fashion. This



User: fableson@msiservices.com
Server: http://android12.msi-wireless.com/

Figure 12.6 **The home screen. Less is more.**

application employs private SharedPreferences to accomplish this. Why? Despite the fact that we're largely ignoring security for this sample application, using private SharedPreferences means that other applications can't casually access this potentially important data. For example, we presently use only an identifier (let's call it an email address for simplicity) and a server URL in this application. But we might also include a password or a PIN in a production-ready application, so keeping this data private is a good practice.

The Prefs class can be described as a helper or wrapper class. This class wraps the SharedPreferences code and exposes simple getter and setter methods, specific to this application. This implementation knows something about what we're trying to accomplish, so it adds value with some default values as well. Let's look at the following listing to see how our Prefs class is implemented.

Listing 12.4 Prefs class

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source code
public class Prefs {
     private SharedPreferences _prefs = null;
     private Editor _editor = null;
     private String _useremailaddress = "Unknown";
     private String _serverurl =
         "http://android12.msi-wireless.com/getjoblist.php";
     public Prefs(Context context) {
       _prefs = context.getSharedPreferences(
"PREFS_PRIVATE",
Context.MODE_PRIVATE);
       _editor = _prefs.edit();
     }
     public String getValue(String key,String defaultvalue){
        if (_prefs == null) return "Unknown";
        return _prefs.getString(key,defaultvalue);
     }
     public void setValue(String key,String value) {
        if (_editor == null) return;
       _editor.putString(key,value);
     }
     public String getEmail(){
        if (_prefs == null) return "Unknown";
         _useremailaddress = _prefs.getString("emailaddress","Unknown");
         return _useremailaddress;
     }
     public void setEmail(String newemail)  {
        if (_editor == null) return;
        _editor.putString("emailaddress",newemail);
     }
 ... (abbreviated for brevity)
     public void save() {
       if (_editor == null) return;
      _editor.commit();
     }
}
```

1 SharedPreferences object

2 Implement Handler

Default values

4 Initialize SharedPreferences

5 Generic set/get methods

6 Extract email value

7 Set email value

8 Save preferences

To persist the application's settings data, you employ a SharedPreferences object ❶. To manipulate data within the SharedPreferences object, here named _prefs, you use an instance of the Editor class ❷. This snippet employs some default settings values ❸, which are appropriate for your application. The Prefs() constructor ❹ does the necessary housekeeping so you can establish your private SharedPreferences object, including using a passed-in Context instance. The Context class is necessary because the SharedPreferences mechanism relies on a Context for segregating data. This snippet shows a pair of set and get methods that are generic in nature ❺. The getEmail ❻ and setEmail methods ❼ are responsible for manipulating the email setting value. The save() method ❽ invokes a commit() on the Editor, which persists the data to the SharedPreferences store.

Now that you have a feel for how this important preference data is stored, let's return to examine the code of FieldService.java.

12.3.3 Implementing the FieldService Activity

Recall that the FieldService.java file implements the FieldService class, which is essentially the home screen of our application. This code does the primary dispatching for the application. Many of the programming techniques in this file have been shown earlier in the book, but please note the use of startActivityForResult and the onActivityResult methods as you read through the code shown in the following listing.

Listing 12.5 FieldService.java, which implements the FieldService Activity

```
package com.msi.manning.UnlockingAndroid;
// multiple imports trimmed for brevity, see full source code
public class FieldService extends Activity {
    final int ACTIVITY_REFRESHJOBS = 1;
    final int ACTIVITY_LISTJOBS = 2                      Useful constants
    final int ACTIVITY_SETTINGS = 3;
    Prefs myprefs = null;                                ❶ Prefs instance
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);                          Set up UI   ❷ Instantiate
        setContentView(R.layout.fieldservice);                         Prefs
        myprefs = new Prefs(this.getApplicationContext());             instance
        RefreshUserInfo();                               Initiate UI field contents
        final Button refreshjobsbutton =
              (Button) findViewById(R.id.getjobs);
        refreshjobsbutton.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
              try {                                               Connect
                 startActivityForResult(new                  button to UI ❸
 Intent(v.getContext(),RefreshJobs.class),ACTIVITY_REFRESHJOBS);
            } catch (Exception e) {
            }
            }
        });
        // see full source comments
    }
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent
     data) {
      switch (requestCode) {
          case ACTIVITY_REFRESHJOBS:
            break;
          case ACTIVITY_LISTJOBS:
            break;
          case ACTIVITY_SETTINGS:
                    RefreshUserInfo();
                    break;
        }
    }
    private void RefreshUserInfo() {
      try {
            final TextView emaillabel = (TextView)
     findViewById(R.id.emailaddresslabel);
            emaillabel.setText("User: " + myprefs.getEmail() + "\nServer: " +
     myprefs.getServer() + "\n");
     } catch (Exception e) {
    }
  }
}
```

4 onActivityResult processing

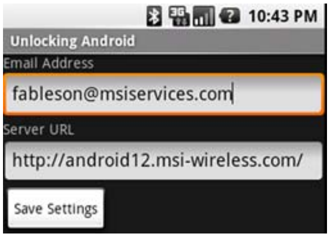
5 RefreshUserInfo

This code implements a simple UI that displays three distinct buttons. As each is selected, a particular `Activity` is started in a synchronous, call/return fashion. The `Activity` is started with a call to `startActivityForResult` 3. When the called `Activity` is complete, the results are returned to the `FieldService Activity` via the `onActivityResult` method 4. An instance of the `Prefs` class 1, 2 is used to obtain values for displaying in the UI. Updating the UI is accomplished in the method `RefreshUserInfo` 5.

Because the settings are so important to this application, the next section covers the management of the user and server values.

12.3.4 Settings

When the user clicks the Settings button on the main application screen, an `Activity` is started that allows the user to configure her user ID (email address) and the server URL. The screen layout is basic (see listing 12.6). It's shown graphically in figure 12.7.



fableson@msiservices.com

http://android12.msi-wireless.com/

Figure 12.7 Settings screen in use

Listing 12.6 showsettings.xml, which contains UI elements for the settings screen

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Email Address"
    />
    <EditText
    android:id="@+id/emailaddress"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoText="true"
    />
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Server URL"
    />
    <EditText
    android:id="@+id/serverurl"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:autoText="true"
    />
    <Button android:id="@+id/settingssave"
    android:text="Save Settings"
    android:layout_height="wrap_content"
    android:layout_width="wrap_content"
    android:enabled="true"
    />
</LinearLayout>
```

TextView for display of labels

TextView for display of labels

Button to initiate saving data

The source code behind the settings screen is also basic. Note the use of the PopulateScreen() method, which makes sure the EditView controls are populated with the current values stored in the SharedPreferences. Note also the use of the Prefs helper class to retrieve and save the values, as shown in the following listing.

Listing 12.7 ShowSettings.java, which implements code behind the settings screen

```
package com.msi.manning.UnlockingAndroid;
// multiple imports trimmed for brevity, see full source code
public class ShowSettings extends Activity  {
    Prefs myprefs = null;
    @Override
    public void onCreate(Bundle icicle) {
      super.onCreate(icicle);
      setContentView(R.layout.showsettings);

      myprefs = new Prefs(this.getApplicationContext());
      PopulateScreen();
      final Button savebutton = (Button) findViewById(R.id.settingssave);
      savebutton.setOnClickListener(new Button.OnClickListener() {
      public void onClick(View v) {
        try {
         final EditText email=
```

1 Initialize Prefs instance

2 Populate UI elements

```
            (EditText)findViewById(R.id.emailaddress);
            if (email.getText().length() == 0) {
             // display dialog, see full source code
              return;                                            ❸ Connect EditText to UI
             }
            final EditText serverurl =
                 (EditText)findViewById(R.id.serverurl);
             if (serverurl.getText().length() == 0) {
               // display dialog, see full source code
               return;
            }
             myprefs.setEmail(email.getText().toString());
             myprefs.setServer(serverurl.getText().toString());  ❹ Store and save settings
             myprefs.save();
             finish();                                           ❺ Finish this Activity
            } catch (Exception e) {
            }
          }
        });
}
    private void PopulateScreen()  {                             ❻ PopulateScreen method sets up UI
      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) {
    }
   }
}
```

This Activity commences by initializing the SharedPreferences instance ❶, which retrieves the setting's values and subsequently populates the UI elements ❷ by calling the application-defined PopulateScreen method ❻. When the user clicks the Save Settings button, the onClick method is invoked, wherein the data is extracted from the UI elements ❸ and put back into the Prefs instance ❹. A call to the finish method ❺ ends this Activity.

Once the settings are in order, it's time to focus on the core of the application: managing jobs for our mobile worker. To get the most out the higher-level functionality of downloading (refreshing) and managing jobs, let's examine the core data structures in use in this application.

12.3.5 Managing job data

Data structures represent a key element of any software project and, in particular, projects consisting of multiple tiers, such as our field service application. Job data is exchanged between an Android application and the server, so the elements of the job are central to our application. In Java, you implement these data structures as classes, which include helpful methods in addition to the data elements. XML data shows up in many locations in this application, so let's start there.

The following listing shows a sample XML document containing a joblist with a single job entry.

Listing 12.8 XML document containing data for the field service application

```
<?xml version="1.0" encoding="UTF-8" ?>
<joblist>
<job>
<id>22</id>
<status>OPEN</status>
<customer>Big Tristan's Imports</customer>
<address>2200 East Cedar Ave</address>
<city>Flagstaff</city>
<state>AZ</state>
<zip>86004</zip>
<product>UnwiredTools UTCIS-PT</product>
<producturl>http://unwiredtools.com</producturl>
<comments>Requires tuning - too rich in the mid range RPM.
Download software from website before visiting.</comments>
</job>
</joblist>
```

Now that you have a feel for what the job data looks like, we'll show you how the data is handled in our Java classes.

JOBENTRY

The individual job is used throughout the application, and therefore it's essential that you understand it. In our application, you define the JobEntry class to manage the individual job, as shown in listing 12.9. Note that many of the lines are omitted from this listing for brevity; please see the available source code for the complete code listing.

Listing 12.9 JobEntry.java

```
package com.msi.manning.UnlockingAndroid;
import android.os.Bundle;
public class JobEntry {
 private String _jobid="";
 private String _status = "";
  // members omitted for brevity
private String _producturl = "";
 private String _comments = "";
 JobEntry() {
 }
 // get/set methods omitted for brevity
 public String toString() {
  return this._jobid + ": " + this._customer + ": " + this._product;
 }
 public String toXMLString() {
   StringBuilder sb = new StringBuilder("");
   sb.append("<job>");
   sb.append("<id>" + this._jobid + "</id>");
   sb.append("<status>" + this._status + "</status>");
   sb.append("<customer>" + this._customer + "</customer>");
```

1 Bundle class import

2 Each member is a String

3 toString method

4 toXMLString method

```
   sb.append("<address>" + this._address + "</address>");
   sb.append("<city>" + this._city + "</city>");
   sb.append("<state>" + this._state + "</state>");
   sb.append("<zip>" + this._zip + "</zip>");
   sb.append("<product>" + this._product + "</product>");
   sb.append("<producturl>" + this._producturl + "</producturl>");
   sb.append("<comments>" + this._comments + "</comments>");
   sb.append("</job>");
   return sb.toString() + "\n";
}
 public Bundle toBundle() {
   Bundle b = new Bundle();
   b.putString("jobid", this._jobid);
   b.putString("status", this._status);
   // assignments omitted for brevity
   b.putString("producturl", this._producturl);
   b.putString("comments", this._comments);
   return b;
}
public static JobEntry fromBundle(Bundle b) {
   JobEntry je = new JobEntry();
   je.set_jobid(b.getString("jobid"));
   je.set_status(b.getString("status"));
   // assignments omitted for brevity
 je.set_producturl(b.getString("producturl"));
   je.set_comments(b.getString("comments"));
   return je;
}
}
```

❺ **toBundle method**

❻ **fromBundle method**

This application relies heavily on the `Bundle` class ❶ for moving data from one `Activity` to another. (We'll explain this in more detail later in this chapter.) A `String` member ❷ exists for each element in the job, such as `jobid` or `customer`. The `toString()` method ❸ is rather important, as it's used when displaying jobs in the `ManageJobs Activity` (also discussed later in the chapter). The `toXMLString()` method ❹ generates an XML representation of this `JobEntry`, complying with the `job element` defined in the previously presented DTD. The `toBundle()` method ❺ takes the data members of the `JobEntry` class and packages them into a `Bundle`. This `Bundle` is then able to be passed between activities, carrying with it the required data elements. The `fromBundle()` static method ❻ returns a `JobEntry` when provided with a `Bundle`. `toBundle()` and `fromBundle()` work together to assist in the passing of `JobEntry` objects (at least the data portion thereof) between activities. Note that this is one of many ways in which to move data throughout an application. Another method, as an example, is to have a globally accessible class instance to store data.

Now that you understand the `JobEntry` class, we'll move on to the `JobList` class, which is a class used to manage a collection of `JobEntry` objects.

JOBLIST

When interacting with the server or presenting the available jobs to manage on the Android device, the field service application works with an instance of the `JobList` class. This class, like the JobEntry class, has both data members and helpful methods. The JobList class contains a typed List data member, which is implemented using a Vector. This is the only data member of this class, as shown in the following listing.

Listing 12.10 JobList.java

```
package com.msi.manning.UnlockingAndroid;
import java.util.List;
import org.xml.sax.InputSource;
import android.util.Log;
// additional imports omitted for brevity, see source code
public class JobList  {
     private Context _context = null;
     private List<JobEntry> _joblist;
     JobList(Context context){
        _context = context;
        _joblist = new Vector<JobEntry>(0);
     }
     int addJob(JobEntry job){
        _joblist.add(job);
        return _joblist.size();
     }
     JobEntry getJob(int location) {
        return _joblist.get(location);
     }
     List<JobEntry> getAllJobs() {
        return _joblist;
     }
     int getJobCount() {
        return _joblist.size();
     }
     void replace(JobEntry newjob) {
        try {
          JobList newlist = new JobList();
          for (int i=0;i<getJobCount();i++) {
             JobEntry je = getJob(i);
             if (je.get_jobid().equals(newjob.get_jobid())) {
                newlist.addJob(newjob);
             } else {
               newlist.addJob(je);
             }
          }
          this._joblist = newlist._joblist;
          persist();
        } catch (Exception e) {
        }
}
void persist() {
    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++) {
```

1 List class imported for Vector

2 InputSource imported, used by XML parser

Familiar logging mechanism

3 Constructor

4 addJob/getJob methods

5 getAllJobs method

6 replace method

7 persist method

```
          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) {                ❽ parse method
     try {
       FileInputStream fis = context.openFileInput("chapter12.xml");
       if (fis == null) {
           return null;
       }
       InputSource is = new InputSource(fis);
       SAXParserFactory factory = SAXParserFactory.newInstance();
       SAXParser parser = factory.newSAXParser();
       XMLReader xmlreader = parser.getXMLReader();
       JobListHandler jlHandler =
new JobListHandler(null /* no progress updates when reading file */);
       xmlreader.setContentHandler(jlHandler);
       xmlreader.parse(is);
       fis.close();
       return jlHandler.getList();
     } catch (Exception e) {
        return null;
     }
   }
}
```

The list of jobs is implemented as a `Vector`, which is a type of `List` ❶. The XML structure containing job information is parsed with the SAX parser, so you need to be sure to import those required packages ❷. `JobEntry` objects are stored in the typed `List` object named `_joblist` ❸. Helper methods for managing the list are included as `addJob` and `getJob` ❹. The `getAllJobs()` method ❺ returns the list of `JobEntry` items. Note that generally speaking, the application uses the `getJob()` method for individual `JobEntry` management, but the `getAllJobs()` method is particularly useful when you display the full list of jobs in the `ManageJobs Activity`, discussed later in this chapter.

The `replace()` method ❻ is used when you've closed a job and need to update your local store of jobs. Note that after it has updated the local list of `JobEntry` items, `replace()` calls the `persist()` ❼ method, which is responsible for writing an XML representation of the entire list of `JobEntry` items to storage. This method invokes the `toXMLString()` method on each `JobEntry` in the list. The `openFileOutput` method creates a file within the application's private file area. This is essentially a helper method to ensure you get a file path to which you have full read/write privileges.

The `parse` method ❽ obtains an instance of a `FileInputStream` to gain access to the file and creates an instance of an `InputStream` ❷, which is required by the SAX XML parser. In particular, take note of the JobListHandler. SAX is a callback parser, meaning that it invokes a user-supplied method to process events in the parsing process. It's up to the JobListHandler (in our example) to process the data as appropriate.

We have one more class to go before we can jump back to the higher-level functionality of our application. The next section takes a quick tour of the JobListHandler, which is responsible for putting together a JobList from an XML data source.

JOBLISTHANDLER

As presented earlier, our application uses an XML data storage structure. This XML data can come from either the server or from a local file on the filesystem. In either case, the application must parse this data and transform it into a useful form. This is accomplished through the use of the SAX XML parsing engine and the JobListHandler, which is shown in listing 12.11. The JobListHandler is used by the SAX parser for our XML data, regardless of the data's source. Where the data comes from dictates how the SAX parser is set up and invoked in this application. The JobListHandler behaves slightly differently depending on whether the class's constructor includes a Handler argument. If the Handler is provided, the JobListHandler will pass messages back for use in a ProgressDialog. If the Handler argument is null, this status message passing is bypassed. When parsing data from the server, the ProgressDialog is employed; the parsing of a local file is done quickly and without user feedback. The rationale for this is simple—the network connection may be slow and we need to show progress information to the user. An argument could be made for always showing the progress of the parse operation, but this approach gives us an opportunity to demonstrate more conditionally operating code.

Listing 12.11 JobListHandler.java

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source code
public class JobListHandler extends DefaultHandler {
     Handler phandler = null;
     JobList _list;
     JobEntry _job;
     String _lastElementName = "";
     StringBuilder sb = null;
     Context _context;
     JobListHandler(Context c,Handler progressHandler) {     ❶ JobListHandler constructor
          _context = c;
          if (progressHandler != null) {
               phandler = progressHandler;                   ❷ Check for progress handler
               Message msg = new Message();
               msg.what = 0;
               msg.obj = (Object)("Processing List");
               phandler.sendMessage(msg);
          }
     }
     public JobList getList() {                              ❸ getList method
```

```
          Message msg = new Message();
          msg.what = 0;
          msg.obj = (Object)("Fetching List");
          if (phandler != null) phandler.sendMessage(msg);
          return _list;
     }
     public void startDocument() throws SAXException {       4 startDocument method
          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 endDocument method
          Message msg = new Message();
          msg.what = 0;
          msg.obj = (Object)("End of Document");
          if (phandler != null) phandler.sendMessage(msg);
     }
     public void startElement
       (String namespaceURI, String localName,String qName,
       Attributes atts) throws SAXException {
          try {
               sb = new StringBuilder("");
               if (localName.equals("job")) {
                    Message msg = new Message();
                    msg.what = 0;
                    msg.obj = (Object)(localName);
                    if (phandler != null) phandler.sendMessage(msg);
                    _job = new JobEntry();
               }
          } catch (Exception ee) {
          }
     }
     public void endElement
       (String namespaceURI, String localName, String qName)
       throws SAXException {
          if (localName.equals("job")) {                    6 Check for end of job element
               // 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);                              7 Build up String incrementally
     }
}
```

The JobListHandler constructor ❶ takes a single argument of Handler. This value may be null. If null, Message passing is omitted from operation. When reading from a local storage file, this Handler argument is null. When reading data from the server over the internet, with a potentially slow connection, the Message-passing code is utilized to provide feedback for the user in the form of a Progress-Dialog. The ProgressDialog code is shown later in this chapter in the discussion of the RefreshJobs Activity. A local copy of the Handler ❷ is set up when using the ProgressDialog, as described in ❶.

The getList()❸ method is invoked when parsing is complete. The role of getList is to return a copy of the JobList that was constructed during the parse process. When the startDocument() callback method ❹ is invoked by the SAX parser, the initial class instances are established. The endDocument() method ❺ is invoked by the SAX parser when all of the document has been consumed. This is an opportunity for the Handler to perform additional cleanup as necessary. In our example, a message is posted to the user by sending a Message.

For each element in the XML file, the SAX parser follows the same pattern: startElement is invoked, characters() is invoked (one or more times), and endElement is invoked. In the startElement method, you initialize StringBuilder and evaluate the element name. If the name is "job," you initialize the class-level JobEntry instance.

In the endElement() method, the element name is evaluated. If the element name is "job" ❻, the JobListHandler adds this JobEntry to the JobList data member, _joblist, with a call to addJob(). Also in the endElement() method, the data members of the JobEntry instance (_job) are updated. Please see the full source code for more details.

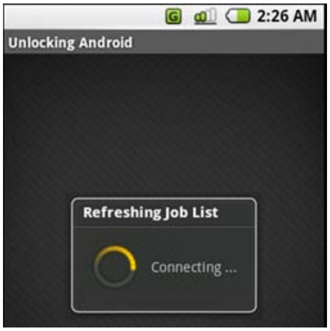
The characters() method is invoked by the SAX parser whenever data is available for storage. The JobListHandler simply appends this string data to a StringBuilder instance ❼ each time it's invoked. It's possible that the characters method may be invoked more than once for a particular element's data. That's the rationale behind using a StringBuilder instead of a single String variable; StringBuilder is a more efficient class for constructing strings from multiple substrings.

After this lengthy but important look into the data structures and the accompanying explanations, it's time to return to the higher-level functionality of the application.

12.4 Source code for managing jobs

Most of the time our mobile worker is using this application, he'll be reading through comments, looking up a job address, getting product information, and performing other aspects of working on a specific job. Our application must supply the functionality for the worker to accomplish each of these job-management tasks. We examine each of these Activitys in detail in this section. The first thing we review is fetching new jobs from the server, which gives us the opportunity to discuss the JobListHandler and the management of the jobs list used throughout the application.

12.4.1 RefreshJobs



Refreshing Job List

Connecting ...

Figure 12.8
The `ProgressDialog` in use during `RefreshJobs`

The `RefreshJobs` `Activity` performs a simple yet vital role in the field service application. Whenever requested, the `RefreshJobs` `Activity` attempts to download a list of new jobs from the server. The UI is super simple—just a blank screen with a `ProgressDialog` informing the user of the application's progress, as shown in figure 12.8.

The code for `RefreshJobs` is shown in listing 12.12. The code is straightforward, as most of the heavy lifting is done in the `JobListHandler`. This code's responsibility is to fetch configuration settings, initiate a request to the server, and put a mechanism in place for showing progress to the user.

Listing 12.12 RefreshJobs.java

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class RefreshJobs extends Activity {
     Prefs myprefs = null;
     Boolean bCancel = false;
     JobList mList = null;
     ProgressDialog progress;
     Handler progresshandler;
     @Override
     public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.refreshjobs);
        myprefs = new Prefs(this.getApplicationContext);
        myprogress = ProgressDialog.show(this, "Refreshing Job List",
          "Please Wait",true,false);
     progresshandler = new Handler() {
           @Override
           public void handleMessage(Message msg) {
            switch (msg.what) {
               case 0:
                 myprogress.setMessage("" + (String) msg.obj);
                 break;
               case 1:
                 myprogress.cancel();
                 finish();
                 break;
               case 2:     // error occurred
                 myprogress.cancel();
                 finish();
                 break;
            }
              super.handleMessage(msg);
            }
```

1 Progress indicator

2 Set up ProgressDialog

3 Define Handler

4 Update UI with textual message

5 Handle cancel and cancel with error

6 Use openFileInput for stream

```
        };
        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;     // error occurred
              msg.obj = (Object)("Caught an error retrieving
               Job data: " + e.getMessage());
              progresshandler.sendMessage(msg);
            }
        }
     }
}
```

7 Initiate DoReadJobs class instance

8 Create Message object

9 Define looping construct

10 Prepare status message

11 Prepare to parse data

12 Instantiate JobListHandler

13 Persist data

14 Set status flag for completion

15 Set status flag for error

A ProgressDialog ❶ is used to display progress information to the user. There are a number of ways to display progress in Android. This is perhaps the most straightforward approach. A Handler is employed to process Message instances. Though the Handler itself is defined as an anonymous class, the code requires a reference to it for passing to the JobListHandler when parsing, which is shown in ⓬. When instantiating the ProgressDialog ❷, the arguments include

- Context
- Title of Dialog
- Initial Textual Message
- Indeterminate
- Cancelable

Using true for the Indeterminate parameter means that you're not providing any clue as to when the operation will complete (such as percentage remaining), just an indicator that something is still happening, which can be a best practice when you don't have a good handle on how long an operation may take. A new Handler ❸ is created to process messages sent from the parsing routine, which will be introduced momentarily. An important class that has been mentioned but thus far not described is Message. This class is used to convey information between different threads of execution. The Message class has some generic data members that may be used in a flexible manner. The first of interest is the what member, which acts as a simple identifier, allowing recipients to easily jump to desired code based on the value of the what member. The most typical (and used here) approach is to evaluate the what data member via a switch statement.

In this application, a Message received with its what member equal to 0 represents a textual update message ❹ to be displayed in the ProgressDialog. The textual data itself is passed as a String cast to an Object and stored in the obj data member of the Message. This interpretation of the what member is purely arbitrary. You could've used 999 as the value meaning textual update, for example. A what value of 1 or 2 indicates that the operation is complete ❺, and this Handler can take steps to initiate another thread of execution. For example, a value of 1 indicates successful completion so the ProgressDialog is canceled, and the RefreshJobs Activity is completed with a call to finish(). The value of 2 for the what member has the same effect as a value of 1, but it's provided here as an example of handling different result conditions; for example, a failure response due to an encountered error. In a production-ready application, this step should be fleshed out to perform an additional step of instruction to the user and/or a retry step. Any Message not explicitly handled by the Handler instance should be passed to the super class ❻. In this way system messages may be processed.

When communicating with a remote resource, such as a remote web server in our case, it's a good idea to perform the communications steps in a thread other than the primary GUI thread. A new Thread ❼ is created based on the DoReadJobs class, which implements the Runnable Java interface. A new Message object 8 is instantiated and initialized. This step takes place over and over throughout the run method of the DoReadJobs class. It's important to not reuse a Message object, as they're literally passed and enqueued. It's possible for them to stack up in the receiver's queue, so reusing a Message object will lead to losing data or corrupting data at best and Thread synchronization issues or beyond at worst.

Why are we talking about a commented-out line of code 9? Great question—because it caused so much pain in the writing of this application! A somewhat odd and confusing element of Android programming is the Looper class. This class provides static methods to help Java Threads to interact with Android. Threads by default don't have a message loop, so presumably Messages don't go anywhere when sent. The first call to make is Looper.prepare(), which creates a Looper for a Thread that doesn't already have one established. Then by placing a call to the loop() method, the flow of Messages takes place. Prior to implementing this class as a Runnable interface, we experimented with performing this step in the same thread and attempted to get the ProgressDialog to work properly. That said, if you run into funny Thread/Looper messages on the Android Emulator, consider adding a call to Looper.prepare() at the beginning of your Thread and then Looper.loop() to help Messages flow.

When you want to send data to the user to inform him of your progress, you update an instance of the Message class 10 and send it to the assigned Handler.

To parse an incoming XML data stream, you create a new InputSource from the URL stream 11. This step is required for the SAX parser. This method reads data from the network directly into the parser without a temporary storage file.

Note that the instantiation of the JobListHandler 12 takes a reference to the progresshandler. This way the JobListHandler can (optionally) propagate messages back to the user during the parse process. Once the parse is complete, the JobListHandler returns a JobList object, which is then persisted 13 to store the data to the local storage. Because this parsing step is complete, you let the Handler know by passing a Message 14 with the what field set to 1. If an exception occurs, you pass a message with what set to 2, indicating an error 15.

Congratulations, your Android application has now constructed a URL object with persistently stored configuration information (user and server) and successfully connected over the internet to fetch XML data. That data has been parsed into a JobList containing JobEntry objects, while providing our patient mobile worker with feedback, and subsequently storing the JobList to the filesystem for later use. Now we want to work with those jobs, because after all, those jobs have to be completed for our mobile worker friend to make a living!

12.4.2 Managing jobs: The ManageJobs Activity

The ManageJobs Activity presents a scrollable list of jobs for review and action. At the top of the screen is a simple summary indicating the number of jobs in the list, and each individual job is enumerated in a ListView.

Earlier we mentioned the importance of the JobEntry's toString() method:

```
public String toString() {
   return this._jobid + ": " + this._customer + ": " + this._product;
}
```

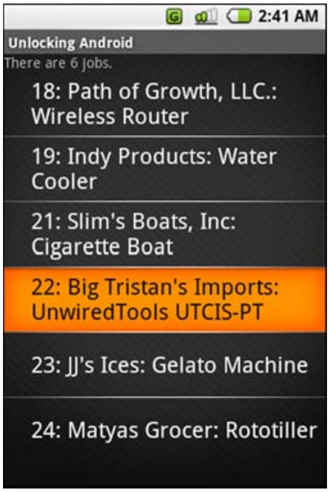
This method generates the `String` that's used to represent the `JobEntry` in the `ListView`, as shown in figure 12.9.

The layout for this `Activity`'s `View` is simple: just a `TextView` and a `ListView`, as shown in the following listing.

Listing 12.13 managejobs.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/joblistview"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:scrollbars="vertical"
    >
    <TextView     android:id="@+id/statuslabel"
    android:text="list jobs here "
    android:layout_height="wrap_content"
    android:layout_width="fill_parent"
    />
    <ListView   android:id="@+id/joblist"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    />
</LinearLayout>
```

The code in listing 12.14 for the `ManageJobs Activity` connects a `JobList` to the GUI and reacts to the selection of a particular job from the `ListView`. In addition, this class demonstrates taking the result from another, synchronously invoked `Activity` and processing it according to its specific requirement. For example, when a job is completed and closed, that `JobEntry` is updated to reflect its new status.



Unlocking Android
There are 6 jobs.
18: Path of Growth, LLC.: Wireless Router
19: Indy Products: Water Cooler
21: Slim's Boats, Inc: Cigarette Boat
22: Big Tristan's Imports: UnwiredTools UTCIS-PT
23: JJ's Ices: Gelato Machine
24: Matyas Grocer: Rototiller

Figure 12.9 `ManageJobs Activity` lists downloaded jobs.

Listing 12.14 ManageJobs.java, which implements the `ManageJobs Activity`

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class ManageJobs extends Activity implements OnItemClickListener {
     final int SHOWJOB = 1;
     Prefs myprefs = null;
     JobList _joblist = null;
     ListView jobListView;
     @Override
     public void onCreate(Bundle icicle) {
       super.onCreate(icicle);
```

```
    setContentView(R.layout.managejobs);

    myprefs = new Prefs(this.getApplicationContext());
    TextView tv =
        (TextView) findViewById(R.id.statuslabel);
    _joblist = JobList.parse(this.getApplicationContext());
    if (_joblist == null) {
     _joblist = new JobList(this.getApplicationContext());
    }

    if (_joblist.getJobCount() == 0){
       tv.setText("There are No Jobs Available");
    } else {
      tv.setText("There are " + _joblist.getJobCount() + " jobs.");
    }

    jobListView = (ListView) findViewById(R.id.joblist);
    ArrayAdapter<JobEntry> adapter = new ArrayAdapter<JobEntry>(this,
    android.R.layout.simple_list_item_1, _joblist.getAllJobs());
    jobListView.setAdapter(adapter);
    jobListView.setOnItemClickListener(this);
    jobListView.setSelection(0);
   }

   public void onItemClick(AdapterView parent,
    View v, int position, long id) {
    JobEntry je = _joblist.getJob(position);
    Log.i("CH12", "job clicked! [" + je.get_jobid() + "]");
    Intent jobintent = new Intent(this, ShowJob.class);
    Bundle b = je.toBundle();
    jobintent.putExtras(b);
    startActivityForResult(jobintent, SHOWJOB);
   }

  @Override
  protected void onActivityResult(int requestCode, int resultCode, Intent
   data) {
   switch (requestCode) {
    case SHOWJOB:
      if (resultCode == 1){
        Log.d("CH12","Good Close, let's update our list");
        JobEntry je = JobEntry.fromBundle(data.getExtras());
       _joblist.replace(je);
      }
      break;
   }
  }

}
```

Connect TextView to UI

❶ Parse data in storage

❷ Handle bad parse

Check for empty JobList

Process click events on List

❸ Connect ListView to UI

❹ Fetch job from list by ordinal

❺ Use Bundle to store Job data

❻ Start ShowJob Activity

❼ Check return code

Extract returned JobEntry

❽ Update the list with via replace method

The objective of this code is to display a list of available jobs to the user in a ListView ❸. To display the list of jobs, we must first parse the list stored on the device ❶. Note that the Context argument is required to allow the JobList class access to the private file area for this application. If the parse fails, we initialize the JobList instance to a new, empty list. This is a somewhat simplistic way to handle the error without the GUI falling apart ❷.

When a specific job is selected, its details are extracted via a call to the getJob method ❹. The job is stored in a Bundle, put into an Intent ❺, and subsequently sent to the ShowJob Activity for display and/or editing ❻. Note the use of the constant SHOWJOB as the last parameter of the startActivityForResult method. When the called-Activity returns, the second parameter to startActivityForResult is "passed back" when the onActivityResult method is invoked ❼ and the return code checked. To obtain the changed JobEntry, you need to extract it from the Intent with a call to getExtras(), which returns a Bundle. This Bundle is turned into a JobEntry instance via the static fromBundle method of the JobEntry class. To update the list of jobs to reflect this changed JobEntry, call the replace method ❽.

More on bundles

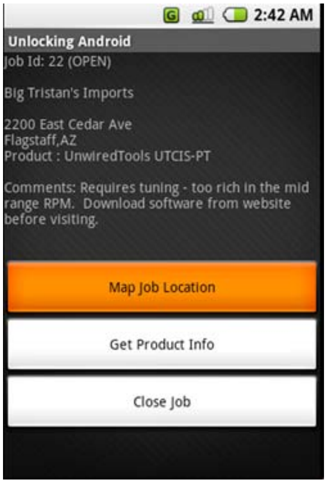
You need to pass the selected job to the ShowJob Activity, but you can't casually pass an object from one Activity to another. You don't want the ShowJob Activity to have to parse the list of jobs again; otherwise you could simply pass back an index to the selected job by using the integer storage methods of a Bundle. Perhaps you could store the currently selected JobEntry (and JobList for that matter) in a global data member of the Application object, had you chosen to implement one. If you recall in chapter 1 when we discussed the ability of Android to dispatch Intents to any Activity registered on the device, you want to keep the ability open to an application other than your own to perhaps pass a job to you. If that were the case, using a global data member of an Application object would never work! The likelihood of such a step is low, particularly considering how the data is stored in this application. This chapter's sample application is an exercise of evaluating some mechanisms you might employ to solve data movement around when programming for Android. The chosen solution is to package the data fields of the JobEntry in a Bundle (❺ in listing 12.14) to move a JobEntry from one Activity to another. In the strictest sense, you're not moving a real JobEntry object but a representation of a JobEntry's data members. The net of this discussion is that this method creates a new Bundle by using the toBundle() method of the JobEntry.

Now that you can view and select the job of interest, it's time to look at just what you can do with that job. Before diving into the next section, be sure to review the ManageJobs code carefully to understand how the JobEntry information is passed between the two activities.

12.4.3 Working with a job with the ShowJob Activity

The ShowJob Activity is the most interesting element of the entire application, and it's certainly the screen most useful to the mobile worker carrying around his Android-capable device and toolbox. To help in the discussion of the various features available to the user on this screen, take a look at figure 12.10.

The layout is straightforward, but this time you have some Buttons and you'll be changing the textual description depending on the condition of a particular job's status. A TextView is used to present job details such as address, product requiring


2:42 AM

Unlocking Android

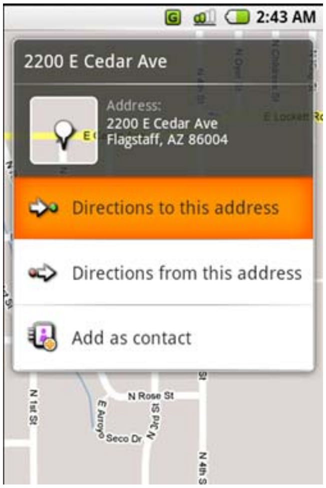
Job Id: 22 (OPEN)

Big Tristan's Imports

2200 East Cedar Ave
Flagstaff,AZ
Product : UnwiredTools UTCIS-PT

Comments: Requires tuning - too rich in the mid range RPM. Download software from website before visiting.

Figure 12.10 An example of a job shown in the ShowJob Activity


2:43 AM

2200 E Cedar Ave

Address:
2200 E Cedar Ave
Flagstaff, AZ 86004

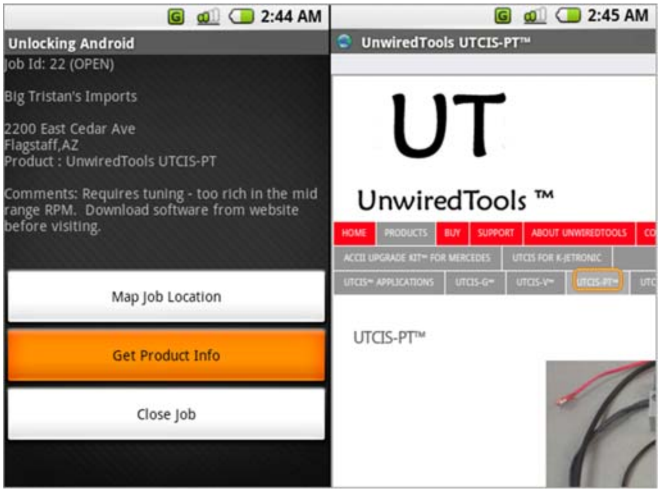
Figure 12.11 Viewing a job address in the Maps application

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.

The second button, Get Product Info, launches a browser window to assist users in learning more about the product they're being called on to work with. Figure 12.12 shows this in action.

The third requirement is to allow the user to close the job or to view the signature if it's already closed; we'll cover the details in the next section on the `CloseJob Activity`.


2:44 AM

Unlocking Android

Job Id: 22 (OPEN)

Big Tristan's Imports

2200 East Cedar Ave
Flagstaff,AZ
Product : UnwiredTools UTCIS-PT

Comments: Requires tuning - too rich in the mid range RPM. Download software from website before visiting.

Figure 12.12 Get Product Info takes the user to a web page specific to this job.

Fortunately, the steps required for the first two operations are quite simple with Android—thanks to the Intent. The following listing and the accompanying annotations show you how.

Listing 12.15 ShowJob.java

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class ShowJob extends Activity  {
     Prefs myprefs = null;
     JobEntry je = null;
     final int CLOSEJOBTASK = 1;
     public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.showjob);
        myprefs = new Prefs(this.getApplicationContext());
        StringBuilder sb = new StringBuilder();
        String details = null;
        Intent startingIntent = getIntent();
        if (startingIntent != null) {
             Bundle b = startingIntent.getExtas();
             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.";
          TextView tv = (TextView) findViewById(R.id.details);
          tv.setText(details);
          return;
        }
        TextView tv = (TextView) findViewById(R.id.details);
        tv.setText(details);
        Button bmap = (Button) findViewById(R.id.mapjob);
        bmap.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
              // clean up data for use in GEO query
              String address = je.get_address() + " " +
 je.get_city() + " " +
                je.get_zip();
              String cleanAddress = address.replace(",", "");
              cleanAddress = cleanAddress.replace(' ','+');
              try {
                Intent geoIntent = new Intent("android.intent.action.VIEW",
android.net.Uri.parse("geo:0,0?q=" +
```

Get Intent

Extract Bundle from Intent

Update UI upon error and return

```
         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")) {
           bclose.setText("Job is Closed. View Signature");
        }
        bclose.setOnClickListener(new Button.OnClickListener() {
            public void onClick(View v) {
            if (je.get_status().equals("CLOSED")) {
               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);
            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());
                finish();
              }
              break;
    }
  }
}
```

Build and launch geo query

Obtain product info via URL

Selectively update Button label

Initiate CloseJob Activity

1 Handle newly closed JobEntry

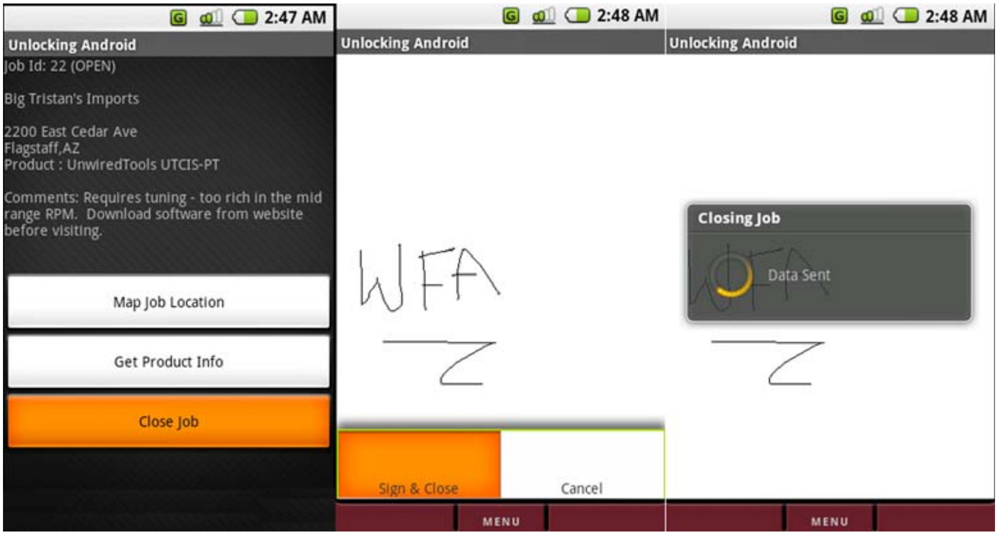
Upon completion of the CloseJob Activity, the onActivityResult callback is invoked. When this situation occurs, this method receives a Bundle containing the data elements for the recently closed JobEntry ❶. If you recall, the ShowJob Activity was launched "for result.", which permits a synchronous pattern, passing the result back to the caller. The requirement is to propagate this JobEntry data back up to the calling Activity, ManageJobs. Calling setResult() and passing the Bundle (obtained with getExtras()) fulfills this requirement.

Despite the simple appearance of some text and a few easy-to-hit buttons, the ShowJob Activity provides a significant amount of functionality to the user. All that remains is to capture the signature to close out the job. Doing so requires an examination of the CloseJob Activity.

12.4.4 Capturing a signature with the CloseJob Activity

Our faithful mobile technician has just completed the maintenance operation on the part and is ready to head off to lunch before stopping for another job on the way home, but first he must close out this job with a signature from the customer. To accomplish this, the field service application presents a blank screen, and the customer uses a stylus (or a mouse in the case of the Android Emulator) to sign the device, acknowledging that the work has been completed. Once the signature has been captured, the data is submitted to the server. The proof of job completion has been captured, and the job can now be billed. Figure 12.13 demonstrates this sequence of events.

This Activity can be broken down into two basic functions: the capture of a signature and the transmittal of job data to the server. Notice that this Activity's UI has no layout resource. All of the UI elements in this Activity are generated dynamically, as



Job Id: 22 (OPEN)

Big Tristan's Imports

2200 East Cedar Ave
Flagstaff,AZ
Product : UnwiredTools UTCIS-PT

Comments: Requires tuning - too rich in the mid range RPM. Download software from website before visiting.

Figure 12.13 The CloseJob Activity capturing a signature and sending data to the server

Local queuing

One element not found in this sample application is the local queuing of the signature. Ideally this would be done in the event that data coverage isn't available. The storage of the image is quite simple; the perhaps more challenging piece is the logic on when to attempt to send the data again. Considering all the development of this sample application is done on the Android Emulator with near-perfect connectivity, it's of little concern here. But in the interest of best preparing you to write real-world applications, it's worth reminding you of local queuing in the event of communications trouble in the field.

shown in listing 12.16. In addition, the `ProgressDialog` introduced in the `RefreshJobs Activity` is brought back for an encore, to let our mobile technician know that the captured signature is being sent when the Sign & Close menu option is selected. If the user selects Cancel, the `ShowJob Activity` resumes control. Note that the signature should be made prior to selecting the menu option.

Listing 12.16 CloseJob.java—GUI setup

```
package com.msi.manning.UnlockingAndroid;
// multiple imports omitted for brevity, see full source
public class CloseJob extends Activity {
     ProgressDialog myprogress;
     Handler progresshandler;
     Message msg;
     JobEntry je = null;
     private closejobView sc = null;
    @Override
    public void onCreate(Bundle icicle)  {
        super.onCreate(icicle);
        Intent startingIntent = getIntent();
        if (startingIntent != null) {
           Bundle b = startingIntent.getExtras()
           if (b != null) {
             je = JobEntry.fromBundle(b);
           }
        }
        sc = new closejobView(this);
        setContentView(sc);
        if (je == null) {

          finish();
        }
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
     super.onCreateOptionsMenu(menu);
     menu.add(0,0,"Sign & Close");
     menu.add(0,1,"Cancel");
     return true;
    }
```

1 Instantiate instance of closejobView

2 Define available menus

```
    public boolean onOptionsItemSelected(Menu.Item item) {
      Prefs myprefs = new Prefs(CloseJob.this.getApplicationContext());
      switch (item.getId()) {                                          Handle
         case 0:                                                  selected menu ❸
          try {
             myprogress = ProgressDialog.show(this, "Closing Job ",
               "Saving Signature to Network",true,false);
             progresshandler = new Handler() {
             @Override
             public void handleMessage(Message msg) {
              switch (msg.what) {
                 case 0:
                    myprogress.setMessage("" + (String) msg.obj);
                    break;
                 case 1:
                    myprogress.cancel();
                    finish();
                    break;
              }
         super.handleMessage(msg);
        }
      };
      Thread workthread = new                                 ❹ Start Thread
Thread(new DoCloseJob(myprefs));                                to CloseJob
           workthread.start();
    } catch (Exception e) {
       Log.d("closejob",e.getMessage());
       msg = new Message();
       msg.what = 1;
       progresshandler.sendMessage(msg);
    }
    return true;
  case 1:
     finish();
     return true;
  }
  return false;
}
```

Unlike previous activities in this chapter, the UI doesn't come from a design time-defined layout, but rather an instance of a closejobView ❶ is the primary UI. The closejobView is defined in listing 12.17.

The onCreateOptionsMenu method ❷ is an override of the base View's method, allowing a convenient way to add menus to this screen. Note that two menus are added, one for Sign & Close and one for Cancel. The onOptionsItemSelected method ❸ is invoked when the user selects a menu item. A ProgressDialog and accompanying Handler are instantiated when the user chooses the menu to close a job. Once the progress-reporting mechanism is in place, a new Thread is created and started in order to process the steps required to close the job ❹. Note that an instance of Prefs is passed in as an argument to the constructor, as that will be needed to store a signature, as we'll show in listing 12.18.

The UI at this point is only partially set up; we need a means to capture a signature on the screen of our Android device. The next listing implements the class closejobView, which is an extension of the View class.

Listing 12.17 CloseJob.java—closejobView class

```
public class closejobView extends View {
     Bitmap _bitmap;
     Canvas _canvas;
     final Paint _paint;
     int lastX;
     int lastY;
     public closejobView(Context c) {
       super(c);
       _paint = new Paint();
       _paint.setColor(Color.BLACK);
       lastX = -1;
     }
     public boolean Save(OutputStream os){
        try {
          _canvas.drawText("Unlocking Android", 10, 10, _paint);
          _canvas.drawText("http://manning.com/ableson", 10, 25, _paint);
          _canvas.drawText("http://android12.msi-wireless.com",
               10, 40, _paint);
          _bitmap.compress(Bitmap.CompressFormat.JPEG, 100, os);
          invalidate();
          return true;
        } catch (Exception e) {
           return false;
        }
     }
     @Override
     protected void onSizeChanged(int w, int h, int oldw, int oldh) {
      Bitmap img =
         Bitmap.createBitmap(w, h,Bitmap.Config.ARGB_8888);
      Canvas canvas = new Canvas();
      canvas.setBitmap(img);
      if (_bitmap != null) {
         canvas.drawBitmap(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);
       }
      }
      @Override
      public boolean onTouchEvent(MotionEvent event) {
       int action = event.getAction();
       int X = (int)event.getX();
       int Y = (int)event.getY();
```

1 closejobView extends base class View

2 Required classes for drawing

Initialize drawing classes

3 Save method persists signature

4 Add contextual data to image

5 Draw image on screen

6 Handle touch events

```
       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;
      }
    }
}
```

The closejobView extends the base View class ❶. The Bitmap and Canvas classes ❷ work together to form the drawing surface for this Activity. Note the call to the Canvas.drawColor method, which sets the background color to white. When the onDraw() method is invoked, the canvas draws its associated bitmap with a call to drawBitmap() ❺.

The logic for where to draw relies on the onTouchEvent method ❻, which receives an instance of the MotionEvent class. The MotionEvent class tells what happened and where. ACTION_UP, ACTION_DOWN, and ACTION_MOVE are the events captured, with some logic to guide when and where to draw. Once the signature is complete, the Save method ❸ is responsible for converting the contents of the image to a form usable for submission to the server. Note that additional text is drawn on the signature ❹. In this case, it's little more than a shameless plug for this book's web page, but this could also be location-based data. Why is this important? Imagine someone forging a signature. It could happen, but it would be more challenging and of less value to a rogue mobile technician if the GPS/location data were stamped on the job, along with the date and time. When converting the image to our desired JPEG format, there's an additional input argument to this method—an OutputStream, used to store the image data. This OutputStream reference was an input argument to the Save method.

Now that the UI has been created and a signature drawn on the screen, let's look at the code used to close the job. Closing the job involves capturing the signature and sending it to the server via an HTTP POST. The class DoCloseJob is shown in the following listing.

Listing 12.18 CloseJob.java—DoCloseJob class

```
 class DoCloseJob implements Runnable {
 Prefs _myprefs;
 DoCloseJob(Prefs p) {
   _myprefs = p;
 }
 public void run() {
   try {
       FileOutputStream os =
       getApplication().openFileOutput("sig.jpg", 0);
        sc.Save(os);
       os.flush();
       os.close();
       // reopen to so we can send this data to server
       File f = new
File(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) {
           bSuccess = true;
        }
     }
     wr.close();
     rd.close();
     if (bSuccess) {
        msg = new Message();
```

Constructor uses Prefs instance

1 Open file for storing signature

2 Construct storage URL

3 Write data to server

4 Read server response

5 Check for successful processing

```
             msg.what = 0;
             msg.obj = (Object)("Job Closed Successfully");
             progresshandler.sendMessage(msg);
             je.set_status("CLOSED");
             CloseJob.this.setResult(1,"",je.toBundle());
          } else {
            msg = new Message();
            msg.what = 0;
            msg.obj = (Object)("Failed to Close Job");
            progresshandler.sendMessage(msg);
            CloseJob.this.setResult(0);
          }
     } catch (Exception e) {
         Log.d("CH12","Failed to submit job close signature: " +
     e.getMessage());
     }
     msg = new Message();
     msg.what = 1;
     progresshandler.sendMessage(msg);
  }
}
```

❻ Update local JobEntry status

❼ Set result and store updated JobEntry

At this point, you have a signature on the screen and need to capture it. A new FileOutputStream ❶ is obtained for a file on the local filesystem, and the signature is written to this file. You're now ready to transmit this file to the server—remember, you want to bill the client as soon as possible for work completed!

In preparation for sending the signature to the server, the signature file contents are read into a byte array via an instance of a FileInputStream. Using the Prefs instance to get specific configuration information, a URL ❷ is constructed in order to POST data to the server. The query String of the URL contains the jobid and the POST data contains the image itself. A BufferedOutputStream ❸ is employed to POST data, which consists of the captured signature in JPEG format.

Once the job data and signature have been sent to the server, the response data is read back from the server ❹. A specific string indicates a successful transmission ❺.

Upon successful closing, the JobEntry status member is marked as CLOSED ❻, and this JobEntry is converted to a Bundle so that it may be communicated to the caller by invoking the setResult() method ❼. Once the Handler receives the "I'm done" message and the Activity finishes, this data is propagated back to the ShowJob and all the way back to the ManageJob Activity.

And that thankfully wraps up the source code review for the Android side of things! There were some methods omitted from this text to limit this already very long chapter, so please be sure to examine the full source code. Now it's time to look at the server application. jobs

12.5 Server code

A mobile application often relies on server-side resources, and our field service application is no exception. This isn't a book on server-side development techniques, server-related code, and discussion, so we'll present these things briefly. We'll introduce the

UI and the accompanying database structure that makes up our list of job entries, and then we'll review the two server-side transactions that concern the Android application. The server code relies on open source staples: MySQL and PHP. Let's get started with the interface used to enter new jobs, used by the dispatcher.

12.5.1 Dispatcher user interface

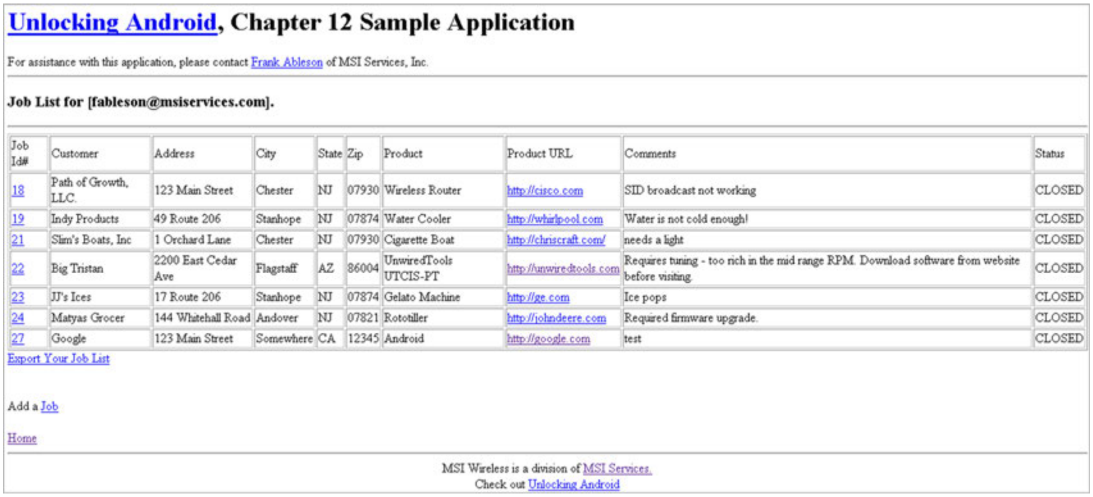
Before jumping into any server code–specific items, it's important to understand how the application is organized. All jobs entered by a dispatcher are assigned to a particular mobile technician. That identifier is interpreted as an email address, as seen in the Android example where the user ID was used throughout the application. Once the user ID is specified, all of the records revolve around that data element. For example, figure 12.14 demonstrates this by showing the jobs assigned to the author, fableson@msiservices.com.

> **NOTE** This application is available for testing the sample application yourself. It's located at http://android12.msi-wireless.com. Sign on and add jobs for your email address.

Let's now turn our attention to the underlying data structure, which contains the list of jobs.

12.5.2 Database

As mentioned earlier in section 12.1.3, the database in use in this application is MySQL,[2] with a single database table called tbl_jobs. The SQL to create this table is provided in the next listing.



Unlocking Android, Chapter 12 Sample Application

For assistance with this application, please contact Frank Ableson of MSI Services, Inc.

Job List for [fableson@msiservices.com].

Job Id#	Customer	Address	City	State	Zip	Product	Product URL	Comments	Status
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	Stanhope	NJ	07874	Water Cooler	http://whirlpool.com	Water is not cold enough!	CLOSED
21	Slim's Boats, Inc	1 Orchard Lane	Chester	NJ	07930	Cigarette Boat	http://chriscraft.com/	needs a light	CLOSED
22	Big Tristan	2200 East Cedar Ave	Flagstaff	AZ	86004	UnwiredTools UTCIS-PT	http://unwiredtools.com	Requires tuning - too rich in the mid range RPM. Download software from website before visiting.	CLOSED
23	JJ's Ices	17 Route 206	Stanhope	NJ	07874	Gelato Machine	http://ge.com	Ice pops	CLOSED
24	Matyas Grocer	144 Whitehall Road	Andover	NJ	07821	Rototiller	http://johndeere.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

[2] For more on development using MySQL, try the developer zone: http://dev.mysql.com/.

Listing 12.19 Data definition for tbl_jobs

```
CREATE TABLE IF NOT EXISTS 'tbl_jobs' (
  'jobid' int(11) NOT NULL auto_increment,            ❶ Unique record ID
  'status' varchar(10) NOT NULL default 'OPEN',
  'identifier' varchar(50) NOT NULL,             ❷ User identification
  'address' varchar(50) NOT NULL,
  'city' varchar(30) NOT NULL,
  'state' varchar(2) NOT NULL,
  'zip' varchar(10) NOT NULL,
  'customer' varchar(50) NOT NULL,
  'product' varchar(50) NOT NULL,
  'producturl' varchar(100) NOT NULL,            ❸ Product URL
  'comments' varchar(100) NOT NULL,
  UNIQUE KEY 'jobid' ('jobid')
) ENGINE=MyISAM  DEFAULT CHARSET=ascii AUTO_INCREMENT=25 ;
```

Each row in this table is uniquely identified by the `jobid` ❶, which is an auto-incrementing integer field. The `identifier` field ❷ corresponds to the user ID/email of the assigned mobile technician. The `producturl` field ❸ is designed to be a specific URL to assist the mobile technician in the field in quickly gaining access to helpful information for completing the assigned job.

The next section provides a road map to the server code.

12.5.3 PHP dispatcher code

The server-side dispatcher system is written in PHP and contains a number of files working together to create the application. Table 12.3 presents a brief synopsis of each source file to help you navigate the application if you choose to host a version of it yourself.

Table 12.3 Server-side source code

Source file	Description
addjob.php	Form for entering new job information
closejob.php	Used by Android application to submit signature
db.php	Database connection information
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 the Android application to request a job XML stream
header.php	Used to create a consistent look and feel for the header of each page
index.php	Home page, including the search form
manage.php	Used to delete jobs on the web application
savejob.php	Used to save a new job (called from addjob.php)
showjob.php	Used to display job details and load into a form for updating
showjobs.php	Displays all jobs for a particular user

Table 12.3 Server-side source code *(continued)*

Source file	Description
updatejob.php	Used to save updates to a job
utils.php	Contains various routines for interacting with the database

Of all these files, only two concern the Android application. We'll discuss them in the next section.

12.5.4 PHP mobile integration code

When the Android application runs the `RefreshJobs Activity`, the server side generates an XML stream. Without going into excessive detail on the server-side code, we explain the getjoblist.php file in the following listing.

Listing 12.20 getjoblist.php

```
<?
require('db.php');
require('utils.php');
$theuser = $_GET['identifier'];
print (getJobsXML($theuser));
?>
```

The `getJobsXML` function retrieves data from the database and formats each row into an XML representation. It wraps the list of XML-wrapped job records in the `<joblist>` tags along with the `<?xml ...>` header declaration to generate the expected XML structure used by the Android application. Remember, this is the data ultimately parsed by the SAX-based `JobListHandler` class, as shown in listing 12.11.

The other transaction that's important to our Android field service application is the closejob.php file, examined in the next listing.

Listing 12.21 closejob.php

```
<?
require('db.php');
require('utils.php');
$data = file_get_contents('php://input');
$jobid = $_GET['jobid'];
$f = fopen("~/pathtofiles/sigs/".$jobid.".jpg","w");
fwrite($f,$data);
fclose($f);
print(closeJob($_GET['jobid']));
?>
```

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`. The `jobid` is extracted from the query `String`. The image file is written out to a directory that contains signatures as JPEG files, keyed by the jobid as part of the filename. When a job has been closed and the signature is requested by the Android application, this file is requested in the Android browser. The closeJob function (implemented in utils.php) updates the database to mark the selected job as CLOSED.

That wraps up the review of the source code for this chapter's sample application.

12.6 Summary

The intent of the sample application was to tie together many things learned in previous chapters into a composite application. Our field service application has real-world applicability to the kind of uses an Android device is capable of bringing to fruition. Is this sample application production ready? Of course not, but almost! That, as they say, is an exercise for the reader.

Starting with a simple splash screen, this application demonstrates the use of Handlers and displaying images stored in the resources section of an Android project. Moving along to the main screen, a simple UI leads to different activities useful for launching various aspects of the realistic application.

Communications with the server involve downloading XML data, while showing the user a ProgressDialog along the way. Once the data stream commences, the data is parsed by the SAX XML parser, using a custom Handler to navigate the XML document.

We demonstrated that managing jobs in a ListView is as easy as tapping on the desired job in the list. The next screen, the ShowJobs Activity, allows even more functionality, with the ability to jump to a Map showing the location of the job and even a specific product information page customized to this job. Both of those functions are as simple as preparing an Intent and a call to startActivity().

Once the mobile technician completes the job in the field, the CloseJob Activity brings the touch-screen elements into play by allowing the user to capture a signature from his customer. That digital signature is then stamped with additional, contextual information and transmitted over the internet to prove the job was done. Jumping back to what you learned earlier, it would be straightforward to add location-based data to further authenticate the captured signature.

The chapter wrapped up with a quick survey of the server-side components to demonstrate some of the steps necessary to tie the mobile and the server sides together.

The sample application is hosted on the internet and is free for you to test out with your own Android application, and the full source code is provided for the Android and server applications discussed in this chapter.

Now that we've shown what can be accomplished when exercising a broad range of the Android SDK, the next chapter takes a decidedly different turn, as we explore the underpinnings of Android a little deeper and look at building native C applications for the Android platform.

13 Building Android applications in C

This chapter covers

- Building an application in C
- Using dynamic linking
- Building a DayTime Server in C
- Building a Daytime Client in Java

Up to this point, 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. In chapter 12 you built a comprehensive application, building on what we introduced in the prior chapters. As you embark on this chapter, you're temporarily leaving behind the comforts of working strictly in the Android SDK, Java, and Eclipse. We'll instead take a close look at the underlying Linux underpinnings of the Android platform—and more specifically, you'll learn how to build an application in C, without the SDK.

The Android SDK is comprehensive and capable, 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're going to use the C programming language. In this chapter, we 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.

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 mobile platforms. This technique has the potential benefit of speeding up development for adding future functionality to Android by leveraging existing code bases. Chapter 19 examines the Android Native Developer's kit (NDK). Using the NDK, programmers can leverage existing C code and map those routines to applications written in Java. This chapter doesn't use the NDK, but rather looks at building standalone C applications capable of running on the Android platform.

We demonstrate the specific steps of building an Android/Linux application in C. We begin with a description of the environment and the required tool chain. After an obligatory Hello World–caliber application, you'll construct a more sophisticated application that implements a *DayTime Server.* Ultimately any application built for Android/Linux needs to bring value to the user in some form. In an effort to meet this objective, it's desirable that Android/Java be able to interact in a meaningful manner with our Android/Linux application. To that end, you'll 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 for building your first C application for Android.

13.1 Building Android apps without the SDK

Applications for Android/Linux are markedly different from applications constructed with the Android SDK. Applications built with Eclipse and the context-sensitive Java syntax tools make 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. Though the Eclipse environment could certainly be leveraged for non-Java development, the focus of this chapter is on core C language[1] 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 The C compiler and linker tools

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 https://support.codesourcery.com/GNUToolchain/release1479. Once installed, the Sourcery G++ tool chain contributes a number of useful tools to assist you in creating applications targeting Linux on ARM, which is the architecture of the Android platform. The ARM platform is a 32-bit reduced instruction set computer

[1] For details on the C programming language start here: http://www.cprogramming.com/.

(RISC) processor, used in numerous devices, including smartphones, PDAs, and technology appliances such as low-end routers and disk drive controllers. 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, but three tools in particular are demonstrated in the construction of this chapter's sample applications. You'll be using these tools right away, so let's briefly introduce them in this section.

The first and most important tool introduced is *gcc*.[2] 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 relocatable, executable binary file, compatible with the Android Emulator's Linux environment. Whereas 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. The `objdump`, or `arm-none-linux-gnueabi-objdump` tool examines an executable application—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's also possible to use this tool chain on a Linux development machine. If you are using Linux for your development environment, you may need to modify the build scripts slightly as the path separator is different and the libraries will require a preceeding dot (".").

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 a Hello World application

The first thing we want you to accomplish with your journey into Android/Linux development is to print something to the emulator screen to demonstrate that you're running something on the platform outside the Android SDK and its Java application environment. There's no better way to accomplish this feat than by writing a variant of

[2] For everything you'd want to know about gcc, go here: http://gcc.gnu.org/.

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. The following listing shows the code for your first Hello Android application.

Listing 13.1 Hello.c

```
#include <stdio.h>
int main(int argc,char * argv[])
{
      printf("Hello, Android!\n");
      return 0;
}
```

Virtually all C language applications require an `#include` 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. The standard C language entry point for user code is the function named `main`. The function returns an integer return code (a value of 0 is returned in this simple example) and takes two arguments. The first, `argc`, is an integer indicating the number of command-line arguments passed in to the program when invoked. The second, `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. This application has but a single useful instruction, `printf`, which is to write to standard output (the screen) a textual string. The `printf` function is declared in the header file, stdio.h.

To build this application, you employ the gcc tool:

```
arm-none-linux-gnueabi-gcc hello.c -static -o hellostatic
```

You'll notice a few things about this command-line instruction:

- The compiler is invoked with the full name: `arm-none-linux-gnueabi-gcc`.
- The source file is named hello.c.
- 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 standalone and ready to be run on the target Android Emulator without any additional components. An application that's statically linked tends to be rather large, because so much code and data are included in the executable file. For example, this statically linked application with basically a single line of code weighs in at around 600 KB. 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, but it'll rely on finding compatible routines on the target system in order to run. For now, let's keep things simple and build the sample application in such a manner that all support routines are linked statically.
- The output switch, `-o`, is used to request that the executable application be assigned the name hellostatic. If this switch isn't provided, the default application name is a.out.

Now that the application is built, it's time for you to try it out on the Android Emulator. To do this, you'll rely on the adb tool introduced in chapter 2.

13.1.3 Installing and running the application

In preparation for installing and running the Hello Android application, let's take a tour of our build and testing environment. You need to identify four distinct environments and tools and clearly understand them when building applications for Android/Linux: Android Emulator, command-line CodeSourcery tools, adb or DDMS, and adb shell.

The first environment to grasp is the big-picture architecture of the Android Emulator running essentially on top of Linux, as shown in figure 13.1.

As presented in the early chapters of this book, there's 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.

Figure 13.1 Android runs atop a Linux kernel.

As shown in the previous section, you need an environment in which to compile your 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 you need to be comfortable operating within.

The next requirement is to copy your newly constructed binary executable application to the Android Emulator. You can do so 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's the syntax for copying the executable file to the Android Emulator:

```
adb push hellostatic /data/ch13/hellostatic
```

Cross compiling

The CodeSourcery tool chain isn't designed to run on the Android/Linux environment itself, so the development work being done here is considered to be *cross-compiling*. The figures and example code presented in this chapter were taken from a Windows development environment used by one of the authors. There are a number of long path and directory structures in the Android SDK and the CodeSourcery tools. To help simplify some of the examples and keep certain command-line entries from running over multiple lines, we set up some drive mappings. For example, a drive letter of `m:` seen in scripts and figures corresponds to the root location of source code examples on the author's development machine. Likewise, the `g:` drive points to the currently installed Android SDK on the author's development machine. Note that this technique may also be used in Linux or Mac OS X environments with a "soft link" (`ln`) command.

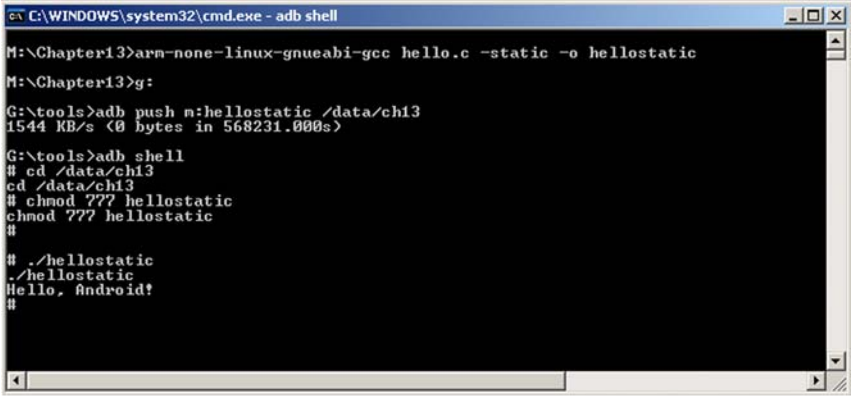
Note a few items about this command:

- The command name is adb. This command takes a number of arguments that guide its behavior. In this case, the subcommand is push, which means to copy a file to the Android Emulator. There's also a pull option for moving files from the Android Emulator filesystem to the local development machine's hard drive.
- After the push option, the next argument, hellostatic in this case, represents the local file, stored on the development machine's hard drive.
- The last argument is the destination directory (and/or filename) for the transferred file. In this sample, you're copying the hellostatic file from the current working directory to the /data/ch13 directory on the Android Emulator.

Be sure that the desired target directory exists first! You can accomplish this with a mkdir command on the adb shell, described next.

The final tool to become familiar with is the shell option of the adb tool. Using this command, we can interact directly on the Android Emulator's filesystem with a limited shell environment. To enter this environment (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'd made a secure shell (ssh) or telnet connection to a remote Unix-based machine. Figure 13.2 shows these steps in action.

Note the sequence shown in figure 13.2. First the application is built with a call to gcc. Next you push the file over to the Android Emulator. You then connect to the Android Emulator via the adb shell command, which gives you the # prompt, indicating that you're now in the shell. Next you 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, tuning on the executable bits and allowing the application to be invoked. Lastly, you invoke the application with a call to ./hellostatic. The search path for executable applications doesn't by default include the current directory on a Linux system, so you must provide a more properly qualified path, which explains the ./ prefix. Of course, you can see that our application has run successfully because you see the "Hello, Android!" text displayed on the screen.



```
M:\Chapter13>arm-none-linux-gnueabi-gcc hello.c -static -o hellostatic

M:\Chapter13>g:

G:\tools>adb push m:hellostatic /data/ch13
1544 KB/s (0 bytes in 568231.000s)

G:\tools>adb shell
# cd /data/ch13
cd /data/ch13
# chmod 777 hellostatic
chmod 777 hellostatic
#

# ./hellostatic
./hellostatic
Hello, Android!
#
```

Figure 13.2 **The build, copy, run cycle**

Congratulations! You have a successful, albeit simple, Android/Linux application running on the Android Emulator. In the next section, we look at streamlining this build process by combining the multiple build operations into a script.

13.1.4 C application build script

In the previous section, we reviewed each step in building and preparing to test our application. Due to the rather tedious nature of executing each of these steps, you likely want 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; when multiple source files must be linked together, the thought of having a build script is appealing. The need for a build script (shown in listing 13.2) is particularly evident where there are numerous source files to compile and link, a situation you'll encounter later in this chapter.

This listing 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 hellostatic
g:\tools\adb push hellostatic /data/ch13
g:\tools\adb shell "chmod 777 /data/ch13/hellostatic"
```

A call to `arm-none-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 hellostatic. The file hellostatic is copied to the Android Emulator and placed in the directory /data/ch13. 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.

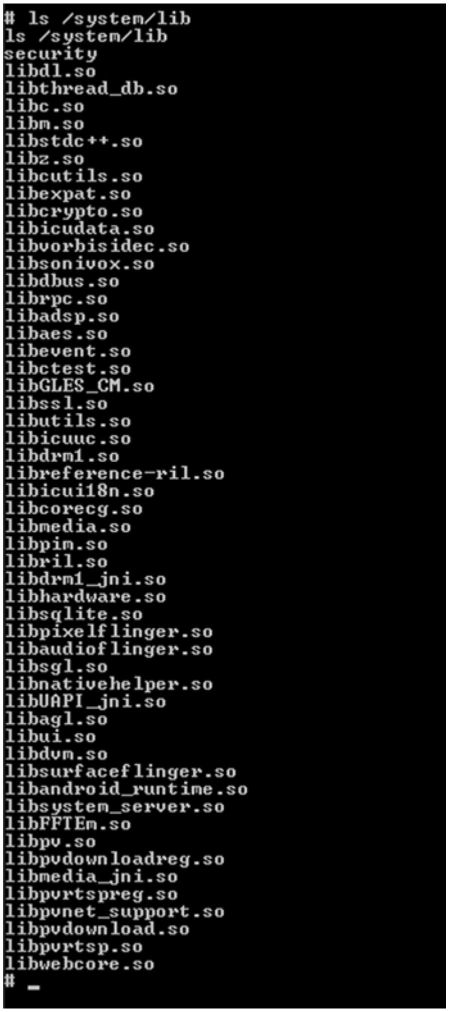
This example can be extended to perform other build steps or cleanup procedures such as removing temporary test data files on the Android Emulator or any similarly helpful tasks. As you progress, it'll become clear what commands you need to put into your build script to make the testing process more efficient.

Now that the pressure is off—you've successfully written, built, and executed an application in the Android/Linux environment—it's time to deal with the problematic issue of a simple application requiring such an enormous file size!

13.2 Solving the problem with dynamic linking

That was fun, but who wants a 500+ KB file that only displays something to the screen? Recall that the `-static` flag links the essentials for running the application, including the input/output routines required for printing a message to the screen. If you're thinking that there must be a better way, you're correct; you need to link the application to existing system libraries rather than include all that code in the application's executable file.

13.2.1 Android system libraries



```
# 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
libFFTEm.so
libpv.so
libpvdownloadreg.so
libmedia_jni.so
libpvrtspreg.so
libpvnet_support.so
libpvdownload.so
libpvrtsp.so
libwebcore.so
# _
```

Figure 13.3 **Available libraries in /system/lib**

When an application is built with the `-static` flag, it's entirely self-contained, meaning that all the routines it requires are linked directly into the application. This information isn't new to you; we've already discussed this. 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 deeper to understand why. To do this, we have to look at the filesystem of Android/Linux.

System libraries in Android/Linux are stored in the directory /system/lib. This directory contains important functionality, such as OpenGL, SQLite, C standard routines, Android runtime, UI routines, and much more. Figure 13.3 shows a list of the available libraries in the Android Emulator. In short, everything that's specific to the Android platform is found in /system/lib, so if you're going to build an application that has any significant functionality, you can't rely on the libraries that ship with CodeSourcery alone. You have to write an application that can interact with the Android system libraries. This calls for a side trip to discuss the functionality of the linker application.

When you're building an application that 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 to inform the tool to simply compile the application and leave the link step to a subsequent build step. Here's 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 hello.o.

This process is repeated for as many source files as necessary for a particular application. For our sample application, you have only this single source file. But to get an executable application, you must employ the services of the linker.

Another important change in the build environment is that you have to get a copy of the Android/Linux libraries. You're compiling on the Windows platform (or Linux if you prefer), so you need to get access to the Android Emulator's /system/lib contents in order to properly link against the library files. Just how do you go about this? You 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
adb pull /system/lib/libthread_db.so    m:\android\system\lib
adb pull /system/lib/libc.so    m:\android\system\lib
adb pull /system/lib/libm.so    m:\android\system\lib
adb pull /system/lib/libGLES_CM.so    m:\android\system\lib
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
many entries omitted for brevity
```

libsqlite.so, SQLite database

Figure 13.4 shows these files now copied over to the development machine.

Once these files are available on the development machine, you can proceed with the build step using the linker.

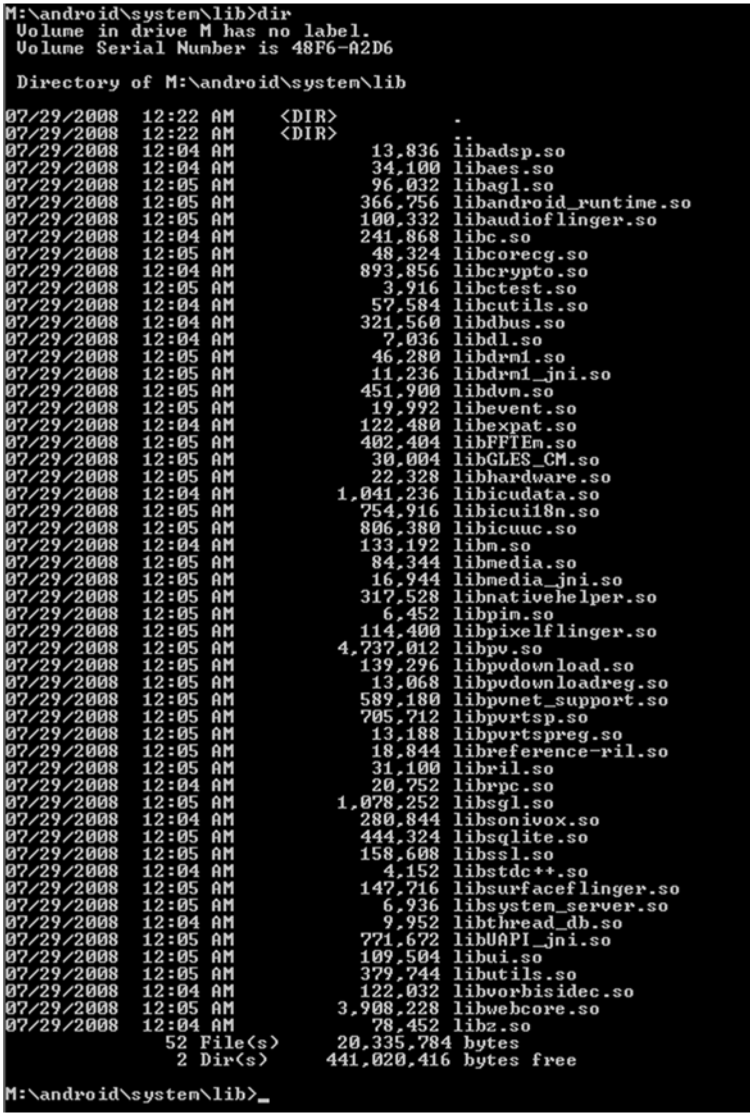
13.2.2 Building a dynamically linked application

The name for the linker is `arm-none-linux-gnueabi-ld`. In most Linux environments, the linker is named simply `ld`. When you're using the linker, many command-line options are available to you for controlling the output. There are so many options that we could write an entire book covering no other topic. Our interest in this chapter is writing applications, and we're taking as streamlined an approach as possible. So although there may be other options that can get the job done, our aim here is to show you how to build an application that gives you as much flexibility as possible to employ the Android system libraries. To that end, the following listing shows the script for building a dynamic version of Hello Android.

Listing 13.4 Build script for dynamically linked Android application

```
arm-none-linux-gnueabi-gcc -c hello.c -o hello.o
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
  hellodynamic hello.o
g:\tools\adb push hellodynamic /data/ch13
g:\tools\adb shell "chmod 777 /data/ch13/hellodynamic"
```

This build script passes the `-c` compiler option when compiling the source file, hello.c. This way, gcc doesn't attempt to link the application. The link command, `arm-none-linux-gnueeabi-ld`, has a number of options. These options are more fully



```
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 libFFTEm.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 libpim.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 libpvdownload.so
07/29/2008  12:05 AM            13,068 libpvdownloadreg.so
07/29/2008  12:05 AM           589,180 libpvnet_support.so
07/29/2008  12:05 AM           705,712 libpvrtsp.so
07/29/2008  12:05 AM            13,188 libpvrtspreg.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             6,936 libsystem_server.so
07/29/2008  12:04 AM             9,952 libthread_db.so
07/29/2008  12:05 AM           771,672 libUAPI_jni.so
07/29/2008  12:05 AM           109,504 libui.so
07/29/2008  12:05 AM           379,744 libutils.so
07/29/2008  12:04 AM           122,032 libvorbisidec.so
07/29/2008  12:05 AM         3,908,228 libwebcore.so
07/29/2008  12:04 AM            78,452 libz.so
              52 File(s)     20,335,784 bytes
               2 Dir(s)     441,020,416 bytes free

M:\android\system\lib>_
```

Figure 13.4 Android libraries pulled to the development machine

described in table 13.1. As in the previous example, adb is used to push the executable file over to the Android Emulator. The permissions are also modified to mark the application as executable.

If our application required routines from the Open GL or SQLite library, the `link` command would have additional parameters of `-l GLES_CM` or `-l sqlite`, respectively. Leaving those library options off the `link` command prevents the application from linking properly because certain symbols (functions, data) can't be found.

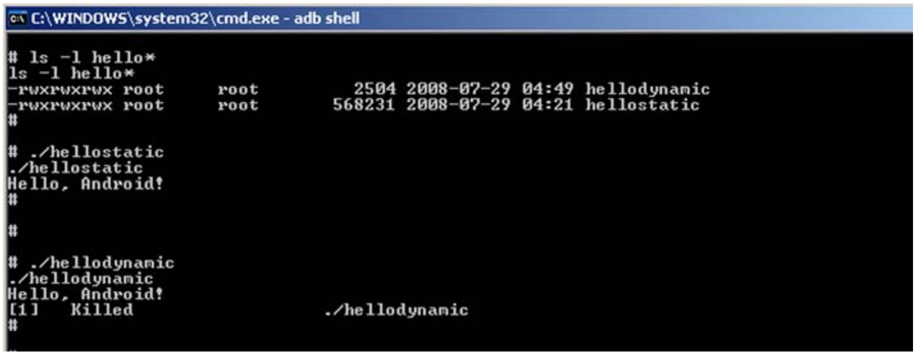
So, did it work? The hellodynamic binary is now only 2504 bytes. That's a great improvement. Figure 13.5 shows a listing of the two Hello Android files for a

Table 13.1 Linker options

Linker option	Description
`-entry=main`	Indicates the entry point for the application, in this case, the function named `main`.
`-dynamic-linker /system/bin/linker`	Tells the application where the dynamic linker application may be found at runtime. The /system/bin/linker path is found on the Android Emulator, not the development environment.
`-nostdlib`	Tells the linker to not include standard C libraries when attempting to resolve code during the link process.
`-rpath /system/lib`	Tells the executable where libraries can be found at runtime. This works in a manner similar to the environment variable `LD_LIBRARY_PATH`.
`-rpath-link /android/system/lib`	Tells the linker where libraries can be found when linking. For Linux add a dot to the beginning of the line, as in ./android/system/lib.
`-L /android/system/lib`	Tells the linker where libraries can be found. This is the linker import directory.
`-l android_runtime`	Tells the linker that this application requires routines found in the library file libandroid_runtime.so.
`-l c`	Tells the linker that this application requires routines found in the library file libc.so.
`-o hellodynamic`	Requests an output filename of hellodynamic.
`hello.o`	Includes hello.o as an input to the link process.

remarkable comparison. Each program is run: first the static version, then the dynamic version.

This looks great, except for one little problem. Note the last line in figure 13.5, which says, “Killed.” Is there a problem with our dynamic version? Let’s look closer.



```
# 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

13.2.3 exit() versus return()

Though our application has successfully linked with the Android system libraries of libc.so and libandroid_runtime.so and can actually run, there are missing pieces that cause the application to not properly execute. When you build an application in this manner, without letting the linker do all its magic of knitting the entire application together, you have to do a bit of housekeeping yourself. Looks like there was something to that 500 KB 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 does it return to? Let's replace the `return` statement with an `exit()` call, as shown in this listing.

Listing 13.5 Add an exit() call

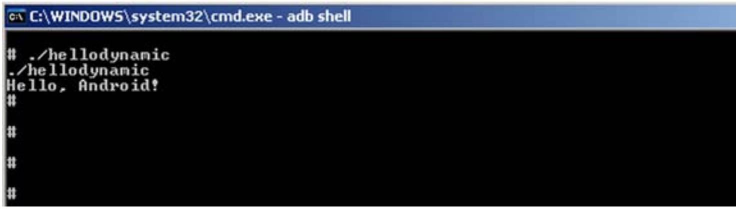
```
#include <stdio.h>
int main(int argc,char * argv[])
{
      printf("Hello, Android!\n");
      exit(0);
      //return 0;
}
```

Add a call to the function `exit()`. This should return execution to the OS. Comment out the call to `return()`. A `return` call in this location causes a stack underflow because there's nowhere within this application to return to!

This fixed the problem—no more killed messages! Look at figure 13.6, where you see that the dynamic version of Hello, Android now runs just fine.

Unfortunately you're not finished. It turns out that the application doesn't properly interact with other libraries, nor does it properly handle the `argc` and `argv[]` arguments to the `main` function. The C library (remember, you're linking against libc.so) has certain expectations for application structure and stack location. You're closer but still not quite ready for prime time.

What this application requires is a start routine, which is called by the operating system when the application is invoked. This function in turn calls the application's `main` function. This start routine must set up the necessary structures to allow the application to properly interact with the operating system and the core C libraries.



```
# ./hellodynamic
./hellodynamic
Hello, Android!
#

#

#

#
```

Figure 13.6 A better-behaving dynamic version of Hello Android

13.2.4 Startup code

You've surmised that the sample application is missing the proper startup code, but just what does startup code for an Android/Linux application on ARM look like? Where do you turn to get this kind of information? Let's look deeper into the bag of CodeSourcery tricks for a clue.

A number of executable applications ship with Android, so pull one of them over to the desktop and see what you can learn. Perhaps you can extract information from that file that can assist in solving this puzzle.

The tool you're going to use to assist 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 (executable and linkable format)* file for examination. This is the kind of file structure used by applications in the Android/Linux environment. Using the `-d` option of 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. The following listing shows 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     mov r0, sp                   ❶ Stack pointer
    96d4:     e3a01000     mov r1, #0; 0x0              ❷ mov instruction
    96d8:     e28f2004     add r2, pc, #4; 0x4          ❸ add instruction
    96dc:     e28f3004     add r3, pc, #4; 0x43
    96e0:     eaffff8b     b   9514 <dlopen-0x21c>      ❹ Branch instruction
    96e4:     ea000e03     b   cef8 <dlclose+0x37bc>    ❺ Branch instruction
    96e8:     0000e408     andeq lr, r0, r8, lsl #8     ❻ Conditional expressions
    96ec:     0000e410     andeq lr, r0, r0, lsl r4
    96f0:     0000e418     andeq lr, r0, r8, lsl r4
    96f4:     0000e420     andeq lr, r0, r0, lsr #8
    96f8:     e1a00000     nop (mov r0,r0)              ❼ nop instruction
    96fc:     e1a00000     nop (mov r0,r0)
```

The first instruction assigns the value of the stack pointer (`sp`) to register 0 (r0) ❶. Next the literal value of 0 is assigned to register r1 ❷. The address counter plus four memory location spaces is stored in registers r2 and r3 ❸. The `b` instruction tells the code to branch to a specific address ❹. In this case, the address is 0x21c bytes prior to the address of the `dlopen` function. This value is 9514 in decimal. The next branch is to an address that's 0x37bc bytes beyond the `dlclose` label ❺. The next few instructions ❻ are conditional operations. The code snippet finishes up with a pair of `nop` instructions ❼. Note that the address of each instruction is shown to the left of each line. Each instruction occurs at a 4 byte offset from its predecessor. Four bytes times 6 bits per byte equals a 32-bit address bus, which makes sense because the ARM processor family is 32 bit.

Okay, so that looks 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's little to tell you why those instructions are there. After doing research on the internet, we found a better example of this code, shown in this listing.

Listing 13.7 crt.S

```
        .text                          ❶ .text directive
        .global _start                                  ❷ global directive
_start:                                ❸ start label
        mov     r0, sp                                   ❹ Set up
        mov     r1, #0                                     stack pointer
        add     r2, pc,
        add     r3, pc,                         ❺ Branch to
        b       __libc_init                       initialization
        b       main                                     ❻ Branch
        .word   __preinit_array_start                      to main
        .word   __init_array_start              ❼ Jump table
        .word   __fini_array_start
        .word   __ctors_start
        .word   0
        .word   0
        .section .preinit_array
__preinit_array_start:                          ❽ Required
        .word   0xffffffff                        sections
        .word   0x00000000
        .section .init_array
__init_array_start:
        .word   0xffffffff
        .word   0x00000000
        .section .fini_array
__fini_array_start:
        .word   0xffffffff
        .word   0x00000000
        .section .ctors
__ctors_start:
        .word   0xffffffff
        .word   0x00000000
```

The `.text` directive indicates that this code should be placed in the `.text` section of the resulting executable ❶. The `global start` directive ❷ makes the start routine visible to the rest of the application and the linker. The `start`: label ❸ indicates the first location of the start routine. The `mov` and `add` instructions perform some housekeeping ❹ with the stack pointer, `sp`, just as seen in the extracted code from the ping program. Initialization takes place via a branch instruction to call the `__libc_init` routine ❺. This routine is found in the library libc.so. When this routine is complete, execution returns to the next instruction, another branch of the main routine ❻. This is the `main()` routine implemented by our C application. The next instructions ❼ set up a jump table to the sections required by a C language executable application. A pair of `nop` instructions round out the table. The sections `preinit_array`, `init_array`, `fini_array`, and `.ctors` are defined ❽. 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 code that crashes.

> **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. You can find additional reference material for low-level Android programming information at http://benno.id.au.

Now that you've found an adequate startup routine, we'll show you how to add this routine to your application. The compiler handles the assembly file just like a C language file:

```
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 /android/system/lib -rpath-link /android/system/lib -L
     \android\system\lib -l c -l android_runtime -l sqlite -o hellodynamic
     hello.o crt0.o
```

At this point, you should feel confident that you can build applications for Android/Linux, so it's time to build something useful. The next section walks through the construction of a DayTime Server.

13.3 What time is it? The DayTime Server

Although we don't talk about it much today, Linux systems (and more generically, Unix systems) have a service running that provides the server's current date and time. This application, known as a DayTime Server, typically runs as a daemon (which means it runs in the background and isn't connected to a particular shell). For our purposes, we'll implement a basic DayTime Server for Android/Linux, but we won't worry about turning it into a background service.

This application helps exercise our interest in developing Android/Linux applications. First and most important, it's an application of some significance beyond a simple `printf` statement. Second, once this application is built, you'll write an Android/Java application to interact with the DayTime Server.

13.3.1 DayTime Server application

Our DayTime Server application has a 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 returns to listening for a new connection.

In addition to the TCP socket interactions, our application logs requests to a SQLite database. Why? Because we can! The purpose of this application is to demonstrate nontrivial activities in the Android/Linux environment, including the use of the SQLite system library. Let's get started by 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 Server application binds to TCP port 1024 when looking for new connections. Ordinarily, a daytime service binds to TCP port 13, but 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. The following listing shows the code for the DayTime Server application.

Listing 13.8 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 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);
}
```

1 Importing required headers

2 Listening port number

3 Defining helpful macro

4 Interacting with SQLite

```
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);
```

5 Setting up & listening on socket

Setting up & listening on socket

6 Accepting socket connection

```
    sprintf(buf,"%.24s",ctime(&ticks));
    printf("sending [%s]\n",buf);
    write(connfd,buf,strlen(buf));
    close(connfd);
    RecordHit(buf);
  }
  return 0;
}
```

❼ Recording activity

As with many C language applications, a number of headers ❶ are required, including definitions and prototypes for time functions, SQLite functions, and for TCP sockets. Note that the sqlite3.h header file isn't provided in the CodeSourcery tool chain. This file was acquired from a sqlite3 distribution, and the file was 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. The `htons` function is typically implemented in the library named socket (libsocket.so). Android doesn't provide this library, nor was this found in any of the system libraries. Therefore `htons` is defined here as a macro ❸. This macro is required to get the network byte ordering correct. When the application is running, you can verify this port by running `netstat -tcp` on the command line in the `adb` shell.

The standard TCP port for a DayTime Server is port 13. In ❷, the application is using port 1024 because our application can't bind to any port numbered 1023 or below. Only system processes may bind to ports below 1024.

In the `RecordHit` function, you see SQLite interaction ❹. The `RecordHit()` function is responsible for inserting a record into the SQLite database created for this application.

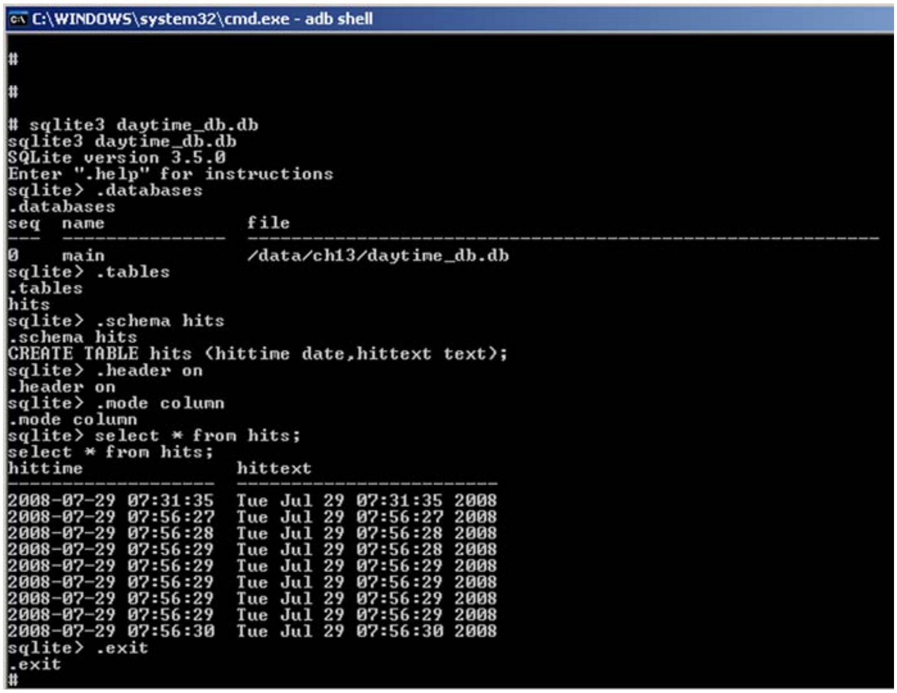
Jumping into the `main` function, you see the socket functions in use to listen on a socket for incoming connections ❺. When a connection is accepted ❻, the current system time is sent to the calling client. After this, the application makes a record of the transaction by calling the `RecordHit` function ❼.

That's all the code necessary to implement our Android/Linux DayTime Server application. Let's look next at the SQLite 3 database interaction in more detail.

13.3.3 The SQLite database

This application employs a simple database structure created with the SQLite 3 application. We interact with SQLite 3 from the adb shell environment, as shown in figure 13.7.

The purpose of this database is to record data each time the DayTime Server processes an incoming request. From a data perspective, this sample is boring, as it simply records the system time along with the text returned to the client (this text 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.



```
#
#
# 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 07:56:30  Tue Jul 29 07:56:30 2008
sqlite> .exit
.exit
#
```

Figure 13.7 Interact with SQLite 3 from the command line in the adb shell.

The previous section of code outlined the syntax for inserting a row into the database; this section shows how to interact with the database using the SQLite 3 tool. The sequence shown in figure 13.7 is broken out and explained in the following listing.

Listing 13.9 Interacting with a SQLite database

```
# 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
```

1 Connect to database file

2 Examine database structure

3 Create statement

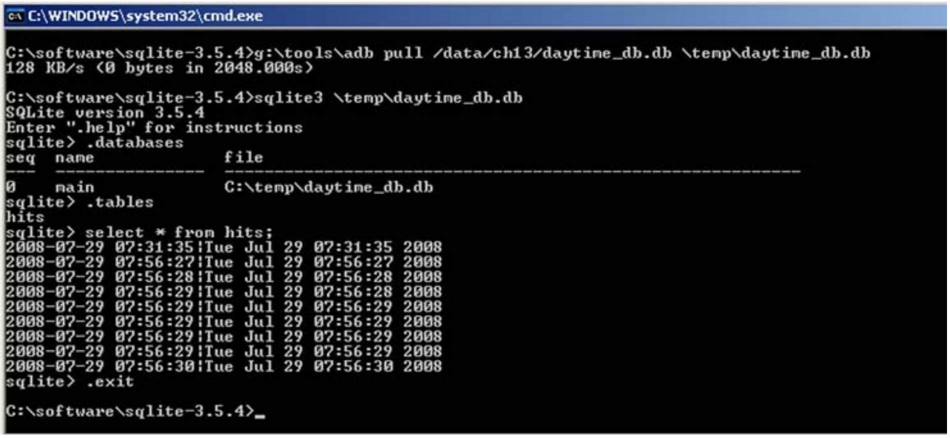
```
sqlite> .mode column
.mode column
sqlite> select * from hits;              ❹ Select rows
select * from hits;
hittime              hittext
-------------------  ------------------------
2008-07-29 07:31:35  Tue Jul 29 07:31:35 2008
2008-07-29 07:56:27  Tue Jul 29 07:56:27 2008
2008-07-29 07:56:28  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
#
```

The SQLite database operates in a similar fashion to other, modern SQL-based environments. In listing 13.9, you see the output from an interactive session where the database for this chapter's sample application is opened ❶. A series of commands given at the `sqlite>` prompt ❷ display the contents of the database in terms of structure. The `schema` command dumps the DDL (Data Definition Language) for a particular table. In this case, you see the `CREATE TABLE` instructions for the hits table ❸. Viewing the data is simple with the use of the familiar `select` statement ❹.

To run the sample code yourself, you'll want to execute the following command sequence from an adb shell:

```
cd /data/ch13
sqlite3 daytime_db.db
create table hits (hittime date,hittext text);
.exit
```

The SQLite database engine is known for its simplicity. This section displayed a simple interaction and just how easy it is to employ. In addition, the SQLite 3 database may be pulled from the Android Emulator and used on the development machine, as shown in figure 13.8.



```
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
hits
sqlite> select * from hits;
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 07:56:30|Tue Jul 29 07:56:30 2008
sqlite> .exit

C:\software\sqlite-3.5.4>_
```

Figure 13.8 The SQLite database on the development machine

This feature makes Android a compelling platform for mobile data collection applications because syncing data can be as simple as copying a database file that's compatible across multiple platforms.

13.3.4 Building and running the DayTime Server

To build this application, you need to combine the components of the previous few sections. You know that our application requires a startup component and must also link against multiple libraries. Because the application interacts with the SQLite database, you must link against the sqlite library in addition to the c and android_runtime libraries. The full build script is shown in the next listing.

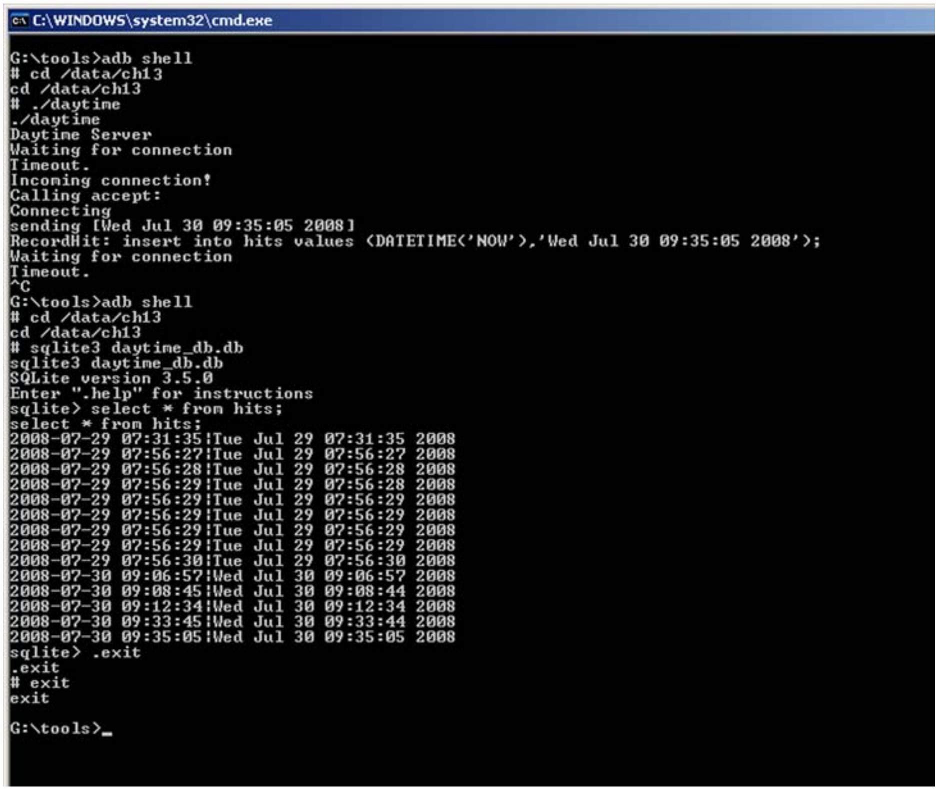
Listing 13.10 Daytime application build script

```
arm-none-linux-gnueabi-gcc -c daytime.c
arm-none-linux-gnueabi-gcc -c -o crt0.o crt.S
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
C:\software\google\<path to android sdk>\tools\adb
               push daytime /data/ch13
g:\tools\adb shell "chmod 777 /data/ch13/daytime"
```

The build script begins by compiling the main source file, daytime.c. The next line compiles the crt.S file, which we introduced in listing 13.7 for our C runtime initialization. The `linker` command contains a number of switches to create the desired application. Note the parameter to the linker to include 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. The input files are found in local (to the development machine) copies of the libraries. adb is employed to push the executable file to the Android Emulator and to modify the permissions, saving a manual step.

Running the DayTime Server application is the easy and fun part of this exercise. Here's a rundown of the sequence shown in figure 13.9:

1. Start the shell by running `adb shell`.
2. Change directories to /data/ch13, where the application resides, previously pushed there with an `adb push` command.
3. Run the ./daytime application.
4. The application binds to a port and begins listening for an incoming connection.
5. A timeout occurs prior to a connection being made. The application displays the timeout and returns to look for connections again.
6. A connection is detected and subsequently accepted.



```
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
Timeout.
^C
G:\tools>adb shell
# cd /data/ch13
cd /data/ch13
# sqlite3 daytime_db.db
sqlite3 daytime_db.db
SQLite version 3.5.0
Enter ".help" for instructions
sqlite> select * from hits;
select * from hits;
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 07:56:30|Tue Jul 29 07:56:30 2008
2008-07-30 09:06:57|Wed Jul 30 09:06:57 2008
2008-07-30 09:08:45|Wed Jul 30 09:08:44 2008
2008-07-30 09:12:34|Wed Jul 30 09:12:34 2008
2008-07-30 09:33:45|Wed Jul 30 09:33:44 2008
2008-07-30 09:35:05|Wed Jul 30 09:35:05 2008
sqlite> .exit
.exit
# exit
exit

G:\tools>_
```

Figure 13.9 DayTime Server running in the shell

7 The `time` string is constructed and sent to the client.

8 A record is inserted into the database with the shown `sql` statement.

9 You kill the application and restart the shell. Note that this is because you didn't 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 discussion here.

10 Run sqlite3 to examine the contents of our application's database.

11 Perform a `select` against the hits table, where you see the recently inserted record.

You've built an Android/Linux application that 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.

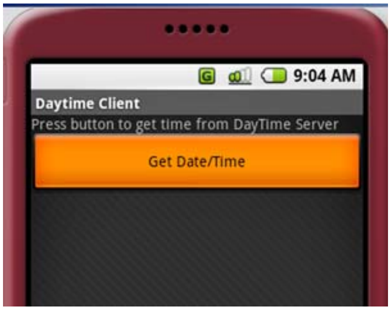
13.4 Daytime Client

One of the stated objectives for this chapter is to connect the Java UI 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.

When a user clicks the Button, the Activity initiates the DayTime Server query and replaces the text of the TextView with the information received from the DayTime Server. There's not much to it, but that's fine, as all we're after in this sample is to demonstrate connectivity between the two applications. The following listing shows the onCreate method for this Activity.



Daytime Client

Press button to get time from DayTime Server

Figure 13.10 The Daytime Client app

Listing 13.11 UI elements of DaytimeClient.java

```
Handler h;
@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.main);
    final TextView statuslabel = (TextView)
 findViewById(R.id.statuslabel);
    h = new Handler() {
        @Override
        public void handleMessage(Message msg) {
             switch (msg.what) {
                case 0:
                    Log.d("CH13","data [" + (String) msg.obj + "]");
                    statuslabel.setText((String) msg.obj);
                    break;
             }
             super.handleMessage(msg);
        }
    };
    Button test = (Button) findViewById(R.id.testit);
    test.setOnClickListener(new OnClickListener() {
        public void onClick(View v) {
             try {
                  Requester r = new Requester();
                  r.start();
             } catch (Exception e) {
                  Log.d("CH13 exception caught : ",e.getMessage());
             }
        }
    });
}
```

1 Declare, implement Handler

2 Implement click listener

3 Create Requester instance

This application is all about detecting the selection of a button ❷ and initiating an action based on that click. The action is the creation of an instance of the `Requester` class ❸, which we discuss in the next section. You handle the response from the socket server with the assistance of a `Handler` ❶. The `Handler` has a single role: updating the UI with textual data stored in the `obj` member of a `Message` object.

Although the UI of this application is simple, the more interesting side of this `Activity` is the interaction with the DayTime Server, which takes place in the `Requester` class, which we'll look at next.

13.4.2 Socket Client

The DayTime Server application listens on a TCP port for incoming connections. To request the date and time, the Daytime Client must establish a client socket connection to the DayTime Server. It's hard to imagine a simpler TCP service than this—open a socket to the server and read data until the socket connection is closed. There's no additional requirement. Most of the networking examples in this book have focused on a higher-level protocol, HTTP, where the request and response are 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's no protocol associated with it beyond being a TCP stream (as opposed to UDP). The following listing demonstrates this lower-level socket communication.

Listing 13.12 Requester class implementation

```
    public class Requester extends Thread {
    Socket requestSocket;
    String message;
    StringBuilder returnStringBuffer = new StringBuilder();
    Message lmsg;
    int ch;
    public void run() {
         try {
              requestSocket = new Socket("localhost", 1024);
                InputStreamReader isr = new
     InputStreamReader(requestSocket.getInputStream(),
"ISO-8859-1");
                while ((ch = isr.read()) != -1) {
                    returnStringBuffer.append((char) ch);
                }
                message = returnStringBuffer.toString();
                lmsg = new Message();
                lmsg.obj = (Object) message;
                lmsg.what = 0;
                h.sendMessage(lmsg);
                requestSocket.close();
             } catch (Exception ee) {
                Log.d("CH13","failed to read data" + ee.getMessage());
         }
    }
    }
```

❶ Extending Thread class

❷ Communicating on Socket

❸ Creating Message object

❹ Sending Message to main thread

The Requestor ❶ class extends the Thread class by implementing the run method. Communications take place via an instance of the Socket class ❷, which is found in the java.net package. Note the port number being used—1024, just like our socket server! A Message ❸ is used to communicate back to the UI thread. Once the Message object is initialized, it's sent back to the calling thread ❹.

With the Daytime Client now coded, it's 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, you can run the Daytime Client.

> **NOTE** If you're unclear on how to build and run the Daytime Client, refer to chapter 2 for information on properly setting up the Android development environment in Eclipse.

Figure 13.11 demonstrates the Daytime Client running, alongside 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.

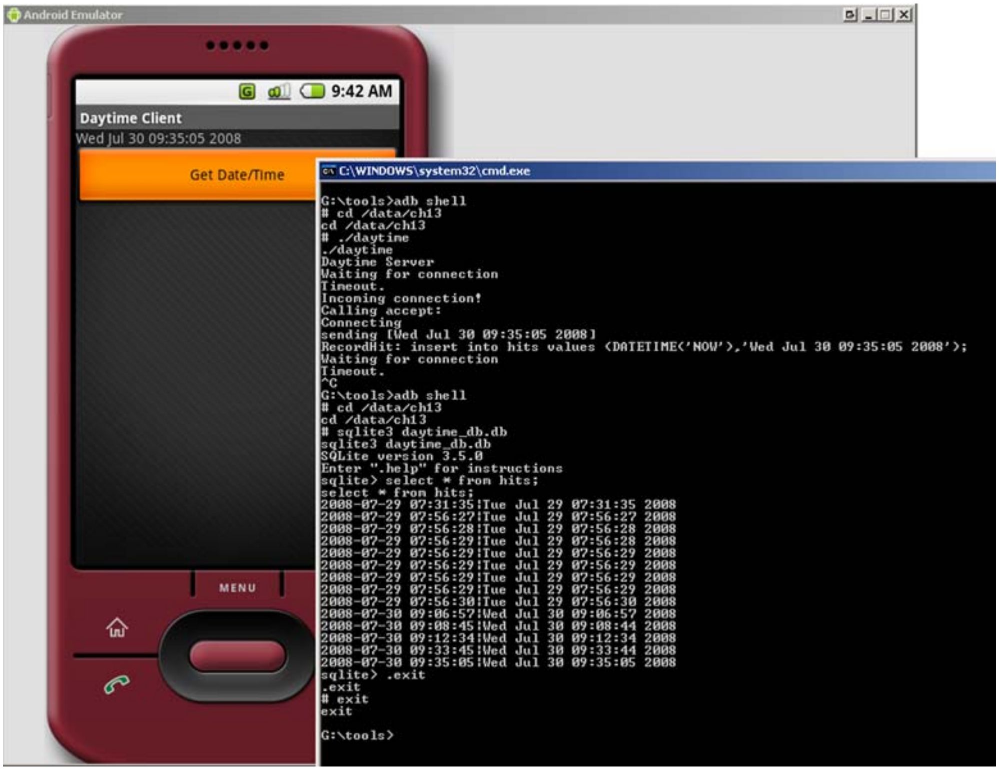
The DayTime Server is exercising both TCP socket functionality and SQLite database record insertions, all running in the Android Emulator. A production-ready Android/Linux application would need to be converted to run as a daemon, which is beyond our aim for this chapter.

13.5 Summary

This chapter hopefully stretched 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 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 CodeSourcery tool chain. This chapter demonstrated using that toolset in conjunction with the adb utility provided in the Android SDK. The adb utility enabled you to push the application to the Android Emulator for testing, as well as extract the Android system libraries essential for linking the application with the Android resident libraries. You used the adb shell to interact directly with the Android Emulator to run the C application.

Our sample application exercised TCP socket communications. The TCP 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



```
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
Timeout.
^C
G:\tools>adb shell
# cd /data/ch13
cd /data/ch13
# sqlite3 daytime_db.db
sqlite3 daytime_db.db
SQLite version 3.5.0
Enter ".help" for instructions
sqlite> select * from hits;
select * from hits;
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 07:56:30|Tue Jul 29 07:56:30 2008
2008-07-30 09:06:57|Wed Jul 30 09:06:57 2008
2008-07-30 09:08:45|Wed Jul 30 09:08:44 2008
2008-07-30 09:12:34|Wed Jul 30 09:12:34 2008
2008-07-30 09:33:45|Wed Jul 30 09:33:44 2008
2008-07-30 09:35:05|Wed Jul 30 09:35:05 2008
sqlite> .exit
.exit
# exit
exit

G:\tools>
```

Figure 13.11 Testing the Daytime Client

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 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 shouldn't be minimized, as it satisfies three important development challenges. The first and most basic accomplishment of this functionality is that we've 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. Second, using a device-resident database that's also accessible from the Java layer means you have an additional (and persistent) interface mechanism between the Java and Linux environments on the platform. Third, Android is a mobile platform. Anytime there's a mobile application, the topic of sharing and syncing data bubbles up. 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 a broad topic, 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's after years of market penetration by these other platforms. Android gets it right from the start.

This chapter took a bit of a detour from the regular Android SDK programming environment. It's time to return to the SDK; in the next chapter you'll learn about Bluetooth and sensors.

Part 4

The maturing platform

In part 3, you learned about two extremes of the Android platform. Chapter 12 provided the full end-to-end SDK application experience, and chapter 13 went to the other extreme of exploring application techniques that might best fit a custom piece of hardware running the Android operating system. The objective of part 4 is to explore some of the features added to the Android platform that take it a step beyond the other platforms to provide a unique and memorable mobile experience.

In chapter 14, we get close to the metal by interrogating onboard sensors and communicating over Bluetooth. The sensors act as inputs for a navigation system to control a LEGO Mindstorms robot.

In chapter 15, we build a sophisticated integration between the Android contact database and business social networking sensation, LinkedIn. The application constructed in chapter 15 has become a popular download on the Android market. Read along and learn how to get up close and personal with your contacts.

In chapter 16, the topic of Android web development is explored. Topics such as building websites for the WebKit-powered Android browser and custom JavaScript handlers are introduced. In addition, local SQL-based storage concepts are examined, enabling next-generation web applications directly on your mobile device.

Chapter 17 presents a nontrivial example of the AppWidget, tying together other key concepts such as services, alarms, and `BroadcastReceivers`. There's something for everyone in chapter 17 as we construct a website monitoring tool that provides near-real-time status information directly to the home page of your Android device.

Chapter 18 circles back to the application constructed in chapter 12, but with a twist. The code in chapter 18 demonstrates the localization of an existing application as the field service application presented in chapter 12 is modified to support multiple languages. The application now supports English and Spanish, depending on the locale of the device.

Chapter 19 wraps up the book with a look at the Native Development Kit (NDK). The NDK permits Android developers to incorporate C language source code into SDK applications. Chapter 19 demonstrates the NDK in the context of an image-processing application that allows the user to capture images with the built-in camera and then perform an edge detection algorithm against the image. It's loads of fun and you'll learn about the Java Native Interface as well as how to integrate the NDK build process directly into Eclipse.

14 Bluetooth and sensors

This chapter covers

- Connecting to a Bluetooth peripheral
- Interacting with the SensorManager
- Building and running the SenseBot application

The majority of the material presented in this book is concerned with employing various capabilities of the Android SDK. At this point, however, you're going to see how to take advantage of an Android device's hardware. Specifically, we'll look at connecting an Android device to remote devices via a Bluetooth wireless connection, as well as reading and interpreting values from a hardware-based orientation sensor. This chapter combines these two hardware-related topics in a sample program that exercises control over a robot constructed from the popular LEGO Mindstorms NXT. The Mindstorms NXT robot supports a communications protocol known as "Direct Commands,"[1] allowing it to be controlled by a remote device. This is the one chapter of the book where you'll want to have access to a physical Android device with version 2 or later of the operating system—the simulator alone isn't adequate for exercising the Bluetooth and sensor functionality.

[1] To learn more about Direct Commands for the Lego Mindstorm, start here: http://mindstorms.lego.com/en-us/support/files/default.aspx.

The code accompanying this chapter is organized into an Android application named SenseBot. SenseBot is a moderately complex example of using Android to manipulate an external object. Android's orientation sensor permits the user to "drive" the robot by simply holding the phone in a particular direction, not unlike a Nintendo Wii or other advanced gaming system. Tilt the phone forward and the robot drives forward. Tilt it backward and the robot reverses direction. Tilting to the left or right causes the robot to spin in the respective direction. With each interpreted sensor motion, the SenseBot application uses Bluetooth to send commands to the robot causing the appropriate physical behavior. The LEGO NXT comes equipped with a built-in command set that permits low-level operations such as direct motor control. The motions of the Android device are interpreted, converted to commands, and transmitted via Bluetooth to the robot.

In addition to basic Bluetooth communications and Sensor management, the code demonstrates the use of a dynamically created BroadcastReceiver employed to handle Bluetooth-related connection events.

The topic of Bluetooth communications is much broader and deeper than we can hope to cover in a single chapter. Likewise, there are at least half a dozen hardware sensors available on the Android platform, yet this chapter demonstrates the use of only one. If you're looking for textbook-like coverage of these two topics, we encourage you to look at the online documentation or perhaps another text on the subject. The aim of this chapter is to explore Bluetooth and sensor functionality on the Android platform in the context of a functional (and fun) application. If you take the time to follow along and build this application and have access to a LEGO Mindstorms NXT robot, I promise that you'll get hooked on "driving" your robot with your phone. Also, a version of the application is available for download from the Android market.

14.1 Exploring Android's Bluetooth capabilities

The first thing that comes to mind with the term *Bluetooth* is wireless headsets. Also known as a *hands-free*, in many parts of the world these wireless wonders are required by law for operating your telephone while driving a vehicle. In actuality, the hands-free device is only one of many uses for the versatile Bluetooth technology.

Bluetooth is a wireless communications protocol similar to WiFi but constrained to usage scenarios for short-range applications reaching a range of approximately 10 meters. In addition to providing functionality as a hands-free microphone and speaker for your cell phone, Bluetooth also enables peer-to-peer network access, object exchange, cable replacement, and advanced audio/media capabilities.

Like any other protocol standard, Bluetooth has its own "stack" of layers, each of which implements distinct capabilities and features of the protocol. This chapter doesn't spend time dissecting these layers, as the Bluetooth stack is well covered in other places. Rather, this chapter demonstrates the approach for establishing a data connection between two peers. The specific Bluetooth "profile" employed here is the RFCOMM[2] cable replacement profile.

In this section you'll learn how to establish a connection between Android and your remote device via the `android.bluetooth` package. Given how the Android platform permits only encrypted connections, your two communicating devices must first be paired or bonded, which will subsequently allow you to connect without a further confirmation or security prompt. Then, in order to know that you've connected to a Bluetooth device, you must register for two events: `ACTION_ACL_CONNECTED` and `ACTION_ACL_DISCONNECTED`. And finally, your Android application will need to have `BLUETOOTH` permission as defined in the AndroidManifest.xml file. Let's get started.

14.1.1 Replacing cables

Today, connecting to the internet to exchange emails or browse the web is an everyday experience for most Android users. With your phone you can connect to computers on the other side of the planet and beyond, but how can you communicate with something in the same room? In the not-so-distant past we programmed interfaces between computers and peripherals across a serial cable, often described as an *RS232* interface. In a few short years, the RS232 serial cable has become a museum piece, having been replaced by the more capable USB and with the Bluetooth Serial Port Profile.

In the same way that USB can be used for many different applications, the Bluetooth wireless protocol also may be deployed in a variety of manners. The Bluetooth capability of interest to us is the cable replacement functionality of the Serial Port Profile (SPP), which is sometimes referred to as *RFCOMM*. The *RF* stands for radio frequency, aka "wireless." The *COMM* stands for communications port, harkening back to its roots as a point-to-point connection-based streaming protocol.

14.1.2 Primary and secondary roles and sockets

The Bluetooth protocol works in a fashion similar to other communications environments where there's a primary (or master) device that initiates communications with one or more secondary (or slave) devices. Android is versatile in that it may be either a primary or a secondary device in a Bluetooth connection.

Regardless of how a connection is established—as a primary or a secondary Bluetooth device—an Android application exchanges data through a *socket interface*. That's right; the familiar networking paradigm of a socket and its associated input stream and output stream is employed for Bluetooth connectivity as well. So once you get past the scaffolding of connecting two Bluetooth devices together in a communications session, you can be less concerned with the underlying details and can simply view the remote device as an application on the other side of a socket. This is much like the relationship between a web browser and a remote server that exchange data over a TCP socket.

[2] To learn more about RFCOMM, look at http://www.bluetooth.com.

To access the Bluetooth environment on an Android device, you need to dig into the android.bluetooth package, which first appeared in Android version 2.0. Though most Android devices prior to version 2 were capable of Bluetooth hands-free operation, it wasn't until version 2 that Android applications could leverage the underlying Bluetooth hardware as discussed in this chapter. Table 14.1 shows the major Java classes used by Bluetooth-enabled Android applications.

Table 14.1 Bluetooth classes

Class	Comment
BluetoothAdapter	This class represents the local Android device's Bluetooth hardware and interface constructs. Everything begins with the BluetoothAdapter.
BluetoothClass	The BluetoothClass provides a convenient means of accessing constant values related to Bluetooth communications and operations.
BluetoothDevice	Any remote device is represented as a BluetoothDevice.
BluetoothSocket	The BluetoothSocket is used for exchanging data. On a more practical note, a primary device initiates a socket connection with a secondary device by first creating a BluetoothSocket. The example code in this chapter demonstrates this technique.
BluetoothServerSocket	A Bluetooth secondary device listens for a primary device to connect through a BluetoothServerSocket in much the same way that a web server awaits a TCP socket connection from a browser. Once connected, a BluetoothSocket is established for the ongoing communication.

This chapter demonstrates the use of the BluetoothAdapter, the BluetoothDevice class, and the BluetoothSocket. The next section shows how an Android device goes about connecting to another Bluetooth-enabled device.

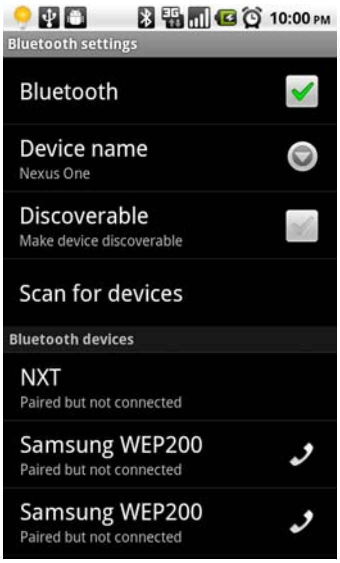
> **NOTE** For the examples in this chapter, the Android device acts as the primary device and a LEGO Mindstorms NXT controller acts as a secondary Bluetooth device.

14.1.3 Trusting a device

Although the broader Bluetooth specification allows for both encrypted and unencrypted communications between peer devices, the Android platform permits only encrypted connections. In essence, this means that the two communicating devices must first be *paired*, or *bonded*. This is the somewhat annoying step of telling each device that the other is trusted. Despite the annoyance factor and the fact that virtually every Bluetooth device on the planet uses its default security pin code of 0000 or 1234, the security aspects of Bluetooth do have their value—sort of.

Devices are paired either through the "settings" screens of the various peers or on demand the first time a connection is requested. This section walks through the steps of pairing an Android device[3] with a LEGO robot controller module.

Figure 14.1 shows a portion of the Bluetooth settings screen from my Nexus One device running Android 2.2.



Bluetooth settings

Bluetooth ☑

Device name
Nexus One

Discoverable ☐
Make device discoverable

Scan for devices

Bluetooth devices

NXT
Paired but not connected

Samsung WEP200
Paired but not connected

Samsung WEP200
Paired but not connected

Figure 14.1 Bluetooth settings screen

From this screen you can see that the following is true:

- Bluetooth is enabled.
- This device name is Nexus One.
- This device isn't currently discoverable. This means that other Bluetooth devices won't see this phone during a "scan." Practically speaking, this means that the phone ignores discovery packets that it detects. There's a button used to initiate a manual scan for nearby Bluetooth devices.
- You can initiate a scan for nearby Bluetooth devices by pressing the Scan for Devices button.
- There are three devices that this phone has previously paired with but aren't currently connected:
  - NXT—the LEGO robot
  - Two instances of a Samsung hands-free device. This isn't a mistake—there are two distinct devices paired with this phone. (This author "solved" his problem of frequently lost hands-free devices by buying a handful of them via eBay, hence the multiple device pairings!)

A long click on one of the entries in the Bluetooth devices list presents options for further operations, with the specific choices depending on the device. For example, selecting one of the Samsung entries presents the options shown in figure 14.2.

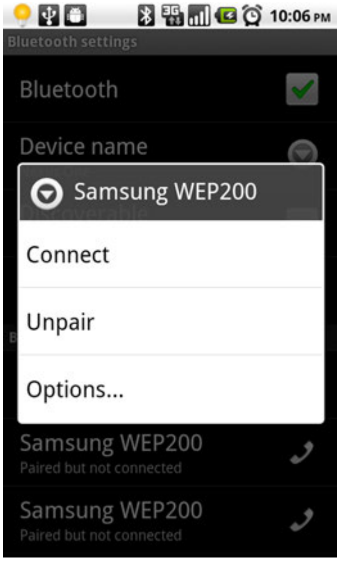


Figure 14.2 Options for a paired device

[3] Join the Talk Android forums to learn more about the types of Android hardware: http://www.talkandroid.com/android-forums/android-hardware/.

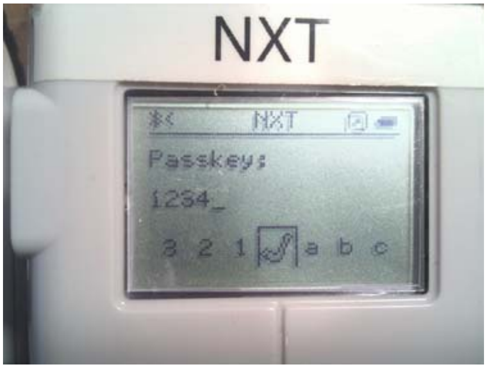
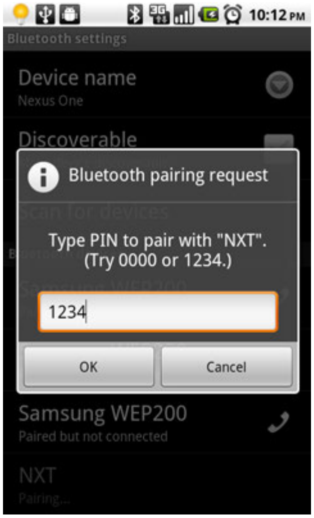


Figure 14.3 LEGO controller prompts for a PIN


10:12 PM

Bluetooth settings

Device name
Nexus One

Discoverable

Bluetooth pairing request

Type PIN to pair with "NXT".
(Try 0000 or 1234.)

1234

OK Cancel

Samsung WEP200
Paired but not connected

NXT
Pairing...

Figure 14.4 Pairing with the LEGO robot

In order to pair with a device, you need to first scan for it. Once it's been added to the list, you can select it to initiate the pairing. Figure 14.3 shows the LEGO robot controller prompting for a PIN after a pairing request.

This PIN value will then be compared to what the user enters on the phone, as shown in figure 14.4.

At this point, your phone and the LEGO robot controller are paired. Moving forward, you'll be able to connect to this device without a further confirmation or security prompt.

14.1.4 Connecting to a remote device

Connecting to a paired, or bonded, device involves a two-step process:

- Get a list of paired devices from the Bluetooth hardware/software stack.
- Initiate an RFCOMM connection to the target device. The following listing demonstrates a basic approach to establishing an RFCOMM, or Serial Port Profile connection, between paired devices.

Listing 14.1 Initiating a connection to a BluetoothDevice

```
public void findRobot(View v)
{
    try
    {
      btInterface = BluetoothAdapter.getDefaultAdapter();
      pairedDevices = btInterface.getBondedDevices();
      Iterator<BluetoothDevice> it = pairedDevices.iterator();
      while (it.hasNext())
      {
```

1 Get adapter

2 Get list of devices

3 Enumerate list

```
        BluetoothDevice bd = it.next();                    ❹ Evaluate
        if (bd.getName().equalsIgnoreCase(ROBOTNAME)) {      device name
          connectToRobot(bd);                      ❺ Connect to Robot
          return;
        }
      }
    }
    catch (Exception e)                                    ❻ Handle
    {                                                        connection
      Log.e(tag,"Failed in findRobot() " + e.getMessage());  related
    }                                                        exceptions
}
private void connectToRobot(BluetoothDevice bd)      ❺ Connect to Robot
{
    try
    {
      socket = bd.createRfcommSocketToServiceRecord
(UUID.fromString("00001101-0000-1000-8000-00805F9B34FB"));  ❼ Get Socket
      socket.connect();                        ❽ Initiate connection  interface
    }
    catch (Exception e)
    {
      Log.e(tag,"Error interacting with remote device [" + e.getMessage() +
     "]");
    }
}
```

All Bluetooth[4] activities begin with the BluetoothAdapter ❶. With a reference to the adapter, you can obtain a list of already-paired devices ❷. You look through this list ❸ for a specific device name ❹ that corresponds to the name of your robot. This name may be hard-coded, as is done in this sample application; entered by the user at runtime; or even selected from a more sophisticated "choose" dialog. One way or another, the aim is to identify which BluetoothDevice you need and then initiate a connection, as done here with a call to the function named connectToRobot ❺. It's a good practice to catch exceptions ❻, particularly when dealing with remote physical devices that may not be in range or may have powered down. To connect to the remote device across the Serial Port Profile, use the createRfComSocketToServiceRecord method of the BluetoothDevice class. The UUID string shown in the code is the identifier for the Serial Port Profile ❼. Once you have a BluetoothSocket available, you call the connect method ❽.

At this point you've found the device of interest and attempted a connection request. Did it work? How do you know? You could make an assumption about the connection status and wait for an error to tell you otherwise. Perhaps that isn't the best approach. There must be a better way—and there is, but it involves working with Intents.

[4] See the Google documentation for more details about Bluetooth and Android: http://developer.android.com/guide/topics/wireless/bluetooth.html.

14.1.5 Capturing Bluetooth events

To verify that you've successfully connected to a BluetoothDevice, you must register for a couple of Bluetooth-related events: `ACTION_ACL_CONNECTED` and `ACTION_ACL_DISCONNECTED`. When these events occur, you know that you have a good connection, or you've lost a connection, respectively. So, how can you use these events in conjunction with your previously created socket? The following listing demonstrates a technique for creating a BroadcastReceiver directly in the `Activity` and registering for the events of interest.

Listing 14.2 Monitoring the Bluetooth connection

```
private BroadcastReceiver btMonitor = null;            ❶ BroadcastReceiver variable

private void setupBTMonitor() {                      ❷ SetupBTMonitor method
    btMonitor = new BroadcastReceiver() {
      @Override                                          ❸ Create BroadcastReceiver
      public void onReceive(Context context,Intent intent) {
        if (intent.getAction().equals(                           onReceive
"android.bluetooth.device.action.ACL_CONNECTED")) {                method ❹
          handleConnected();
        }                                             Connection
        if (intent.getAction().equals(              ❺ established
"android.bluetooth.device.action.ACL_DISCONNECTED")) {
          handleDisconnected();
        }                                       ❻ Connection lost

      }
    };
}
```

To monitor for specific broadcasted events, you need to employ a BroadcastReceiver. Ordinarily you'd do this with a separate class, but this application requires a more tightly integrated UI, so you take an alternative approach. Typically BroadcastReceivers are defined in the AndroidManifest.xml file, but in this case you only want notification under a specific set of circumstances. This code defines an `Activity`-scoped BroadcastReceiver named `btMonitor` ❶. In the `onCreate` method, the `setupBTMonitor` method ❷ is invoked to create the BroadcastReceiver ❸ along with the implementation of the `onReceive` method ❹. Whenever a broadcasted `Intent` is available for this BroadcastReceiver, the `onReceive` method is invoked. In this implementation, you're concerned with the connect and disconnect of a Bluetooth peer. When the devices are connected, the `handleConnected` method ❺ is invoked. Similarly when the remove device disconnects, the `handleDisconnected` method ❻ is called to perform the appropriate housekeeping operations.

With the device now connected, you need to perform some housekeeping to handle things such as setting up the socket's input and output streams. The next listing shows an abbreviated version of the `handleConnected` method showing the Bluetooth relevant portions.

Listing 14.3 The handleConnected method

```
private void handleConnected() {
    try {
      is =
     socket.getInputStream();
      os = socket.getOutputStream();
      bConnected = true;
      btnConnect.setVisibility(View.GONE);
      btnDisconnect.setVisibility(View.VISIBLE);
    } catch (Exception e) {
      is = null;
      os = null;
      disconnectFromRobot(null);
    }
}
```

❶ **Setup IO streams**

❷ **Set flag**

❸ **Swap Button visibility**

❹ **Handle exception**

❺ **Close connection on error**

When the `handleConnected` method is invoked, a valid Bluetooth socket connection has been established, so you need to set up the input and output streams ❶. With these streams established, data communications between the Android device and the LEGO robot may now begin. As you'll see later in this chapter, you only want to process sensor events if you're connected to a robot, so you set a flag ❷ letting the application know the status of the connection. You swap the visibility of a pair of `Buttons` ❸—one is used for connecting to the robot and the other for disconnecting. In the event that an error occurs during this step, you want to clean up by closing down the streams ❹ and initiating a disconnect request ❺.

The code for disconnecting a socket is simply this:

```
socket.close();
```

To perform most Bluetooth operations with Android, there's one important item that must be established: permissions!

14.1.6 Bluetooth permissions

Working with a paired device peer isn't the only place where permissions come into play. In order to exercise the Bluetooth APIs, an Android application must have the `BLUETOOTH` permission defined in the AndroidManifest.xml file:

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

The balance of the Bluetooth communications code is presented in the third section of this chapter, where we discuss in more depth the code that comprises the SenseBot application. Before jumping into the fun of coding and running the robot application, let's look at the `SensorManager` and show how you can put Android's sensors to work for you to drive a robot.

14.2 Interacting with the SensorManager

Android exposes the physical hardware sensors via a class known as the `SensorManager`. The `SensorManager` class is similar to the `BluetoothAdapter` class in that all related activities rely on having a reference to SensorManager. The SensorManager class is part of the android.hardware package. In this section, you'll learn how to read values from the orientation sensor, which you must learn to do before you build the SenseBot application.

Table 14.2 lists the major classes associated with the SensorManager.

Table 14.2 Sensor-related classes

Class	Comment
SensorManager	Primary interface to the various sensors present in the hardware
Sensor	Represents a particular sensor
SensorEvent	Represents the readings from a sensor
SensorEventListener	This interface used to receive SensorEvents in near real time

Working with the SensorManager class is simple. The first requirement is to obtain a reference:

```
SensorManager sManager = (SensorManager)
     getSystemService(Context.SENSOR_SERVICE);
```

Once you've obtained a valid reference, you can use this variable throughout the application to interact with the sensors themselves. For example, the SenseBot application utilizes the orientation sensor. To get a reference to this sensor, call the getDefaultSensor() method of SensorManager:

```
Sensor orientationSensor =
     sManager.getDefaultSensor(Sensor.TYPE_ORIENTATION);
```

We only use the orientation sensor in this chapter, but Android offers many more sensors. Let's look at the available sensor types as of Android 2.2.

14.2.1 Types of sensors

Android supports the sensor types listed in table 14.3.

Table 14.3 Android's common sensors

Sensor.TYPE_ACCELEROMETER	Measures acceleration in three dimensions
Sensor.TYPE_GYROSCOPE	Gyroscope
Sensor.TYPE_LIGHT	Ambient light sensor
Sensor.TYPE_MAGNETIC_FIELD	Measures magnetic field compass
Sensor.TYPE_ORIENTATION	Measures orientation in three dimensions
Sensor.TYPE_PRESSURE	Measures pressure
Sensor.TYPE_PROXIMITY	Measures distance the phone is away from another object, such as your ear
Sensor.TYPE_TEMPERATURE	Measures ambient temperature

Each sensor instance can provide a handful of useful and interesting attributes, including:

- Name of sensor
- Power consumption in mA
- Resolution
- Maximum range
- Vendor
- Version

The orientation sensor on a Nexus One shows the following characteristics:

- Name: AK8973 Orientation Sensor
- Power draw: 7.0 mA
- Resolution 1.0 degree
- Max range 360 degrees

Now that you have a feel for how to gain access to a sensor through `SensorManager`, let's explore reading values from a sensor.

14.2.2 Reading sensor values

You read a value from a sensor by implementing the `SensorEventListener` interface. `SensorEvent` instances are sent to a method named `onSensorChanged()`. The `SensorEvent` class contains four fields, as you can see in table 14.4.

Table 14.4 `SensorEvent`'s fields

Field	Comment
`accuracy`	This integer field represents the sensor's view of the accuracy of this reading.
`Sensor`	This is a reference to the sensor that created this `SensorEvent`.
`timestamp`	This is a nanosecond-based timestamp representing when the event occurred. This field can be helpful when you're correlating multiple events.
`values[3]`	The values from the sensor are provided as an array of floats with three values. The units and precision of the values vary by sensor.

The `SensorEventListener` receives these events each time the corresponding sensor values change. The following listing shows a slimmed-down version of the `onSensorChanged` method for the SenseBot application.

Listing 14.4 Our slimmed-down version of `onSensorChanged`

```
public void onSensorChanged(SensorEvent event) {            ❶ SensorEvent parameter
      try {

        if (bConnected == false) return;                    ❷ Check connected flag
        StringBuilder sb = new StringBuilder();
        sb.append("[" + event.values[0] + "]");
        sb.append("[" + event.values[1] + "]");
        sb.append("[" + event.values[2] + "]");

        readings.setText(sb.toString());

        // process this sensor data
        // updateMotors();
      } catch (Exception e) {
        Log.e(tag,"onSensorChanged Error::" + e.getMessage());
      }
    }
```

❸ Build visual representation

❹ Display values

❺ Interpret values

❻ Move robot accordingly

Each time a SensorEvent ❶ is available, it's passed to the onSensorChanged method. The first thing the code does is a safety check to make sure you have a good connection to the robot ❷. If there's no connection, you ignore the data. Each of the three values is extracted and formatted ❸ for display in a simple TextView widget ❹. The values are interpreted ❺ and the appropriate instructions are passed to control the robot's motors ❻. The logic for the interpretation and interaction with the robot's hardware is provided later in this chapter.

An application must register its SensorEventListener in order to receive these notifications. There's a prescribed manner in performing this registration process, which is up next.

14.2.3 Enabling and disabling sensors

The SensorEventListener interface receives messages only when it's registered. SensorManager provides two bookend-type functions that permit an application to register for a particular sensor's events. In the context of the SenseBot application, you're only interested in receiving orientation sensor events when the Android device is connected to the robot via Bluetooth. As such, you'll implement the registration code inside the previously introduced handleConnected method. The following listing shows the new code to be added to the handleConnected method.

Listing 14.5 Sensor registration code

```
sManager.registerListener(SenseBot.this,
      sManager.getDefaultSensor(
         Sensor.TYPE_ORIENTATION),
      SensorManager.SENSOR_DELAY_UI);
```

❶ Provide SensorEventListener

❷ Specify which Sensor

❸ Sensor update frequency

The registerListener method of the SensorManager takes three arguments in order to marshal sensor data to an application. The first argument is to an implementation instance of SensorEventListener, which is in this case our class itself, SenseBot.this ❶. The second argument is an instance of the sensor of interest. Here you're interested in tracking values for the orientation sensor ❷. The rate at which the sensor data is updated is variable and is specified by the programmer as the third parameter. In this case you use the value SensorManager.SENSOR_DELAY_UI ❸, which is a good general-purpose value. Use faster values for games or other real-time–oriented applications.

If you recall, the orientation sensor has a draw of 7 mA. To conserve power and battery life, you should be mindful to turn off the sensor when it's not required. In the SenseBot application, there are two places where this takes place. The first is in the `handleDisconnected` method—when you lose connection to the robot, you needn't take any further readings from the sensor. The more generic place to add this "unregister" functionality is in the `onStop` `Activity` lifecycle method.

Regardless of where the code is called, a `SensorEventListener` is unregistered with a simple call to the `unregisterListener` method of `SensorManager`:

```
sManager.unregisterListener(SenseBot.this);
```

Note that this call unregisters all sensors for this `SensorEventListener` in the event that your application registered more than one sensor type.

At this point you know how to both connect to the robot and read values from the orientation sensor. It's time to put all this knowledge together and build the SenseBot application!

14.3 Building the SenseBot application

The SenseBot application has a simple premise—you want to drive a LEGO Mindstorms NXT[5] robot by changing the orientation of the Android phone. There are no attached wires—all the communication is done via Bluetooth and the orientation of the phone alone should dictate how the robot moves. Furthermore, though the LEGO robot is programmable, you utilize only the built-in capabilities of the robot to manipulate individual motors. The benefit of this approach is that this program will work on virtually any LEGO robot built, regardless of the skill of the robot programmer. The only requirements of the robot are that the motors be connected to output ports B and C, which is the common manner of constructing LEGO NXT robots. Figure 14.5 shows the robot with a simple two-motor design.

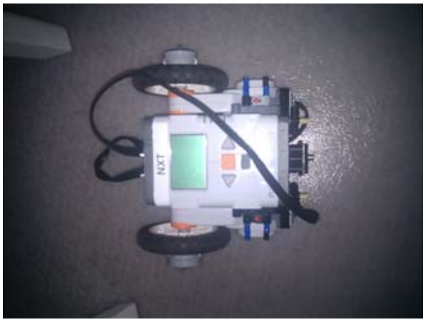


Figure 14.5 Simple LEGO NXT robot with motors connected to B and C ports

The robot can move forward and backward, spin to the left, and spin to the right. To drive the robot, you tilt the phone forward or backward, turn it on its side to the left, and turn it on its side to the right, respectively.

Although the robot is controlled entirely by the motion of the phone, you still have to create a useful and intuitive UI. In fact, the UI has a nontrivial role in the development of this application.

[5] If you have a future engineer or scientist in the making, check out First Lego League: http://www.firstlegoleague.org/.

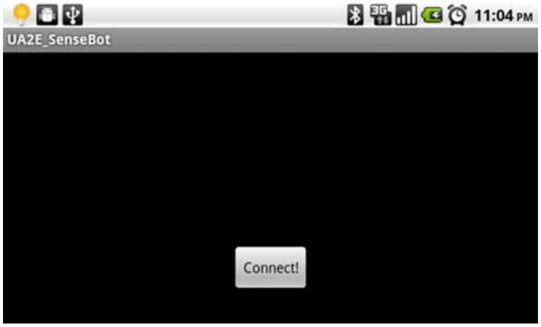


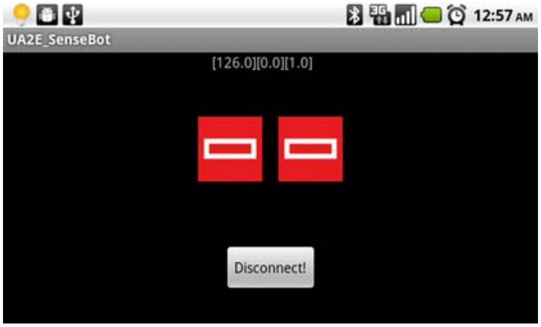
Figure 14.6 **Waiting to connect to a robot**

14.3.1 User interface

The UI for this application is simple but must be also intuitive for the user. You want to show the user what's happening to provide positive feedback on how to use the application. Additionally, you're dealing with a mechanical robot that may not function properly at all times. The robot may perform an unexpected action—therefore it's desirable that you have the ability to compare the robot's movement to the visual indicators you provide to the user. To that end, you need to indicate to the user the state of the motors at all times while the Android device is connected to the robot. Figure 14.6 shows the default user interface prior to connecting to a robot.

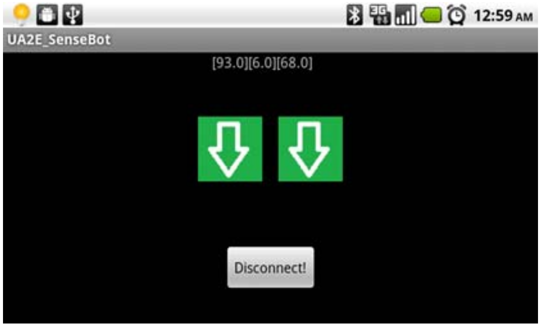
Clicking the Connect button initiates the connection sequence with a call to the `findRobot` method shown earlier in section 1.1.4. Once connected to the robot, you need to hide the Connect button and provide a means of disconnecting from the robot by displaying a Disconnect button. In addition, you want to indicate the state of the motors and display the sensor readings. Figure 14.7 shows the application after it has connected and with the motors in the stopped condition.

> **NOTE** The motor indicators on the screen are the values specified by the application and correlate to motor control instructions sent to the robot. They aren't measured values read from the robot.



[126.0][0.0][1.0]

Figure 14.7 **Connected to the robot with the motors stopped**

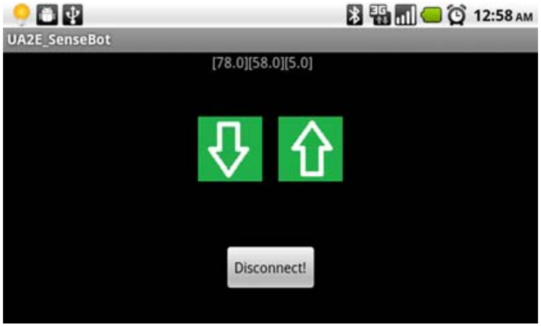


[93.0][6.0][68.0]

Figure 14.8 Both motors are moving backward.

If the robot's motors are moving while the screen indicates that they're both stopped, there's a problem either with the command sent by the robot or with the robot itself. Figure 14.8 is a screenshot taken from the application when guiding the robot to move backward.

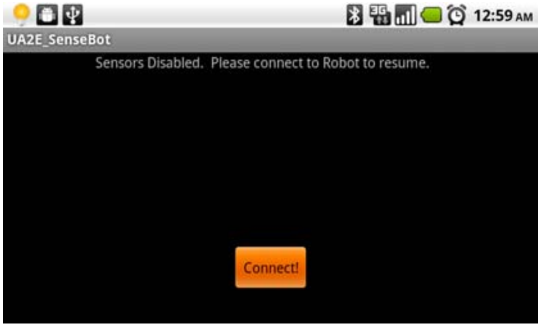
Figure 14.9 shows the application instructing the robot to spin to the left. To accomplish this, we've the left motor turning backward and the right motor turning forward.



[78.0][58.0][5.0]

Figure 14.9 Spinning to the left

Lastly, when the application disconnects from the robot (when you either click the Disconnect button or power off the robot), the application detects the Disconnected condition and calls `handleDisconnect`, and the UI is updated, as shown in figure 14.10.



Sensors Disabled. Please connect to Robot to resume.

Figure 14.10 Disconnected state, waiting for a new connection

The UI is generated by a pair of View widgets and three drawables:[6] stop, up (forward), and down (backward). Based on the values read from the sensors, the respective View widgets have their background changed appropriately.

This application is so dependent on the orientation of the phone for the control of the robot that you can't allow the phone's orientation to change back and forth between portrait and landscape, as it'll both restart the `Activity`, which could wreak some havoc, as well as change the orientation of the sensors. To meet this objective, an attribute was added to the activity tag in the AndroidManifest.xml file:

```
android:screenOrientation=landscape
```

Once this orientation is set up, there's no worry of the orientation changing to portrait while driving the robot. You'll find holding the phone in landscape is comfortable when you're "driving."

By carefully coordinating the UI with the physical motors, you have a ready feedback mechanism to both make you better robot drivers and help troubleshoot any anomalies during the development phase of this engineering project

The communications are established and the orientation sensor is producing values; it's now time to examine the interpretation of the sensor values.

14.3.2 *Interpreting sensor values*

To control the robot with the orientation of the phone, a "neutral zone" should be established with a "center" represented by the position of the phone when being held comfortably in a landscape orientation, slightly tilted back and up. Once this center is defined, a comfortable spacing or "sensitivity" is added in both of the x and y dimensions. As long as the phone's orientation in these dimensions doesn't exceed the sensitivity value, the motors remain in neutral and not powered. Variables named `xCenter`, `yCenter`, and `xSensitivity` and `ySensitivity` govern this "neutral box."

Look at the `onSensorChanged` method: this is where you receive the `SensorEvent` providing the values of each dimension x, y, and z. The following listing shows the complete implementation of this method, including the sensor evaluation and movement suggestions.

Listing 14.6 The `onSensorChanged` method, which interprets orientation

```
public void onSensorChanged(SensorEvent event) {
      try {
        if (bConnected == false) return;
        StringBuilder sb = new StringBuilder();
        sb.append("[" + event.values[0] + "]");
        sb.append("[" + event.values[1] + "]");
        sb.append("[" + event.values[2] + "]");

        readings.setText(sb.toString());
```

[6] Download a drawables application that lists all resources in android.R.drawable for the current Android device: http://www.appbrain.com/app/android-drawables/aws.apps.androidDrawables.

```
    // process this sensor data
    movementMask = MOTOR_B_STOP + MOTOR_C_STOP;             ❶ Default to stopped motors

    if (event.values[2] < (yCenter - ySensitivity)) {
     movementMask = MOTOR_B_FORWARD + MOTOR_C_FORWARD;
     motorPower = 75;        ❹ Set motor speed fast          ❷ Check forward/back

    } else if (event.values[2] > (yCenter + ySensitivity)) {
     movementMask = MOTOR_B_BACKWARD + MOTOR_C_BACKWARD;
     motorPower = 75;        ❹ Set motor speed fast

    } else if (event.values[1] >(xCenter + xSensitivity)) {
     movementMask = MOTOR_B_BACKWARD + MOTOR_C_FORWARD;
     motorPower = 50;        ❺ Set motor speed slow          ❸ Check left/right

    } else if (event.values[1] < (xCenter - xSensitivity)) {
     movementMask = MOTOR_B_FORWARD + MOTOR_C_BACKWARD;
     motorPower = 50;        ❺ Set motor speed slow

    }
   updateMotors();                                          ❻ Update motor values
   } catch (Exception e) {
    Log.e(tag,"onSensorChanged Error::" + e.getMessage());
   }
  }
```

When interpreting the values for the motors, you default to having both motors stopped ❶. Note that the B and C motors are managed separately. You check whether the y sensor value is outside the y quiet zone ❷. If the sensed value is beyond the "titled forward" boundary, you move the robot forward. Likewise, if the sensed value is further back than the resting position, you move the robot backward by marking both motors to be turned backward. If the robot hasn't been determined to be going either forward or backward, you check for the lateral options of left and right ❸. If the robot is moving forward or backward, the speed is set to 75% ❹. If the robot is to be spinning, its power is set to 50% ❺. The final step is to translate these movement masks into real actions by modifying the condition of the motors ❻ and to update the UI to reflect these commands.

Once the onSensorChanged method has completed processing the SensorEvent data, it's time to drive the robot's motors and update the user interface.

14.3.3 Driving the robot

Driving the robot is as simple—and as complex—as turning the motors on with a series of commands. The command protocol itself is shown in the next section; for now let's focus on the updateMotors method to see how both the UI and the motor positions are modified. The following listing displays the updateMotors method.

Listing 14.7 The updateMotors method

```
private void updateMotors() {
  try {
   if ((movementMask & MOTOR_B_FORWARD) == MOTOR_B_FORWARD) {    ❶ Check motor bitmask
    motorB.setBackgroundResource(R.drawable.uparrow);            ❷ Update graphic images
    MoveMotor(MOTOR_B,motorPower);          ❸ Send command to motor
   } else if ((movementMask & MOTOR_B_BACKWARD) == MOTOR_B_BACKWARD) {
    motorB.setBackgroundResource(R.drawable.downarrow);
    MoveMotor(MOTOR_B,-motorPower);         ❸ Send command to motor
   } else {
    motorB.setBackgroundResource(R.drawable.stop);
    MoveMotor(MOTOR_B,0);
   }

   if ((movementMask & MOTOR_C_FORWARD) == MOTOR_C_FORWARD) {
    motorC.setBackgroundResource(R.drawable.uparrow);
    MoveMotor(MOTOR_C,motorPower);          ❸ Send command to motor
   } else if ((movementMask & MOTOR_C_BACKWARD) == MOTOR_C_BACKWARD) {
    motorC.setBackgroundResource(R.drawable.downarrow);
    MoveMotor(MOTOR_C,-motorPower);         ❸ Send command to motor
   } else {
    motorC.setBackgroundResource(R.drawable.stop);
    MoveMotor(MOTOR_C,0);
   }

  } catch (Exception e) {
   Log.e(tag,"updateMotors error::" + e.getMessage());
  }
}
```

❶ Check motor bitmask

❷

The updateMotors method compares the requested movement as defined in the movementMask variable with each of the motors individually ❶. When a match is found—for example, when the MOTOR_B_FORWARD bit is set—the particular motor is enabled in the specified direction and speed ❸. A negative direction means backwards and the power value is scaled between 0 and 100. Additionally, the UI is updated ❷ in conjunction with the motors themselves, thereby giving the user as accurate a picture as possible of their performance as a driver.

14.3.4 Communication with the robot

The communications protocol for interacting with the LEGO NXT robot is a structured command with optional response protocol. Each packet of data is wrapped in an envelope describing its size. Within the envelope, each "direct command" has a standard header followed by its own specific parameters. For this application you need but a single command—to set the output state of the motor. The code that builds and sends these packets is shown in the next listing.

Listing 14.8 The MoveMotor method

```
private void MoveMotor(int motor,int speed)
{
  try
  {
   byte[] buffer = new byte[14];           ❶ Declare buffer
```

```
        buffer[0] = (byte) (14-2); //length lsb
        buffer[1] = 0; // length msb
        buffer[2] =  0; // direct command (with response)
        buffer[3] = 0x04; // set output state
        buffer[4] = (byte) motor; // output 0,1,2 (motors A,B,C)
        buffer[5] = (byte) speed; // power
        buffer[6] = 1 + 2; // motor on + brake between PWM
        buffer[7] = 0; // regulation
        buffer[8] = 0; // turn rotation
        buffer[9] = 0x20; // run state
        buffer[10] = 0; // four bytes of position data.
        buffer[11] = 0; // leave zero
        buffer[12] = 0;
        buffer[13] = 0;

        os.write(buffer);
        os.flush();
        byte response [] = ReadResponse(4);
      }
      catch (Exception e)
      {
        Log.e(tag,"Error in MoveForward(" + e.getMessage() + ")");
      }
}
```

❷ Format buffered command

❸ Write command

This code performs the simple yet precise operation of formatting a command, which is sent to the LEGO robot to provide direct control over the motors. A buffer of the appropriate size is declared ❶. The size for this buffer is dictated by the `SetOutputState` command, which is one of many commands supported by the robot. Each of the various data elements are carefully provided ❷ in their respective locations. Once the command buffer is formatted, it's written and flushed to the socket ❸. The response code is consumed as a good measure by the `ReadResponse` method. As you can see, aside from the specific formatting related to controlling the robot, sending and receiving data with Bluetooth is as simple as reading or writing from a byte-oriented stream.

At this point, the sensors are working and the Android device and LEGO robot are communicating. In time, with practice you'll be an expert Android LEGO pilot. The full source code to this application is available for download.

14.4 Summary

This chapter introduced two hardware-oriented features of the Android platform: Bluetooth and sensors. From these seemingly unrelated areas of functionality grew a fun application to operate a LEGO Mindstorms NXT robot. We demonstrated the essential steps required to connect an Android device to a remote Bluetooth-enabled peer via the use of the RFCOMM cable replacement protocol. This communications channel is used to exchange a command set known as the Direct Command protocol provided by the LEGO NXT controller. Through this command set, you can manipulate the robot's motors to drive the robot. To make the user experience as intuitive as possible, use the orientation sensor built into most Android hardware to sense motions made by the user. The position of the device is interpreted and a corresponding set of commands is given to navigate the robot. Not only do the sensors provide a functional means for driving the robot, it's quite fun!

In addition to these core Bluetooth communications and sensor interactions, this chapter also demonstrated techniques for providing intuitive user feedback during the operation of the application. For example, as the motors are engaged, the user visually sees the direction each motor is being driven. Likewise, the user's driving motions are only processed when an active Bluetooth connection is detected. Because this is an event-driven scenario, the application demonstrates listening for these events through the use of a dynamically registered `BroadcastReceiver` with appropriate `IntentFilters`.

Hopefully you've enjoyed learning about Bluetooth and sensors in this chapter, and perhaps you even have access to a LEGO Mindstorm robot to take for a spin!

In the next chapter you'll learn about another means of connecting your Android device to the outside world—this time working with the integration capabilities of the platform to sync data with the popular business networking site LinkedIn.

15 Integration

This chapter covers

- Manipulating and extending Android contacts
- Managing multiple accounts
- Synchronizing data to a remote server

No phone is an island. A mobile smartphone's primary purpose is to connect with others, whether through voice calls, email, text messaging, or some other way of reaching out. But phones have historically acted like islands when it came to storing information. You might painstakingly save phone numbers for years on your device, only to lose everything and start all over again when you switched phones.

Android leads the charge in breaking away from the old device-centric way of storing contacts and related data. Like all of Google's services, Android looks to the cloud as a vast storage location for all your data. Integration allows your phone to stay in sync with what you care about, so you don't need to manually copy doctors' appointments and important emails from multiple computers to your phone.

For the most part, Android users can feel happily oblivious to much of this; everything "just works." As a developer, you may want to take advantage of Android's integration features and add new conveniences for your users. This chapter shows exactly how Android handles personal data such as contacts, how accounts work, what synchronization does, and what hooks you can play with. Along the way, we'll build a real-world example: a plug-in that allows people to automatically sync their Android contacts with connections they've made on LinkedIn. LinkedIn is the social network most closely associated with business users, who use it to maintain their professional contacts. Synchronizing with this account will allow users to connect with their LinkedIn colleagues, making their information available anytime and anywhere.

15.1 Understanding the Android contact model

Contacts have long been the single most important feature on a mobile phone. After all, a phone exists to call people, and you don't want to memorize the 10-digit phone number for every person you might call.

If you've owned mobile phones for a long time, you've probably noticed a gradual increase in the functionality of mobile contact applications. In the early days, people could only enter a name and a single phone number, similar to a speed-dial on a landline phone. Modern phones allow you to enter multiple numbers for each contact, as well as email addresses, photos, birthdays, and more. Not only does this provide greater convenience to consumers, who now can turn to a single source for all related information about a person; it also opens up many new opportunities for application developers, who can take advantage of the wealth of information on a phone to make their apps more useful and relevant.

Android has always offered access to a user's contacts, but its contact model has changed through different versions. The model is now extremely robust and flexible, offering great features for querying and updating contact information.

15.1.1 Choosing open-ended records

Imagine that you just got a new phone and started entering contacts. You entered three names and phone numbers. The final result would look something like figure 15.1. From a user's perspective, you have all that you need. But how is this data being stored?

In older phones, the underlying contact model might result in something similar to figure 15.2. This *fixed-length record* would dedicate a predetermined number of bytes to each contact. Such an approach had hard limits: for example, you might only be allowed 10 digits per phone number, 12 characters per name, and 500 records per device. This led to both undesirable truncation and wasted space.

In addition to the problem of truncation, you can see that this style of contact storage doesn't allow the user to add new types of data. Figure 15.2 shows that there's no space between Chris and Frank to insert a

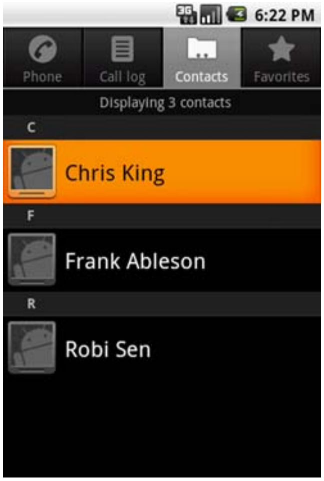
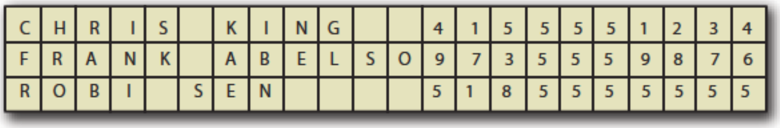


Figure 15.1 Making entries in the native Contacts application



C	H	R	I	S		K	I	N	G			4	1	5	5	5	5	1	2	3	4
F	R	A	N	K		A	B	E	L	S	O	9	7	3	5	5	5	9	8	7	6
R	O	B	I		S	E	N					5	1	8	5	5	5	5	5	5	5

Figure 15.2 Fixed-length storage records

mailing address or job title. Fortunately, Android uses a database to store its contact data, so the storage looks more like figure 15.3. Each contact is a collection of related data, and each piece of data can be as short or as long as it needs to be. Contacts can omit the data that they don't require. These are called open-ended records: records can be freely extended to include previously unknown pieces of data.

You can see exactly how the data is stored by querying the contacts database. See chapter 5 if you need a review of using the sqlite3 tool, with which we'll query the tables that were created when adding the initial three contacts. In figure 15.4, we connect to the sqlite database stored at `/data/data/com.android.providers.contacts/databases/contacts2.db`. Querying the `data` table returns the individual pieces of contact information on the device. The `contacts` table includes individual contact entities. Joining these two tables results in the information visible in the contacts application.

Why bother with fixed-length records?

Given these limitations, you might wonder why anyone would've adopted fixed-length records in the first place. They do have the advantage of being fast. Because each record is the same length, the phone can immediately jump to the exact location in memory where, say, the 42nd record starts, without needing to search for it.

When phones had extremely limited processors and slow flash memory, this design allowed for a much speedier experience. Now that phones have gotten faster, OS designers can afford to be more flexible.

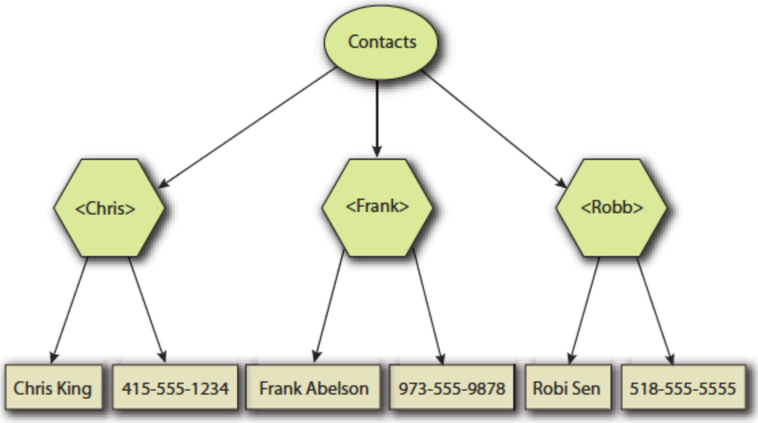
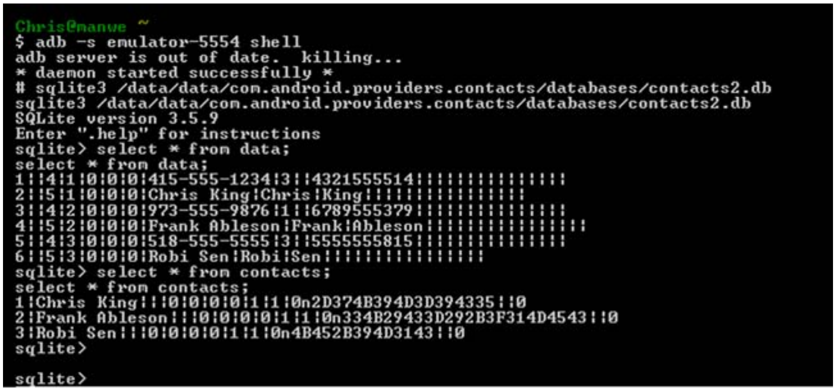


Figure 15.3 Open-ended storage records



```
Chris@manwe ~
$ adb -s emulator-5554 shell
adb server is out of date.  killing...
* daemon started successfully *
# sqlite3 /data/data/com.android.providers.contacts/databases/contacts2.db
sqlite3 /data/data/com.android.providers.contacts/databases/contacts2.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> select * from data;
select * from data;
1||4|1|0|0|0|415-555-1234|3||4321555514|||||||||||||||
2||5|1|0|0|0|Chris King|Chris|King||||||||||||||||
3||4|2|0|0|0|973-555-9876|1||6789555379|||||||||||||||
4||5|2|0|0|0|Frank Ableson|Frank|Ableson||||||||||||||||
5||4|3|0|0|0|518-555-5555|3||5555555815|||||||||||||||
6||5|3|0|0|0|Robi Sen|Robi|Sen||||||||||||||||
sqlite> select * from contacts;
select * from contacts;
1|Chris King|||0|0|0|0|1|1|0n2D374B394D3D394335||0
2|Frank Ableson|||0|0|0|0|1|1|0n334B29433D292B3F314D4543||0
3|Robi Sen|||0|0|0|0|1|1|0n4B452B394D3143||0
sqlite>

sqlite>
```

Figure 15.4 Querying the contacts database

Android's contacts are highly extensible because they're stored as records in a database. Third-party applications can choose to define and add their own enhanced data fields. For example, a business application may include employee ID numbers, and a gaming application may use online handles. Other phone platforms require developers to maintain parallel data stores to add this kind of data, but Android allows you to insert directly into the main contacts database. This leads to much better consistency, and your enhanced information will automatically be updated when the user renames or deletes a contact.

15.1.2 Dealing with multiple accounts

Most Android phones require users to activate the device and associate it with a Google account. Upon activation, all of the user's existing Gmail contacts will be available on the phone. This is convenient for people who keep all their information on Gmail, but not terribly useful for people who have multiple accounts or don't want to use Gmail at all.

The early versions of Android were hampered by this problem, but fortunately, the new Android contact model offers a richer set of features that includes support for multiple accounts. Users still must generally tie the phone to a Google account, but they may also connect with a Microsoft Exchange[1] account for business contacts and email, with Facebook for friends' statuses, and possibly other private accounts as well.

Using multiple accounts offers great convenience but also greater complications. Consider the following issues:

- *Differing data*—Each account will support disparate types of data. For example, Facebook may offer an Instant Messaging screen name, whereas Exchange may provide a contact's supervisor.

[1] For information on Microsoft Exchange Server development technologies, go to http://www.outlookcode.com/archive0/d/exstech.htm.

- *Different formats*—Even if the same type of data is supported by multiple accounts, it may display differently. For example, one account may return a phone number as (415) 555-1234, whereas another returns it as 4155551234.
- *Consolidation*—Having multiple accounts isn't very convenient if you need to search through three versions of each contact to find the one with the information you need. Instead, you want to have a single contact that includes data from all your accounts.
- *Linkage*—On a related note, the phone needs some way to determine when two different accounts are referring to the same entity. If one refers to "James Madison" and the other to "Jim Madison," a human could guess that they refer to the same contact, but software requires some more information to determine how they link together.
- *Ownership*—Some contacts may only appear on one account, whereas others may be listed in several. When you change a contact, you want to be sure they're updated appropriately.

Android resolves these issues gracefully. Each account stores its data in separate rows in the contacts database. The raw_contacts table contains all the contacts that have been received from all of the user's accounts, including potentially duplicate contacts. Table 15.1 shows a hypothetical view of a device's contacts.

Table 15.1 Raw contacts including duplicates from multiple accounts

ID	Name	Email	Phone number	Status
1	James Madison	jmadison@gmail.com		
2	Adam Smith	asmith@gmail.com		
3	James Madison		2024561414	"Negotiating with the British…"
4	Adam Smith	adam.smith@example.com	+442055512345	

NOTE As you've seen, the actual contacts data fields will reside in the data table. The raw_contacts table allows you to associate the various rows in the data table with their corresponding accounts.

Splitting out the accounts' views like this gives Android a great deal of flexibility. Whenever a piece of data is updated, it knows where that data originally came from. It can store data in the format most appropriate to the original account. And, because each account gets its own set of rows, no account can mess up the data that originally came from another account. This type of loose coupling extends a familiar Android philosophy to the world of contacts and provides the same sorts of benefits that you've seen elsewhere: more extensibility, stability, and transparency, at the cost of greater complication.

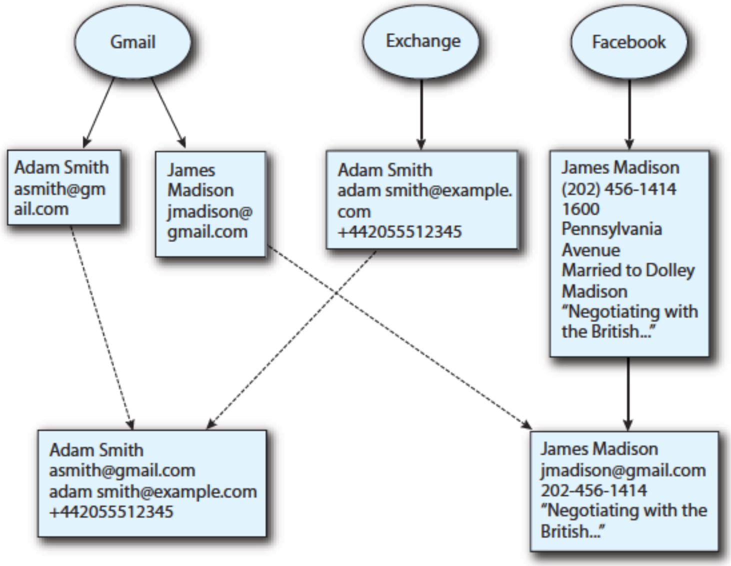


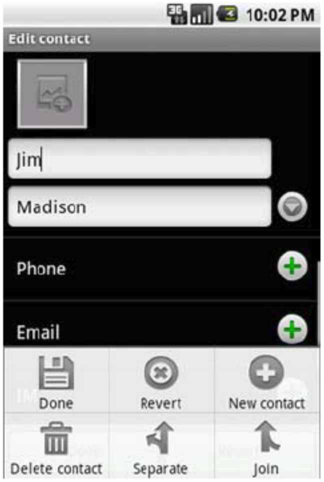
Figure 15.5 Combining raw contact data into unique contacts

15.1.3 Unifying a local view from diverse remote stores

Storing raw contact data from each account makes architectural sense, but by itself it'd lead to a horrible user experience. If you had five accounts, you might have five separate entries for the same person, which would lead to extra complications whenever you wanted to call her. You want to have just one contact entry for each "real" contact you have, no matter how many accounts they're associated with. Android provides exactly this utility.

Figure 15.5 shows how Android consolidates the information received from multiple accounts. In this example, the user has two contacts that are spread across three remote accounts. In the first phase, contacts are retrieved from each account. Next, Android will associate the matching accounts and consolidate each into a single logical contact. The raw contact data remains on the device, but the user will see and interact with the consolidated contact.

This process works well but isn't perfect. Android may require additional help to recognize that two contacts refer to the same person; sometimes it may mistakenly combine two records when it should've kept them separate. Fortunately, because Android never removes the raw contact data, you can easily join and separate accounts at any time, as shown in figure 15.6. This updates the contacts table and either creates or deletes a row there, but will leave raw_contacts untouched.



Jim

Madison

Figure 15.6 Joining and separating contacts

CAUTION Android currently doesn't offer a way to specify what name should be used after joining two records. You may get lucky, or you may be stuck with the wrong one.

15.1.4 Sharing the playground

With all the flexibility of Android's new contact model comes potential for abuse. A single consolidated database holds all contacts information on the device, and any application can read from or write to it. Users must give apps permission before modifying the database, but within the database, there's no notion of particular columns or rows being "owned" by a particular app or service.

When you write apps that use contacts data, try to be a good citizen. This includes the following principles:

- Only read and write Android's built-in data fields and fields that you've created for your application.
- Be sure to provide unique types for new contact data that you define, such as `vnd.manning.cursor.item/birthday` instead of `birthday`.
- Always ask users for permission before doing anything destructive to their data.
- Clearly state if and how your app uses their personal contact data.
- If you store contact data taken from the user, be certain it's well secured. You don't want to be responsible for addresses falling into the hands of spammers or identity thieves.

Not only does upholding these standards garner good karma, but it also keeps your users happy and more likely to use your app.

15.2 Getting started with LinkedIn

You'll build a sample app throughout the rest of this chapter that illustrates the concepts and techniques we've covered. To learn how contacts, accounts, and synchronizers all work together, you'll build a plug-in for LinkedIn.[2]

Like most social networking sites, LinkedIn offers a developer API for building custom applications on top of their platform. To protect against abuse, LinkedIn requires the use of a developer API key with each request; this allows the company to shut off an app if it's found to be misbehaving. You can get started developing with LinkedIn by following these steps:

1. Go to http://developer.linkedin.com.
2. Follow the links to request an API key for the LinkedIn APIs.
3. Fill in your information and join the Developer Network. You'll need to create a LinkedIn profile if you don't already have one.
4. Add a new application. There's no charge, and you should automatically receive your API keys.

[2] Check out the LinkedIn blog for developers: http://blog.linkedin.com/category/linkedin-developer-network/.

Once you receive your API key, take a look at the documentation available at http://developer.linkedin.com. LinkedIn also maintains a fairly active developer community in forums hosted at the same site if you need advice or want to see what the experts are saying.

Create a new Eclipse project called LinkedIn. The following listing shows a simple class called `LinkedIn` that holds useful constants you'll use later.

Listing 15.1 Constants for `LinkedIn`

```
package com.manning.unlockingandroid.linkedin;
public class LinkedIn {
   public static final String MIME_TYPE =
      "vnd.android.cursor.item/vnd.linkedin.profile";
   public static final String TYPE =
      "com.manning.unlockingandroid.linkedin";
   public static final String API_KEY = "";
   public static final String SECRET_KEY = "";
   public static final String AUTH_TOKEN = "AuthToken";
   public static final String AUTH_TOKEN_SECRET = "AuthTokenSecret";
}
```

Use your LinkedIn API keys

The `MIME_TYPE` refers to the new type of data you'll be adding to contacts. With this information, Android will be able to distinguish between a contact's LinkedIn data and data received from other sources. In the next section, you'll use this type to insert new LinkedIn records. The other constants will be useful in later sections as you begin to connect with the LinkedIn server.

The LinkedIn docs describe how to use their API. You can follow this to implement your app, but you'll be taking advantage of the `linkedin-j` project. Written by Nabeel Siddiqui, it comes under a generous Apache 2 license and provides a simple, logical wrapper around the raw API calls. Visit the project page at http://code.google.com/p/linkedin-j. Once you've downloaded the JAR files, add them as external JARs to your Java project.

While you're doing the initial setup, let's define all the strings for this application in strings.xml, based on the following listing. Most will be used by the login UI activity, but some will be used for other user-visible data.

Listing 15.2 Externalized strings for LinkedIn

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">LinkedInAdapter</string>
  <string name="login_activity_email_label">Email Address</string>
  <string name="login_activity_instructions">Please click the
    button below to log in to LinkedIn.</string>
  <string name="remoteLogin">Log In Online</string>
  <string name="login_activity_pin_label">PIN</string>
  <string name="login_activity_ok_button">Sign in</string>
  <string name="empty_fields_error">Please enter your email
    address and password.</string>
  <string name="start_login_error">Unable to connect to LinkedIn.
```

```
    Please verify your network connection and try again later.</string>
  <string name="login_fail_error">Login failed.
    Please click Log In Online and get a new PIN.</string>
  <string name="working">Finishing authentication.
    Please wait...</string>
  <string name="login_label">LinkedIn Login</string>
  <string name="contact_summary">LinkedIn</string>
  <string name="auth_token_label">LinkedIn</string>
</resources>
```

15.3 Managing contacts

Now that you understand the purpose behind Android's contact model, it's time to start interacting with it. Many applications can be improved by plugging into a user's contacts. The degree of integration that you need, though, will differ depending on the type of app you're writing. This section looks at three increasingly complex ways to hook into the phone's contacts.

15.3.1 Leveraging the built-in contacts app

Why reinvent the wheel? Android's engineers have spent the time and effort to build an attractive and familiar contacts app. In many cases, it's quicker and better to just use that app when you need a contact.

As you'd expect, you communicate with the native contacts app by using an `Intent`. The platform will respond to an `ACTION_PICK` request by bringing the list of contacts to the foreground, allowing the user to select a single contact, and then returning that selection to your application. You specify that you want to select a contact by requesting the `android.provider.ContactsContract.Contacts.CONTENT_URI` data type.

> **NOTE** The `ContactsContract` class and all related classes were introduced during the overhaul of the contacts model in Android 2.0. You may occasionally see legacy code that uses the `android.provider.Contacts` class for interacting with contacts. This usage is deprecated; though it works, it won't properly handle information from non-Google accounts.

After the user selects a contact, the response returns to your activity's `onActivityResult` method. You can query this result to pull out information such as the contact's name and ID, as shown in the following listing from a class called `ContactPicker`.

Listing 15.3 Selecting a contact from the native contacts app

```
public static final int CONTACT_SELECTED = 1;
private void selectContact() {
   Intent chooser = new Intent(Intent.ACTION_PICK,
      ContactsContract.Contacts.CONTENT_URI);
   startActivityForResult(chooser, CONTACT_SELECTED);
}
public void onActivityResult(
   int requestCode, int resultCode, Intent data) {
   super.onActivityResult(requestCode, resultCode, data);
```

Define unique ID for result

```
   switch (requestCode) {
   case (CONTACT_SELECTED):
      if (resultCode == Activity.RESULT_OK) {
         Uri contactData = data.getData();
         Cursor c = managedQuery(contactData, null, null, null, null);
         if (c.moveToFirst()) {
            int nameIndex = c.getColumnIndexOrThrow
               (ContactsContract.Contacts.
               DISPLAY_NAME);
            String name = c.getString(nameIndex);
            Toast.makeText(this, name, 2000).show();
         }
      }
   }
}
```

Detect selection

Retrieve name from result

If you inspect the data returned by this cursor, you'll notice that it doesn't contain everything that you might expect from a contact. For example, though it includes fields for the name and photo, it won't include any email addresses or phone numbers. As you saw in section 15.1.1, the open-ended style of contact record storage in Android requires contact data to be stored in a separate table from the actual contacts. Table 15.2 illustrates one hypothetical contacts table.

Table 15.2 The contacts table holds minimal information about each contact.

Contact ID	Contact name
1	Chris King
2	Frank Ableson
3	Robi Sen

When you pick a contact, you've found the entry in the first table; you can then use that contact ID as a foreign key to retrieve the extended data that you want. Table 15.3 shows the corresponding detailed information from the data table.

To retrieve the email address for a selected contact, you look up the contact's ID, and then use that ID in a new query against the `data` table. For convenience, the most useful data types have definitions in classes within the `ContactsContract.CommonDataKinds` class. For email, you can use `CommonDataKinds.Email`, which provides the URI and column names. The following listing expands listing 15.3 by making a new database query to determine whether the selected contact has an email address.

Data ID	Data type	Contact ID	Data value
1	4	1	415-555-1234
2	1	1	cking@example.com
3	4	2	973-555-9876
4	4	3	518-555-5555

Table 15.3 The data table holds extended information for contacts.

Listing 15.4 Retrieving email information for a selected contact

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
   super.onActivityResult(requestCode, resultCode, data);
   switch (requestCode) {
   case (CONTACT_SELECTED):
      if (resultCode == Activity.RESULT_OK) {
         Uri contactData = data.getData();
         Cursor c = managedQuery(contactData, null, null, null, null);
         if (c.moveToFirst()) {
            try {
               int contactID = c.getInt(c.getColumnIndexOrThrow
                  (ContactsContract.Contacts._ID));
               Uri uri = ContactsContract.
               CommonDataKinds.Email.CONTENT_URI;
               String[] projection = new String[] {
                  ContactsContract.CommonDataKinds.
                     Email.DATA }; #B
               String selection = ContactsContract.
                  CommonDataKinds.Email.CONTACT_ID +
                  "=?";
               String[] selectionArgs = new String[]
                  { "" + contactID };
               c.close();
               c = managedQuery(uri, projection, selection,
                  selectionArgs, null);
               String message;
               if (c.moveToFirst()) {
                  message = "Selected email address " + c.getString(0);
               } else {
                  message = "No email address found.";
               }
               Toast.makeText(this, message, 2000).show();
            } finally {
               c.close();
            }
         }
      }
      break;
   }
}
```

Limit results to this contact

If you run this code, you'll see a `Toast` with either a selected email address or a graceful error message. Please review chapter 8 if you'd like a reminder of how `Toast` objects work. The `data` table is open-ended, so a user may have multiple email addresses. Depending on the needs of your application, you could choose to iterate through all of them or only pick one, as shown in this example.

You can adopt this technique to retrieve any other data about a contact selected by the user. Browse the classes within `ContactsContract.CommonDataKinds` for natively supported fields, or use custom fields added by your own application.

Storing contact identifiers

An Android contact is a transient thing. Users may join different contacts together, split them apart, or delete them altogether. As such, you should avoid holding onto a contact ID for a long time and using it in future queries. If you need to retain a long-lived reference to a contact, such as a list of the user's gaming buddies, then instead of the contact ID you should use the lookup key, which is a column defined by `ContactsContract.Contacts.LOOKUP_KEY`. The lookup key will continue to work even if the user joins or separates the contact.

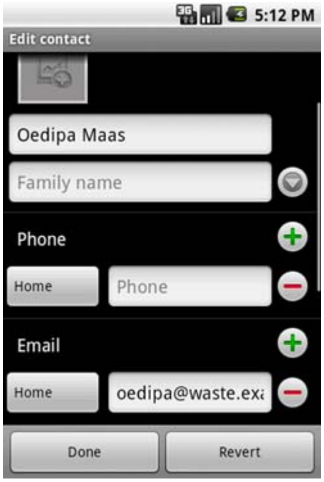
But using the lookup key is slower than the contact ID, so if speed is critical in your application, you may want to keep both the contact ID and the lookup key, and only use the lookup key if retrieving by ID fails.

15.3.2 *Requesting operations from your app*

Now that we've retrieved contact data from the native app, let's see how to edit that data. You'll often want to use the native contact app to perform edits, because users are comfortable and familiar with this interface.

To create a new contact, use `Intent.ACTION_INSERT`. By itself this will pop open an empty contact for the user to fill out, which isn't terribly useful. Most often, you'll have some pieces of information about a new contact and ask the user to supply the rest. For example, your app might retrieve screen names and email addresses from a social networking service. You could fill out these portions of the contact ahead of time and let the user finish adding the person's name in the native contacts app. Figure 15.7 shows a prepopulated contacts screen that was generated by an application.

Listing 15.5 shows how to generate this sort of screen. The fields that you can optionally prepopulate are available as static fields in the `ContactsContract.Intents.Insert` convenience class.



Oedipa Maas

oedipa@waste.exa

Figure 15.7 Partially complete contact requested by application

Listing 15.5 Adding a contact using the native contacts app

```
private void createContact() {
   Intent creator = new Intent(Intent.ACTION_INSERT,
      ContactsContract.Contacts.CONTENT_URI);
   creator.putExtra(ContactsContract.Intents.Insert.
      NAME, "Oedipa Maas");
   creator.putExtra(ContactsContract.Intents.Insert.
      EMAIL, "oedipa@waste.example.com");
   startActivity(creator);
}
```

Insert contact

Define initial values

To edit a contact, request `Intent.ACTION_EDIT`. But unlike creating or picking a contact, when editing a contact you need a reference to a specific person. You could retrieve one by launching the picker, but that's needlessly cumbersome. Instead, query for a particular contact, and then use its returned ID to launch the edit activity, as shown in the next listing.

Listing 15.6 Looking up a contact by name and then editing it

```
private void editContact() {
   Cursor c = null;
   try {
      Uri uri = ContactsContract.Contacts.CONTENT_URI;
      String[] projection = new String[] { ContactsContract.Contacts._ID };
      String selection = ContactsContract.Contacts.DISPLAY_NAME + "=?";
      String[] selectionArgs = new String[] { "Oedipa Maas" };
      c = managedQuery(uri, projection, selection, selectionArgs, null);
      if (c.moveToFirst()) {
         int id = c.getInt(0);
         Uri contact = ContentUris.appendId(
            ContactsContract.Contacts.CONTENT_URI.
            buildUpon(), id).build();
         Intent editor = new Intent(Intent.ACTION_EDIT, contact);
         startActivity(editor);
      }
   } finally {
      if (c != null)
      c.close();
   }
}
```

Build URI to retrieved contact

NOTE You can't insert new values into an existing contact in the same way that you can for a new contact. Any `Intent` extra fields will be ignored. Instead, use the techniques in the next section to edit the contact manually, and then display the updated contact for the user to further edit and approve.

15.3.3 Directly reading and modifying the contacts database

In many situations you want to take advantage of the built-in contacts app. But other times you should bypass it entirely and operate on the data itself. For example, your app might perform batch operations on a large number of contacts; it'd be tedious for the user to manually approve each individual change. Contact information is ultimately just another type of content stored on the phone, and you can use existing tools to look up and manipulate content.

You may have noticed that no special permissions were required to insert or edit contacts through the native application. This is because your application isn't directly modifying the data itself; it's merely issuing a request. But if you want to directly modify contact data, you'll need to secure the `android.permission.WRITE_CONTACTS` permission in your app's manifest.

You can retrieve the relevant URIs through queries such as those shown in listing 15.6, or by using the convenience definitions included under the `ContactsContract` class. Instead of retrieving a Cursor to enumerate results, use the ContentResolver class to manipulate data. Listing 15.7 demonstrates how to perform a conditional global replacement across all email addresses in the user's contacts list. Specifically, it'll find all addresses that contain the phrase *waste* and replace them with a concealed address.

Listing 15.7 Updating contact information using a ContentResolver

```
private void silentEditContact() {
   ContentResolver resolver = getContentResolver();
   Uri contactUri = ContactsContract.Data.CONTENT_URI;
   ContentValues values = new ContentValues();
   values.put(ContactsContract.CommonDataKinds.
      Email.DATA, "concealed@example.com");
   String where = ContactsContract.CommonDataKinds.
      Email.DATA + " like '%waste%'";
   resolver.update(contactUri, values, where, null);
}
```

❶ All contact data

❷ New email address

In this example, the Uri ❶ operates across all information for all contacts on the device. This is essential because we're updating the data associated with a contact; the ContactsContract.Contacts.CONTENT_URI only refers to basic contact information without extended data. The ContentValues ❷ describes the new information to place within the contacts. By itself, this would replace the email for every contact on the device, which would be very destructive. The selection clause limits the update so it only applies to addresses containing this string.

> **CAUTION** Few things will anger users quicker than wrecking their contact data. If your app directly creates, edits, or deletes contact data, you should clearly explain to users what it does, and test thoroughly to ensure your app behaves itself.

This is only one example of how to directly modify contact data, but combined with the earlier discussions in this chapter, you should now be able to change any contact data. You only need to find where the information is stored in the data table, find or describe the rows that need to be modified, then use ContentResolver to insert or update your data.

15.3.4 Adding contacts

Our LinkedIn app begins with people. In addition to standard information such as contacts' names, LinkedIn includes expanded details such as their educational background and work history. You'll take advantage of Android's open-ended contact model to store a unique type of data, the LinkedIn *headline*. The headline is a brief and punchy summary of someone's role, such as "Code Ninja" or "Senior Director at Stationary Networks."

To create a LinkedIn contact, you'll make a new entry in the raw_contacts table, and also create any new data fields for that contact. The following listing shows a utility class that accepts LinkedIn-style data and either creates a new record or updates an existing record.

Listing 15.8 `Helper` class for storing LinkedIn connections

```
package com.manning.unlockingandroid.linkedin;
// imports omitted for brevity
public class ContactHelper {
   public static boolean addContact(ContentResolver resolver,
         Account account, String name, String username, String headline) {
      ArrayList<ContentProviderOperation> batch =                 ❶ Batch
         new ArrayList<ContentProviderOperation>();                 operations

      ContentProviderOperation.Builder builder = ContentProviderOperation
         .newInsert(RawContacts.CONTENT_URI);
      builder.withValue(RawContacts.ACCOUNT_NAME, account.name);
      builder.withValue(RawContacts.ACCOUNT_TYPE, account.type);
      builder.withValue(RawContacts.SYNC1, username);             Unique ID for
      batch.add(builder.build());                                  future sync

      builder = ContentProviderOperation
         .newInsert(ContactsContract.Data.CONTENT_URI);
      builder.withValueBackReference(ContactsContract.CommonDataKinds.
         StructuredName.RAW_CONTACT_ID, 0);
      builder.withValue(ContactsContract.Data.MIMETYPE, ContactsContract.
         CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE);
      builder.withValue(ContactsContract.CommonDataKinds.StructuredName.
         DISPLAY_NAME, name);
      batch.add(builder.build());

      builder = ContentProviderOperation.newInsert(
         ContactsContract.Data.CONTENT_URI);
      builder.withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID,
         0);
      builder.withValue(ContactsContract.Data.MIMETYPE,
         LinkedIn.MIME_TYPE);
      builder.withValue(ContactsContract.Data.DATA1,            ❷ Unique data
         headline);                                                for contact
      builder.withValue(ContactsContract.Data.DATA2, "LinkedIn");

      batch.add(builder.build());
      try {
         resolver.applyBatch(ContactsContract.AUTHORITY, batch);
         return true;
      } catch (Exception e) {
         return false;
      }
   }

   public static boolean updateContact(ContentResolver resolver,
         Account account, String username, String headline) {
      ArrayList<ContentProviderOperation> batch =                 ❶ Batch
         new ArrayList<ContentProviderOperation>();                 operations
      ContentProviderOperation.Builder builder = ContentProviderOperation
         .newInsert(ContactsContract.Data.CONTENT_URI);
      builder.withValue(ContactsContract.Data.RAW_CONTACT_ID, 0);
      builder.withValue(ContactsContract.Data.MIMETYPE,
         LinkedIn.MIME_TYPE);
      builder.withValue(ContactsContract.Data.DATA1,            ❷ Unique data
         headline);                                                for contact
      builder.withValue(ContactsContract.Data.DATA2, "LinkedIn");
      batch.add(builder.build());
      try {
         resolver.applyBatch(ContactsContract.AUTHORITY, batch);
         return true;
      } catch (Exception e) {
         return false;
      }
   }
}
```

Because you're performing multiple insertions at a time, we use a batch operation ❶ to wrap all the tasks involved. This operation performs more quickly and has the added benefit of protecting against partially updated records if an error occurs midway through processing.

The DATA1 field ❷ refers to custom data. Android defines many similar columns, such as DATA1, DATA2, DATA3, and so on. Each new type of contact data can decide for itself how it wants to define and interpret its data. In our example, you're placing the headline in the first available data slot. For more complex fields such as structured names, you can use the various data fields to store multiple components such as first and last names.

Now that you can create contacts, you need to tell the native Contacts app how to display LinkedIn data. This is done through a custom XML file, shown in the following listing.

Listing 15.9 contacts.xml

```
<?xml version="1.0" encoding="utf-8"?>
<ContactsSource xmlns:android="http://schemas.android.com/apk/res/android">
   <ContactsDataKind
      android:mimeType=                                         ❶ Custom
         "vnd.android.cursor.item/vnd.linkedin.profile"            content type
      android:icon="@drawable/icon"
      android:summaryColumn="data2"
      android:detailColumn="data1"
   />
</ContactsSource>
```

Google hasn't documented this file well, but fortunately there isn't too much to know. The code in ❶ helps the Contacts app find the rows in the data table that correspond to LinkedIn. The icon to display in Contacts and the summary column help users identify this type of data. Finally, the detail column describes the information unique to this contact. In our case, this is the headline.

15.4 Keeping it together

People get calls all day long on their office phones at work and on their mobile phones. They have personal friends, business contacts, and fellow alumni. They belong to Facebook, LinkedIn, and Bebo. They have dental appointments, choir recitals, and quarterly reports.

Today's mobile phone users are caught in a dilemma. On one hand, they want a single point of contact that handles all their information for them; why carry a pager, a scheduler, a phone, and a little black address book, when you can have it all in one smartphone? On the other hand, they want to compartmentalize their information. Most people like to keep their work and personal email accounts separate, and may not want friends on one social network to see friends on another network.

Android manages this tricky balancing act through the use of multiple accounts, as described in section 15.1.2. By understanding how accounts are managed and kept up-to-date, you'll be able to build powerful apps that smoothly fulfill your users' expectations.

15.4.1 The dream of sync

If you've ever changed email providers, you've probably experienced the pain of transferring your data. Copying old email manually is a cumbersome process, and transferring your address book isn't much easier.

A similar problem constantly happens today with accounts that have multiple terminals. When you send and receive Gmail from your desktop, laptop, and phone, you expect the same emails to be visible everywhere. But if you use a POP3 email program from an ISP, emails might only be seen from the computer that sent them. When the same information is available in multiple places, we say that it's synchronized. *Synchronization* refers to the process of ensuring that the most up-to-date information is available on all terminals.

Software developers have been trying to solve the synchronization problem for decades, and though they've gotten closer, it continues to be a tricky problem. A single master repository should hold the definitive view of a user's information, such as their calendar. Users generally won't directly interact with this repository; instead, they'll use other programs to create calendar events and receive event reminders. If the user creates an event in one program, it must also display on the other. Figure 15.8 shows how this process is handled for Google accounts. All devices connect to the cloud, which stores all the data. When a user creates an event on her phone, it's transferred to her remote calendar. The next time the user connects to the calendar with her desktop browser, she receives the event.

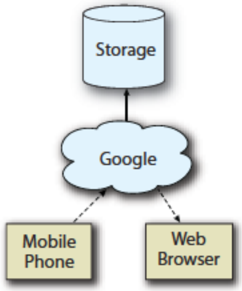


Figure 15.8 **Synchronizing data across multiple terminals**

Because mobile devices have unreliable network connections, the system needs to be extremely robust in the

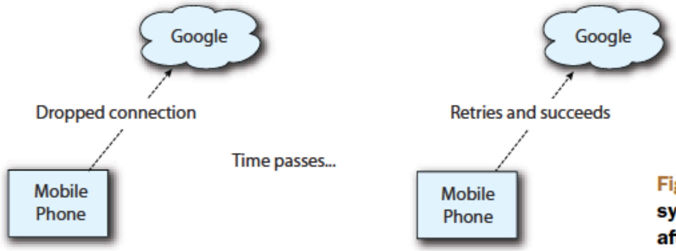


Figure 15.9 **Retrying a synchronization operation after initial failure**

event of failure. If network synchronization fails, as in figure 15.9, the device will be responsible for retrying at a later time until it succeeds. Each provider will follow its own strategy for syncing, depending on the data being transferred. For example, in some situations the provider will want to abort all transferred data if the connection is interrupted; in other cases, the provider will accept the data that was already transmitted, and wait to receive the rest later.

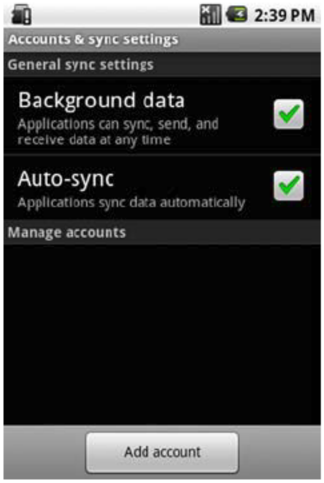
Try to keep this as transparent to the user as possible. Things should "just work": emails sent on one device should display on another without any effort involved. Because of the complexity involved in synchronization, though, there are many ways for synching to fail, which can lead to baffling results. By seeing what's happening behind the scenes, you've gained insight into detecting and handling these kinds of errors.

15.4.2 Defining accounts

Section 15.1.2 described how Android handles multiple sets of contacts by creating separate accounts for each set. Users can manually manage their contacts by visiting Accounts, which is usually available as a menu option within the native Contacts activity. Figure 15.10 shows where to add and manage accounts.

So what, exactly, is an account? An account combines your identity information, such as a login and password, with associated data, such as contacts and email.

Because accounts store their contact data separately from one another in the raw_contacts table, they can be toggled on and off without affecting unrelated information. For example, if you leave a company, you can remove all your former colleagues without worrying about destroying your personal data. You can create your own type of account by providing three components.



Background data
Applications can sync, send, and receive data at any time ☑
Auto-sync
Applications sync data automatically ☑

Figure 15.10 **Managing accounts**

SERVICE

Android will use accounts to retrieve the user's remote information while running in the background, so a service will wrap all of the account's actions.

IBINDER

The binder will service requests for this type of account, such as adding a new account, authenticating the user, or storing credentials. You can create an appropriate binder by extending `AbstractAccountAuthenticator`.

ACTIVITY

Users will need to enter their information to authenticate with their accounts. Different accounts may have differing requirements; for example, some may require the use of a key phrase, selecting a picture, or entering a soft crypto token. Because Android can't provide a generic UI to cover all possible types of authentication, each account must implement an activity that extends `AccountAuthenticatorActivity` to present the user with a login screen and handle the results. You'll see examples of all three components in section 15.5.

15.4.3 Telling secrets: The AccountManager service

Earlier versions of Android only had native support for a single Google account. Developers did write apps for other account types, such as Exchange, but these solutions were isolated and inconsistent. Each developer needed to rewrite from scratch the system for logging in to an account and managing the user's data. Even worse, there were no enforceable standards for managing users' secret data, such as passwords.

Android 2.0 fixed this problem by adding a new service called `AccountManager`. `AccountManager` provides a consistent and uniform means for interacting with all the accounts registered on a device, including adding and removing accounts or modifying their security. `AccountManager` also provides a single and secure storage facility for private user credentials; by using this service, your app is released from the burden of protecting the user's secrets.

Each account can define whatever types of information are necessary to authenticate the user. Typically, this will include the following:

- *Login name*—The unique identifier for the user, such as `name@example.com`.
- *Password*—A secret string, known to the user, which verifies that he is the owner of this account.
- *Authentication token*—A transient string, known to the account but generally not to the user, which demonstrates that the user has successfully logged in to the account. Often referred to as an *auth token*, this string frees the user from the need to manually type the password every time she wishes to connect with the account, and saves the server from performing additional, time-consuming authentications. An auth token is usually valid for a limited amount of time, and must occasionally be refreshed by performing a full login.

All of this data is stored and managed by `AccountManager`, as shown in figure 15.11. In this example, the user already has two active accounts on his device, which are currently authenticated. A third account can be added by providing the relevant account details, in this case a name and password. Once verified by `AccountManager`, this account will be accessible like the others.

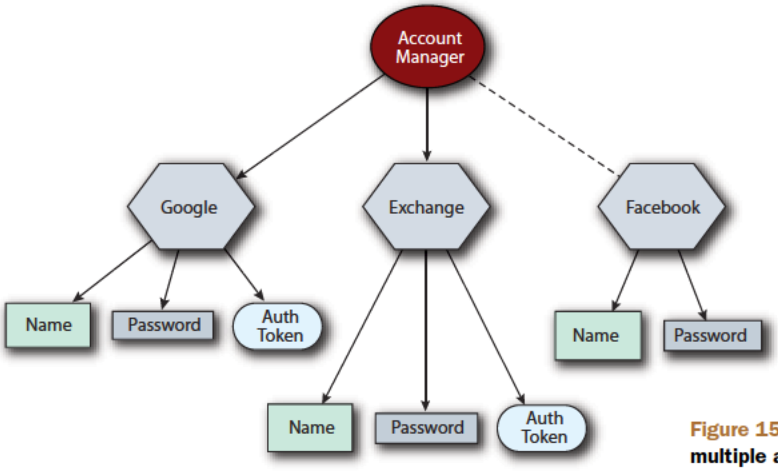


Figure 15.11 Managing multiple accounts

> **CAUTION** Many permissions are required for implementing accounts. Be sure to claim the `MANAGE_ACCOUNTS` permission if your app will be creating or modifying accounts, and the `AUTHENTICATE_ACCOUNTS` permission for managing passwords and auth tokens. Many other permissions may be required as well, including `GET_ACCOUNTS`, `USE_CREDENTIALS`, and `WRITE_SETTINGS`. Most applications that provide new accounts will eventually require all of these and more. See a complete list in the sample manifest near the end of this chapter.

15.5 Creating a LinkedIn account

From a user's perspective, connecting to LinkedIn looks simple. You just type in your email address and password, and you get access. Behind the scenes, though, a lot more is going on, particularly when connecting from a third-party application. No fewer than seven tokens are involved as part of the authentication process, and all must be handled correctly before you can start requesting data. This isn't unique to LinkedIn, and learning how to navigate the LinkedIn authentication process will equip you well for handling almost any authentication scheme.

15.5.1 Not friendly to mobile

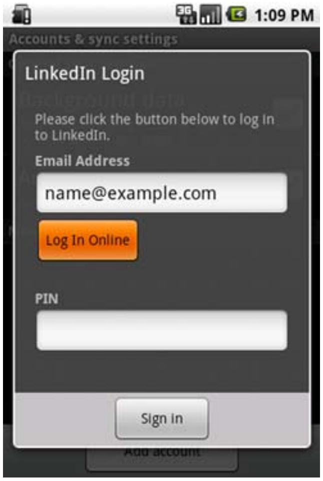
To protect against people viewing others' private information, LinkedIn requires each user of a third-party app to sign in to their account through the LinkedIn portal. Once they do, LinkedIn returns two authentication tokens that'll be included in all future requests. LinkedIn wrote its API assuming that developers would write web applications. The web app will redirect the user to the official LinkedIn login page, and LinkedIn will return the user to a URL specified by the developer once the login is complete.

But this system doesn't work well for other platforms such as desktop and mobile applications. LinkedIn has no way of passing the authentication token directly to your app, so you must jump through extra hoops to obtain this necessary information.

15.5.2 Authenticating to LinkedIn

Fortunately, Android's `AccountManager` system provides a perfect solution to circumvent this problem. You can use the custom login `Activity` to guide the user through the process of authenticating with LinkedIn, and then securely store the tokens for all future requests. LinkedIn is unusual in that it actually returns two auth tokens, both a public one and a secret version. Android's `AccountManager` is robust and extensible enough to accept arbitrary data beyond the traditional username/password/authtoken combo, and so the complexity of authenticating with LinkedIn can be hidden.

The UI will contain a few important sections. An email address will uniquely identify the account. The activity will contain a link to the LinkedIn login page where the user can remotely authenticate. After she gets her secure token, it can be entered back into the login page to finish the authentication process.



LinkedIn Login

Please click the button below to log in to LinkedIn.

Email Address: name@example.com

PIN:

Figure 15.12 Login screen for LinkedIn account

You start by defining the UI, which will look like figure 15.12. The layout XML for this listing is available online on the website for this book.

Next, you write the main `Activity` for our application, shown in the following listing. In addition to configuring the UI, it manages the two-stage process of authenticating against LinkedIn. This activity is responsible for many details; only the most important are printed here, but you can view the entire class in the download from this book's website.

Listing 15.10 Login activity initialization and authentication

```
package com.manning.unlockingandroid.linkedin.auth;
// imports omitted for brevity
public class LinkedInLoginActivity extends AccountAuthenticatorActivity {

   // Constants and most instance variables omitted for brevity
   private Boolean confirmCredentials = false;
   protected boolean createAccount = false;                  ❶ Purpose
                                                               for launch
   LinkedInOAuthService oauthService;
   private String authToken;                           ❷ Temporarily store
   private String authTokenSecret;                       challenge tokens

   @Override
   public void onCreate(Bundle icicle) {
      oauthService = LinkedInOAuthServiceFactory.getInstance()
         .createLinkedInOAuthService(LinkedIn.API_KEY,
         LinkedIn.SECRET_KEY);
      // UI code omitted for brevity
      confirmCredentials = intent.getBooleanExtra(PARAM_CREDENTIALS,
         false);
   }

   private Thread authenticate(final String userToken,
         final String userTokenSecret, final String pin) {
      Thread authenticator = new Thread() {
         public void run() {
            boolean success;
            try {
               LinkedInRequestToken requestToken = new
                  LinkedInRequestToken(userToken, userTokenSecret);
               LinkedInAccessToken access = oauthService
                  .getOAuthAccessToken(requestToken, pin);
               authToken = access.getToken();
               authTokenSecret = access.getTokenSecret();
               success = true;
            } catch (Exception e) {
               success = false;
            }
            final boolean result = success;
            handler.post(new Runnable() {
               public void run() {
                  onAuthenticationResult(result);
               }
            });
         }
      };
      authenticator.start();
      return authenticator;
   }
   // Other methods shown in following listings.
}
```

❷ Temporarily store challenge tokens

Android always shows this screen when adding a new account; additionally, it'll display if the account later fails to sync due to an expired auth token or other security error. As such, instance variables ❶ keep track of the reason for displaying the activity. Here you use OAuth variables ❷ to store the initially provided challenge tokens so you can send them to LinkedIn during the final stages of login. The linkedin-j APIs allow you to write terser and clearer code than would be possible with the raw LinkedIn APIs, which are more web services oriented.

Logging into LinkedIn requires a two-stage process: first, you acquire a PIN number when the user visits the LinkedIn website, and then you use that PIN to complete the authentication. The following listing shows the methods in our Activity that control the handling of this authentication data.

Listing 15.11 Login activity's multiple stages of logging in

```
   @Override
   protected Dialog onCreateDialog(int id) {
      final ProgressDialog dialog = new ProgressDialog(this);
      // UI code omitted for brevity.
      dialog.setOnCancelListener(new DialogInterface.OnCancelListener() {
```

```
      public void onCancel(DialogInterface dialog) {
         if (authentication != null) {
            authentication.interrupt();                      Cancel pending
            finish();                                        auth attempt
         }
      }
   });
   return dialog;
}
                                                             Initiate first
public void startLogin(View view) {                          login phase
   try {
      LinkedInRequestToken requestToken = oauthService
         .getOAuthRequestToken();
      userToken = requestToken.getToken();
      userTokenSecret = requestToken.getTokenSecret();
      String authURL = requestToken.getAuthorizationUrl();
      Intent authIntent = new Intent(Intent.ACTION_VIEW, Uri
         .parse(authURL));
      startActivity(authIntent);
   } catch (Exception ioe) {
      status.setText(R.string.start_login_error);
   }
}
                                                             Initiate second
public void finishLogin(View view) {                         login phase
   if (createAccount) {
      accountName = accountNameField.getText().toString();
   }
   enteredPIN = pinField.getText().toString();
   if (TextUtils.isEmpty(accountName) ||
      TextUtils.isEmpty(enteredPIN)) {
      status.setText(R.string.empty_fields_error);
   } else {
      showProgress();
      authentication = authenticate(userToken,               Kick off auth
         userTokenSecret, enteredPIN);                       completion
   }
}
```

After authentication has finished, you'll inspect and handle the outcome of the attempt, as shown in this listing. If it succeeds, the token will be stored for future reuse.

Listing 15.12 Login activity responding to completed login attempt

```
public void onAuthenticationResult(boolean result) {
   hideProgress();
   if (result) {
      if (!confirmCredentials) {
         finishLogin();
      } else {
         finishConfirmCredentials(true);
      }
   } else {
      if (createAccount) {
         status.setText(getText(R.string.login_fail_error));
      }
   }
}

protected void finishLogin() {
   final Account account = new Account(accountName, LinkedIn.TYPE);
   if (createAccount) {
      Bundle data = new Bundle();
      data.putString(LinkedIn.AUTH_TOKEN,
         authToken);
      data.putString(LinkedIn.AUTH_TOKEN_SECRET,
         authTokenSecret);
      accountManager.addAccountExplicitly(account,
         enteredPIN, data);
      ContentResolver.setSyncAutomatically(account,
         ContactsContract.AUTHORITY, true);
   } else {
      accountManager.setPassword(account, enteredPIN);
      accountManager.setUserData(account,
            LinkedIn.AUTH_TOKEN, authToken);
      accountManager.setUserData(account, LinkedIn.
            AUTH_TOKEN_SECRET,  authTokenSecret);
   }
   final Intent intent = new Intent();
   intent.putExtra(AccountManager.KEY_ACCOUNT_NAME, accountName);
   intent.putExtra(AccountManager.KEY_ACCOUNT_TYPE, LinkedIn.TYPE);
   setAccountAuthenticatorResult(intent.getExtras());
   setResult(RESULT_OK, intent);
   finish();
}

protected void finishConfirmCredentials(boolean result) {
   final Account account = new Account(accountName, LinkedIn.TYPE);
   accountManager.setPassword(account, enteredPIN);
   final Intent intent = new Intent();
   intent.putExtra(AccountManager.KEY_BOOLEAN_RESULT, result);
   setAccountAuthenticatorResult(intent.getExtras());
   setResult(RESULT_OK, intent);
   finish();
}
```

Server said auth was OK

1 Permanently store final auth tokens

1 Permanently store final auth tokens

The most important parts of the Activity revolve around the AccountManager, where you store the final auth tokens 1. Note that, although you store the PIN as the password field, for LinkedIn you never touch the user's actual password; the auth tokens are the important pieces.

Next comes an AccountAuthenticator, shown in listing 15.13, which handles the final stages of authenticating against LinkedIn and actually creates the Android account. Most of the actions of an authenticator are boilerplate, so only the most relevant portions are shown here. You can view the entire class online. First come two main entry points that'll be invoked by the system when adding a new account or confirming that an existing account still has valid credentials.

Listing 15.13 Authenticating against the LinkedIn account

```
package com.manning.unlockingandroid.linkedin.auth;
// imports omitted for brevity
class LinkedInAccountAuthenticator extends AbstractAccountAuthenticator {
   private final Context context;

   private LinkedInApiClientFactory factory;

   public Bundle addAccount(
         AccountAuthenticatorResponse response,
         String accountType, String authTokenType,
         String[] requiredFeatures, Bundle options) {
      Intent intent = new Intent(
         context, LinkedInLoginActivity.class);
      intent.putExtra(LinkedInLoginActivity.PARAM_AUTHTOKEN_TYPE,
         authTokenType);
      intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
         response);
      Bundle bundle = new Bundle();
      bundle.putParcelable(AccountManager.KEY_INTENT, intent);
      return bundle;
   }

   public Bundle confirmCredentials(AccountAuthenticatorResponse response,
         Account account, Bundle options) {
      if (options != null
            && options.containsKey(AccountManager.KEY_PASSWORD)) {
         String authToken = options.getString(LinkedIn.AUTH_TOKEN);
         String authSecret = options
            .getString(LinkedIn.AUTH_TOKEN_SECRET);
         boolean verified = validateAuthToken(authToken, authSecret);
         Bundle result = new Bundle();
         result.putBoolean(AccountManager.KEY_BOOLEAN_RESULT, verified);
         return result;
      }
      Intent intent = new Intent(
         context, LinkedInLoginActivity.class);
      intent.putExtra(LinkedInLoginActivity.PARAM_USERNAME, account.name);
      intent.putExtra(LinkedInLoginActivity.PARAM_CREDENTIALS, true);
      intent.putExtra(AccountManager.KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
         response);
      Bundle bundle = new Bundle();
      bundle.putParcelable(
         AccountManager.KEY_INTENT, intent);
      return bundle;
   }
   public Bundle updateCredentials(AccountAuthenticatorResponse response,
         Account account, String authTokenType, Bundle loginOptions) {
      Intent intent = new Intent(context, LinkedInLoginActivity.class);
      intent.putExtra(LinkedInLoginActivity.PARAM_USERNAME, account.name);
      intent.putExtra(LinkedInLoginActivity.PARAM_AUTHTOKEN_TYPE,
         authTokenType);
      intent.putExtra(LinkedInLoginActivity.PARAM_CREDENTIALS, false);
      Bundle bundle = new Bundle();
      bundle.putParcelable(AccountManager.KEY_INTENT, intent);
```

Explicitly specify login activity

Explicitly specify login activity

1 Verify credentials through UI

```
      return bundle;
   }
}
```

Once again, the `linkedin-j` classes help clarify the code and take care of some rote bookkeeping tasks. The authenticator will be called in various situations, including creating a new account and verifying the auth token for an existing account, and so it implements the `AbstractAccountAuthenticator` methods to support the different entry points. Note, for example, that `addAccount` ❶ is responsible for launching into the UI activity for creating a new account. Additionally, the authenticator will provide previously created tokens, as shown in the following listing.

Listing 15.14 Adding support for handling auth token

```
public Bundle getAuthToken(AccountAuthenticatorResponse response,
      Account account, String authTokenType, Bundle loginOptions) {
   // Sanity checking omitted for brevity.
   AccountManager am = AccountManager.get(context);
   String authToken = am.getUserData(account, LinkedIn.AUTH_TOKEN);
   String authTokenSecret = am.getUserData(account,
         LinkedIn.AUTH_TOKEN_SECRET);
   if (authToken != null && authTokenSecret != null) {
      boolean verified = validateAuthToken(authToken,
         authTokenSecret);
      if (verified) {
         // Return bundle omitted for brevity.
      }
   }
   Intent intent = new Intent(
      context, LinkedInLoginActivity.class);
   intent.putExtra(AccountManager.
      KEY_ACCOUNT_AUTHENTICATOR_RESPONSE,
      response);
   intent.putExtra(LinkedInLoginActivity.PARAM_USERNAME, account.name);
   intent.putExtra(LinkedInLoginActivity.PARAM_AUTHTOKEN_TYPE,
      authTokenType);
   Bundle bundle = new Bundle();
   bundle.putParcelable(AccountManager.KEY_INTENT, intent);
   return bundle;
}

private boolean validateAuthToken(String authToken,
      String authTokenSecret) {
   try {
      LinkedInApiClient client = factory.createLinkedInApiClient(
         authToken, authTokenSecret);
      client.getConnectionsForCurrentUser(0, 1);
      return true;
   } catch (Exception e) {
      return false;
   }
}
```

Ask user to re-login

❶ **Issue test network request**

Once the final tokens come back from the server, a dummy API request ❶ ensures that the connection is good. If any problems occur, notify the caller and it can take an appropriate action, such as showing a notification to the user.

Because authentication will run in the background, it must be a `Service`. The next listing shows the lightweight authentication service wrapper, which defers everything to our authenticator.

Listing 15.15 Defining the authentication service

```
package com.manning.unlockingandroid.linkedin.auth;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class LinkedInAuthService extends Service {
   private LinkedInAccountAuthenticator authenticator;

   @Override
   public void onCreate() {
      authenticator = new LinkedInAccountAuthenticator(this);
   }

   @Override
   public void onDestroy() {
   }

   @Override
   public IBinder onBind(Intent intent) {
      return authenticator.getIBinder();
   }
}
```

Finally, the little piece of XML in the next listing will tell Android how to display this type of account in the Manage Accounts screen.

Listing 15.16 authenticator.xml

```
<?xml version="1.0" encoding="utf-8"?>
<account-authenticator xmlns:android="http://schemas.android.com/apk/res/
     android"
    android:accountType=
       "com.manning.unlockingandroid.linkedin"
    android:icon="@drawable/icon"
    android:smallIcon="@drawable/icon"
    android:label="@string/app_name"
/>
```

Type advertised for synchronization

> **CAUTION** Android went to great lengths to separate account authentication and account synchronization; there's no explicit linkage between the two pieces. But early versions of the OS will react badly if you attempt to create an account that doesn't have a synchronizer. For example, the Android 2.1 emulator will crash and reboot if you successfully add an account that doesn't have a matching synchronizer. Future versions of the OS should fix this bug. In the meantime, you may want to wait a bit longer before testing your authentication code.

15.6 Synchronizing to the backend with SyncAdapter

Authenticating an account connects you to the remote server, but by itself does nothing. The real power comes from an account's ability to synchronize data onto and off of the phone. Android 2.0 added the ability to synchronize custom data from arbitrary accounts.

15.6.1 The synchronizing lifecycle

Synchronizing will generally happen in the background, similarly to authentication. The authenticator and the synchronizer are loosely coupled; the synchronizer will retrieve necessary information from the `AccountManager` instead of directly from the authenticator. Again, this is done to keep the user's private information secure.

To perform synchronization, your service should return an `IBinder` obtained from a class you define that extends `AbstractThreadedSyncAdapter`. This defines a single method, `onPerformSync`, which allows you to perform all synching activities.

> **TIP** Synchronizing operations can differ drastically, depending on what type of data you're synching. Though most accounts are oriented around personal information, an account could also be used to deliver daily recipes to an application, or to upload usage reports. The authenticator/synchronizer combo is best for situations where a password is required and you want to transfer data silently in the background. In other cases, a standard service would work better.

15.6.2 Synchronizing LinkedIn data

Now that you've written an account and utilities for our LinkedIn connections, all that remains is to tie the two together. You can accomplish this with a few final classes for synchronization. The most important is `SyncAdapter`, shown in this listing.

Listing 15.17 Synchronizing LinkedIn connections to contacts

```
package com.manning.unlockingandroid.linkedin.sync;
// Imports omitted for brevity
public class SyncAdapter extends AbstractThreadedSyncAdapter {
   private final AccountManager manager;
   private final LinkedInApiClientFactory factory;
   private final ContentResolver resolver;

   String[] idSelection = new String[] {
      ContactsContract.RawContacts.SYNC1 };
   String[] idValue = new String[1];

   public SyncAdapter(Context context, boolean autoInitialize) {
      super(context, autoInitialize);
      resolver = context.getContentResolver();
      manager = AccountManager.get(context);
      factory = LinkedInApiClientFactory.newInstance(LinkedIn.API_KEY,
            LinkedIn.SECRET_KEY);
   }
```

SQL selection to find contacts

```
   @Override
   public void onPerformSync(Account account, Bundle extras,
         String authority, ContentProviderClient provider,
         SyncResult syncResult) {
      String authToken = null;
      try {
         authToken = manager.blockingGetAuthToken(          ❶ Ensure established
               account, LinkedIn.TYPE, true);                  connection
         if (authToken == null) {
            syncResult.stats.numAuthExceptions++;
            return;
         }
         authToken = manager.getUserData(account,               ❷ Retrieve
               LinkedIn.AUTH_TOKEN);                               credentials
         String authTokenSecret = manager.getUserData(
               account, LinkedIn.AUTH_TOKEN_SECRET);
         LinkedInApiClient client = factory.createLinkedInApiClient(
               authToken, authTokenSecret);
         Connections people = client.getConnectionsForCurrentUser();
         for (Person person:people.getPersonList()) {           ❸ Examine all
            String id = person.getId();                            connections
            String firstName = person.getFirstName();
            String lastName = person.getLastName();
            String headline = person.getHeadline();
            idValue[0] = id;                                       ❹ Already
            Cursor matches = resolver.query(                          exists?
ContactsContract.RawContacts.CONTENT_URI, idSelection,
ContactsContract.RawContacts.SYNC1 + "=?", idValue,
null);
            if (matches.moveToFirst()) {
               ContactHelper.updateContact(                         ❺ Update
                  resolver, account, id, headline);                    headline
            } else {
               ContactHelper.addContact(resolver,                 ❻ Insert data
                  account, firstName + " "                           and create
                  + lastName, id, headline);                         contact
            }
         }
      } catch (AuthenticatorException e) {
         manager.invalidateAuthToken(LinkedIn.TYPE, authToken);
         syncResult.stats.numAuthExceptions++;
      } catch (IOException ioe) {
         syncResult.stats.numIoExceptions++;
      } catch (OperationCanceledException ioe) {
         syncResult.stats.numIoExceptions++;
      } catch (LinkedInApiClientException liace) {
         manager.invalidateAuthToken(LinkedIn.TYPE, authToken);
         syncResult.stats.numAuthExceptions++;
      }
   }
}
```

When performing a sync, you first verify a working auth token ❶, then retrieve the two auth tokens ❷ that are needed to interact with LinkedIn. The `linkedin-j` APIs simplify retrieving and manipulating data model objects for the user's connections.

You iterate through these models ❸, check to see whether they're already in your special LinkedIn contacts list ❹, and then add ❺ or update ❻ the contacts as appropriate, using the `ContactHelper` class from listing 15.8. Android will read the `syncResult` variable to determine if and why the sync failed; this can cause the OS to prompt the user to reauthenticate if necessary.

As with authentication, a lightweight wrapper service, shown in the next listing, manages the sync adapter.

Listing 15.18 Defining the synchronization service

```
package com.manning.unlockingandroid.linkedin.sync;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;

public class SyncService extends Service {
   private static final Object syncAdapterLock
      = new Object();
   private static SyncAdapter syncAdapter = null;

   @Override
   public void onCreate() {
      synchronized (syncAdapterLock) {
         if (syncAdapter == null) {
            syncAdapter = new SyncAdapter(getApplicationContext(), true);
         }
      }
   }

   @Override
   public IBinder onBind(Intent intent) {
      return syncAdapter.getSyncAdapterBinder();
   }
}
```

Singleton

And, last but not least, a final piece of XML is shown in the following listing to describe the synchronization service's capabilities.

Listing 15.19 syncadapter.xml

```
<?xml version="1.0" encoding="utf-8"?>
<sync-adapter xmlns:android="http://schemas.android.com/apk/res/android"
    android:contentAuthority="com.android.contacts"
    android:accountType=
       "com.manning.unlockingandroid.linkedin"
    android:supportsUploading="false"
/>
```

❶ Uses the linkedin account

The content authority tells Android what type of data can be updated by this service; contacts are by far the most common. The account type ❶ links the synchronizer to its corresponding authenticator. Finally, the XML describes whether the synchronizer supports one-way downloading only, or whether it also supports uploading changes to data.

15.7 Wrapping up: LinkedIn in action

A few final pieces of code will conclude the sample project. A well-designed account seems invisible; once configured, it'll silently and seamlessly work in the background, pulling in relevant data whenever available. We'll also discuss a few advanced topics that push the limits of integration.

15.7.1 Finalizing the LinkedIn project

You've already written all the code, so all that remains is updating your Android manifest to describe the application's capabilities. The following listing shows the final pieces.

Listing 15.20 AndroidManifest.xml for LinkedIn

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.manning.unlockingandroid.linkedin" android:versionCode="1"
  android:versionName="1.0">
  <application android:icon="@drawable/icon"
    android:label="@string/app_name">
    <service android:name=".auth.LinkedInAuthService"
      android:exported="true">
      <intent-filter>
        <action android:name=
          "android.accounts.AccountAuthenticator" />
      </intent-filter>
      <meta-data android:name="android.accounts.AccountAuthenticator"
        android:resource="@xml/authenticator" />
    </service>
    <service android:name=".sync.SyncService" android:exported="true">
      <intent-filter>
        <action android:name=
          "android.content.SyncAdapter" />
      </intent-filter>
      <meta-data android:name="android.content.SyncAdapter"
        android:resource="@xml/syncadapter" />
      <meta-data android:name="android.provider.CONTACTS_STRUCTURE"
        android:resource="@xml/contacts" />
    </service>
    <activity android:name=
      ".auth.LinkedInLoginActivity" android:label=
      "@string/login_label" android:theme=
      "@android:style/Theme.Dialog"
      android:excludeFromRecents="true">
    </activity>
  </application>
  <uses-permission android:name="android.permission.GET_ACCOUNTS" />
  <uses-permission android:name="android.permission.USE_CREDENTIALS" />
  <uses-permission android:name="android.permission.MANAGE_ACCOUNTS" />
  <uses-permission
    android:name="android.permission.AUTHENTICATE_ACCOUNTS" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.WRITE_SETTINGS" />
```

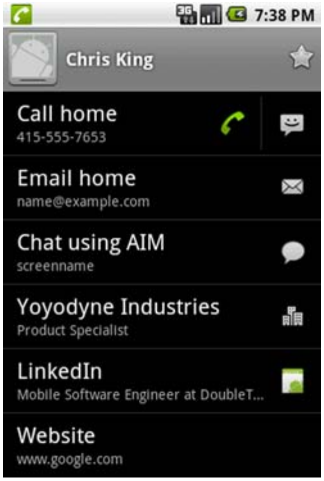
1 Handles adding accounts

2 Handles synching data

3 Private activity for logging in

```
  <uses-permission android:name="android.permission.READ_CONTACTS" />
  <uses-permission android:name="android.permission.WRITE_CONTACTS" />
  <uses-permission android:name="android.permission.READ_SYNC_STATS" />
  <uses-permission android:name="android.permission.READ_SYNC_SETTINGS" />
  <uses-permission android:name="android.permission.WRITE_SYNC_SETTINGS" />
  <uses-sdk android:minSdkVersion="5" />
</manifest>
```

You'll notice a few new `intent-filter` definitions for accounts ❶ and synchronizing ❷. These prompt Android to display your application as an available account type. You'll also notice that the manifest pulls in all of the XML that you previously wrote. Again, these instruct Android how to display custom content within other applications. Finally, note that the Activity definition ❸ doesn't include an `intent-filter`. Users shouldn't launch this activity themselves; as configured, it can only be started by our own classes that know its full class name.



7:38 PM

Chris King

Call home
415-555-7653

Email home
name@example.com

Chat using AIM
screenname

Yoyodyne Industries
Product Specialist

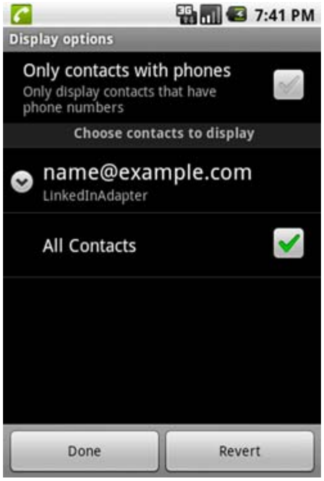
LinkedIn
Mobile Software Engineer at DoubleT...

Website
www.google.com

Figure 15.13 A contact with standard and LinkedIn data

With the manifest in place, you can now run and synchronize against LinkedIn. After you fill in the account details, your existing contacts will begin receiving their headlines, as shown in figure 15.13.

Most people have many more social network connections than they do phone numbers, so you'd expect the size of your contacts list to balloon after synchronizing with LinkedIn. By default, though, Android will only display your primary contacts; in other words, if someone is in both your phonebook and in LinkedIn, they'll be displayed, but if they only appear in LinkedIn they won't. You can change this behavior by modifying your contacts display options, as shown in figure 15.14. After making this selection, you'll see all your LinkedIn contacts.



7:41 PM

Display options

Only contacts with phones
Only display contacts that have phone numbers ☑

Choose contacts to display

name@example.com
LinkedInAdapter

All Contacts ☑

Figure 15.14 Choosing whether to display all LinkedIn contacts

15.7.2 Troubleshooting tips

Several issues may come up as you develop the LinkedIn app or other integration services. Common items include:

- If you have trouble authenticating, verify that the time on your emulator is correct. Many modern authentication schemes rely on synchronized time settings on the client and server, so if your clock is off by more than a few minutes, the login may fail.

- Because there's no main activity for this application, selecting Debug from within Eclipse won't automatically start debugging the application; it'll only push it to the emulator. To debug, switch to the DDMS tab in Eclipse, select the `com.manning.unlockingandroid.linkedin` process, and click the Debug button.
- If your process isn't already running, or you want to try some other debugging tactics, check out the AccountsTester and SyncTester activities included in the Dev Tools application. These can provide useful entry points into your application.

To solve problems with LinkedIn transactions, visit the developer forums or search for information online. Most likely you're not the only one who's encountered your problem, and someone else has found a solution. Similar resources exist for other popular types of accounts.

15.7.3 Moving on

Simple can be beautiful. For most users, getting a little of the most relevant LinkedIn information gives them want they want without cluttering up their contact screen. If you're looking for more, though, consider the following ideas:

- Pull in more advanced LinkedIn fields, such as educational background and previous job positions.
- Tie in contacts data with the Field Service application. For example, you could create a contact when you receive a new job from a customer, and remove that contact once the job is marked complete.
- In addition to the previous, create a new raw_contacts type for a field service contact; this will allow users to toggle such contacts on and off as they wish.
- Create a new account type and synchronizer for connecting with your own favorite online service.

15.8 Summary

Congratulations! You've written a fully functional, and fully useful, application that seamlessly brings your LinkedIn connections to Android. Even better, you've experienced every piece of the integration puzzle. Android makes no assumptions about what contacts will look like, and offers a pluggable, extensible contacts model that lets developers add their own custom data into the native contacts experience. Users can synchronize their contacts remotely, and Android now supports an arbitrary number of different types of accounts, each of which can follow its own custom authentication scheme and still take advantage of secure storage for its credentials. Synchronization will run in the background, silently delivering useful information to the user.

Integration shows Android's design principles at their finest. Most modern Android phones have built-in support for accounts such as Microsoft Exchange, but as useful as these are, it'd be limiting to restrict users to approved account types. Instead, Android's open architecture has created an ecosystem where any type of account, even from a small startup, can enjoy equal standing on the phone. The best compliment you can pay to this kind of software is to say that it's unnoticeable; great integration apps fade into the background, becoming an essential part of the user's daily experience.

Now that you've integrated with the device's internal store of contacts, it's time to turn your focus outward, toward the web browser. Android has an extremely robust browser that supports unusually rich user experiences. The next chapter will help you evaluate your strategies for developing content that'll look great in the Android browser.

16 Android web development

This chapter covers

- Understanding Android web development
- Introducing WebKit
- Optimizing web applications for Android
- Storing data in the browser
- Building a custom JavaScript handler

Mobile software development has usually required working with a proprietary toolset to build applications. The term *SDK* is a familiar one—the majority of this book covers Android software development using the Android SDK. But the Android SDK isn't the only way to build and deliver applications for the Android platform. With the emergence of accessible and powerful mobile browser capabilities, it's now possible to develop applications for the Android platform with web technologies such as HTML, CSS, and JavaScript.[1]

Opening up the world of mobile development to web technologies introduces a wide array of possibilities for developers and Android users alike. Mobile devices

[1] See *Hello! HTML5 and CSS3* at http://manning.com/crowther. The book is to be published in spring 2011.

rich in user experience (including Android) are in essence shrinking our world as internet connectivity is becoming the norm rather than the exception. Being connected to the internet allows Android users to share and access information from virtually anywhere, at any time. In some ways, the web itself is being redefined as an increasing percentage of content consumption and creation is taking place on mobile devices, with Android powering that growth as it reaches new heights of adoption and market share.

This chapter aims to equip you with an understanding of various approaches to deploying web technologies so that you can deliver an enhanced Android user experience. We start by surveying the major options for Android web technology development, all of which rely on the WebKit open source browser engine. After a brief introduction to WebKit, we look at creating universal web applications—apps that run well on the desktop as well as the Android browser. From there we move on to demonstrating the use of the SQL capabilities available in the browser, commonly referred to as HTML 5 databases. Note that although the browser SQL functionality was originally part of HTML 5, it has since been extracted from the core HTML 5 specification.

The chapter concludes with an example of building a "hybrid" application—one that uses Android SDK Java code along with browser-based HTML and JavaScript code.

16.1 What's Android web development?

Aside from a brief sojourn down the native C path in chapter 13, all of the coding to this point has employed the Java-based Android SDK. This chapter breaks from that mold and demonstrates various web programming capabilities of the Android platform.

In short, web development is all about building applications with the traditional tools that web developers use: HTML for content, CSS for presentation, and JavaScript for programmatic control. In order for this capable and complementary trio to work their magic, the Android platform relies on the WebKit browser engine.

16.1.1 Introducing WebKit

The WebKit browser engine stems from an open source project that can be traced back to the K Desktop Environment (KDE). WebKit made its significant mobile debut when the iPhone was released, and since then WebKit has been adopted by Android and other mobile platforms.

Prior to the adoption of WebKit, early mobile web solutions ranged from laughable, to mediocre, to tolerable, though always limited. These early mobile web offerings were often so constrained that they required content providers to generate a mobile-specific stream in addition to the normal desktop version of their material. In some cases, a server-side component would perform on-the-fly distillation of the HTML into a format more readily digested by the mobile browser. Regardless of the implementation, any requirement for content providers to generate multiple copies of their material severely constrained the volume of content the early mobile devices could readily consume. The early mobile web was virtually nonexistent because browsers were not capable of rendering full pages and sites made for mobile were rare.

Fortunately, WebKit has changed the game thanks to its impressive rendering capabilities and its envelope-pushing feature set. You can expect the WebKit engine to render any web page on a par with your desktop browser. This means that virtually the entire web is open and accessible to an Android user! The pages of your favorite website will render on your Android device's browser, though you'll likely need to scroll the page due to the small screen dimensions, and certain navigation systems that rely on hovering aren't accessible. Despite these drawbacks, the capabilities of WebKit open the broad range of the web to mobile users. In this chapter we demonstrate how to scale your web applications to accommodate for smaller browser windows in a manner that retains desktop browsing compatibility, all without the necessity of creating and managing multiple sites.

WebKit powers the browser on the Android device, but it's also available as an embedded control or widget, permitting SDK-based applications to render HTML directly within a compiled application. This embeddable browser control is highly customizable and thereby empowers the Android developer to exercise a tremendous amount of control over the user experience.

Web programming for Android is a broad and versatile topic. In the next section we examine the approaches to using web technologies for Android application development.

16.1.2 Examining the architectural options

When it comes to employing web technologies in an Android application, you have to examine a few distinct categories of application architecture. Let's look at the pillars of Android web technologies.

The first and most basic intersection of web technologies and Android application development involves the standalone browser. The Android browser is a capable HTML and CSS rendering engine, and it also implements a JavaScript engine. The JavaScript engine is capable of running sophisticated JavaScript, including Ajax, and supports popular scripting libraries such as JQuery and Prototype. As such, the browser itself is capable of running rich internet applications.

The browser can be a good augmentation to an SDK-based application. Let's say you've released a software application for distribution. As part of your application you'd like to register users or perhaps provide access to a list of FAQs. It doesn't make sense to ship all that content with your application because it'll both take up space unnecessarily and will likely be out of date by the time the application is installed on a client device. Likewise, why should you implement data-collection functionality directly in your application if it can be more readily accomplished on your website? In this case, it'd be more appropriate to have the application launch the browser, taking the user to the website, where you can readily maintain your list of FAQs and your registration form. As you've learned, an Android application launches the browser through the use of an `Intent` and the `startActivity()` method of the `Context` class.

A variant of this model is embedding the browser control directly into an `Activity`'s UI. In this scenario, the browser control is defined in a layout file and inflated at runtime. The control is manipulated directly by Java code and directed to render either locally provided content or to navigate to a location on the web. An advantage of this approach is that although users may visit a remote site, they haven't actually left the application. This approach helps in creating a highly scripted experience for the user. If users are taken out of an application, there's a chance they won't return.

A further refinement of the embedded browser experience is the practice of extending the JavaScript environment, thereby permitting the boundary between in-the-browser JavaScript and SDK Java code to be breached, to the benefit of the user experience. Further, the hybrid application can exercise control over which pages are shown and how the browser reacts to events such as bad digital certificates or window opening requests.

Whether your objective is to leverage your web development skills to bring Android applications to market or to enhance your SDK application with browser-based capabilities, the options are plentiful.

It's time to expand on this introduction of WebKit and demonstrate web technologies in action on the Android platform. The next section explores ways in which you can design a traditional web application running in the standalone browser to accommodate Android clients.

16.2 Optimizing web applications for Android

We start this discussion by considering how to code web applications so they're viewable both by desktop clients and by mobile platforms such as Android. Developing web applications for Android can be viewed as a continuum of options. On one end is a site created for universal access, meaning that it'll be visited by both desktop and mobile users. On the other end of the spectrum is a website designed specifically for mobile users. Between these two extremes are a couple of techniques for improving the user experience. We'll use the term *mobile in mind*—this refers to a website that's not exclusively written for mobile users but expects them as part of the regular visitor list. Let's begin with a discussion of designing your site with mobile in mind.

16.2.1 Designing with mobile in mind

There are millions of websites, but only a small percentage were created with mobile devices in mind—in fact, many sites were launched prior to the availability of a "mobile browser." Fortunately, the browser in Android is capable of rendering complex web content—even more powerful than any of the early desktop browsers for which those early sites were designed.

When designing a universal website—a site that's to be consumed by desktop and mobile users alike—the key concept is to frequently check your *boundary conditions*. For example, are you considering fly-out menus that rely on hovering with a mouse? That approach is a nonstarter for an Android user; there's no mouse with which to hover. And unless you're a giant search engine provider, you want to avoid coding your site to the least common denominator approach of a single field per page. That might get you by for a while on a mobile browser, but your desktop users will be both confused and annoyed. You need to meet the needs of both sets of users concurrently. You may be starting from a position that creating two sites—one for desktop and one for mobile—is out of your reach from a budgetary perspective. We'll come back to the multiple-site approach later, but for now let's design with mobile in mind.

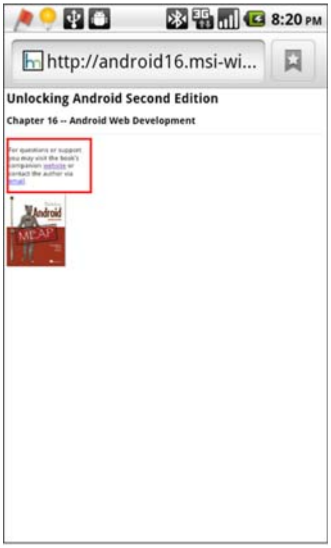
To meet that objective, we examine two approaches to improve the visual appearance and usability of a mobile-in-mind website. Start with a simple HTML page, shown in the following listing.

Listing 16.1 Sample HTML page

```
<html>
<head>
</head>
<body>
<h1>Unlocking Android Second Edition</h1>
<h2>Chapter 16 -- Android Web Development</h2>

<hr />
<div style="width:200px;border:solid 5px red;">
<p>For questions or support you may visit the book's companion <a
href="http://manning.com/ableson2">website</a> or contact the author via
<a href="mailto:fableson@msiservices.com">email</a>.</p>

</div>
<img src="http://manning.com/ableson2/ableson2_cover150.jpg" /></body>
</html>
```

When this HTML page is rendered in the browser, the content is "zoomed out" such that it all fits onto the page. Go ahead; try it yourself by pointing your Android browser to http://android16.msi-wireless.com/index.php. Figure 16.1 shows the content rendered in the standalone browser on an Android device.

The text is too small to easily be read on the phone. The user can of course pinch, zoom, and scroll the content to make it more easily consumed. But if you know that the site visitor is viewing your site on a mobile device, wouldn't it be a good idea to put out the welcome mat for them, keeping their pinching and zooming to a minimum? Fortunately, there's a simple means of modifying the visual appearance of your site so that when visitors arrive at your site via their Android device, you can make them feel like you were expecting them. To accomplish this simple but important task, you use the viewport meta tag.



Unlocking Android Second Edition

Chapter 16 -- Android Web Development

For questions or support you may visit the book's companion website or contact the author via email.

Figure 16.1 Simple web page

16.2.2 Adding the viewport tag

The lowest-cost and least-obtrusive solution to the default view being too small to see is the use of a special meta tag. Meta tags have long been the domain of the search engine optimization (SEO) gurus.[2] A meta tag is placed within the `<head></head>` tags of a web page to specify such things as the page keywords and description—which are used by search engines to help index the site.

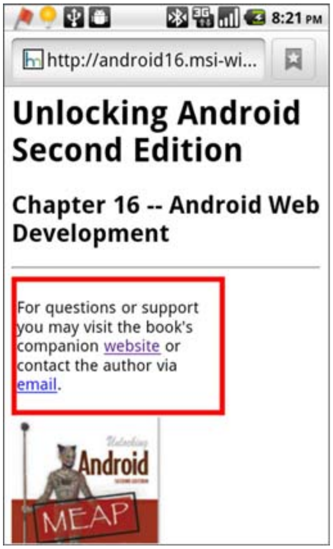
In this case, the meta tag of interest is the *viewport*. A viewport tag tells the client browser how to craft a virtual window, or viewport, through which to view the website. The viewport tag supports a handful of directives that govern the way in which the browser renders the page and interacts with the user.

To provide a more appealing rendering of our sample HTML page, you'll add a viewport tag between the head tags. Listing 16.2 shows the same web page, but it now includes the viewport meta tag. If you want to view the page on your own Android device, you can do so at http://android16.msi-wireless.com/index_view.php.

Listing 16.2 Adding the viewport meta tag

```
<html>
<head>
<meta name="viewport" content="width=device-width" />
</head>
<body>
// omitted html text
</body>
</html>
```

❶ **Viewport meta tag**



8:21 PM

http://android16.msi-wi...

Unlocking Android Second Edition

Chapter 16 -- Android Web Development

For questions or support you may visit the book's companion website or contact the author via email.

Figure 16.2 The viewport tag modifies the appearance of the web page.

This web page has been made more mobile friendly by the addition of the viewport meta tag ❶. The `content` attribute of the tag conveys directives to govern how the viewport should behave. In this case, the browser is instructed to create a viewport with a logical width equal to the screen width of the device.

Figure 16.2 demonstrates the impact this one line of code has on the visual appearance of the web page. Note how the text of the page is larger and more accessible. To be fair, this is an ultrasimple example, but the point is that you can provide a readable page right from the start and the user can easily scroll down vertically to view the remainder of the site without needing to zoom in or out just to make out what the page says.

You can specify the width in pixels rather than requesting the width to be equal to the `device-width`. This approach can be useful if you want to display a graphic in a certain manner or if your site can

[2] See http://searchenginewatch.com for everything SEO related.

remember a user's preferences and by default set up the logical dimensions according to the user's liking. Table 16.1 describes the ways in which you can customize the viewport.

Table 16.1 Viewport meta tag options

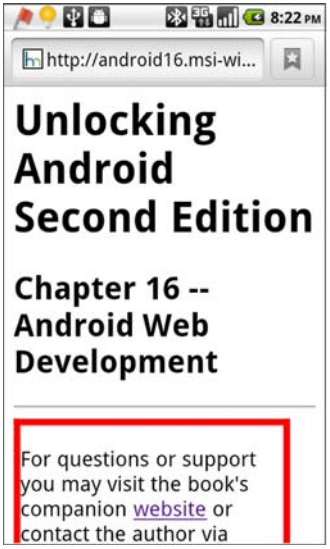
Directive or attribute	Comment
`width`	Used to specify the width of the logical viewport. Recommended value: `device-width`.
`height`	Used to specify the height of the logical viewport. Recommended value: `device-height`.
`initial-scale`	Multiplier used to scale the content up (or down) when the page is initially rendered.
`user-scalable`	Specifies whether the user is permitted to scale the screen via the pinch zoom gesture. Value: `yes` or `no`.
`maximum-scale`	Upper limit on how far a page may be scaled manually by the user. Maximum value is 10.0.
`minimum-scale`	Lower limit of how far a page may be scaled manually by the user. Minimum value is 0.5.

Adding a viewport meta tag to a web page is safe, because any meta tags that aren't supported by a client browser are ignored, with no impact on the page. This one tag provides a simple yet useful enhancement to an existing website. Although this isn't a magic bullet to solve every challenge associated with viewing a website on an Android phone, it does aid in the first impression, which is important.

Before moving on, we have one additional feature of the viewport tag to demonstrate: scaling. Figure 16.3 shows the same web page scaled to 1.3 times the original size. This approach can be used to scale a page up or down, within the constraints defined by the `minimum-scale` and `maximum-scale` directives as described in table 16.1.

This scaled-up web page may or may not provide your desired effect. The good news is that you can adjust the `initial-scale` value to your liking. In practice you'll likely set the value to somewhere between 0.7 and 1.3.

The viewport tag is almost a freebie: add the tag and if the browser recognizes it, the page's rendering will be modified and likely improved. You can take a bit more control than this by selectively loading content or style sheets based on the type of browser visiting your site. That's what we explore next.



Unlocking Android Second Edition

Chapter 16 -- Android Web Development

For questions or support you may visit the book's companion website or contact the author via

Figure 16.3 **Scaled-up web page**

16.2.3 Selectively loading content

Assuming your budget doesn't provide for creating and managing entirely parallel websites to meet the needs of your desktop and mobile visitors, you need a strategy for adequately delivering quality content to both types of users. To go beyond the functionality of the viewport tag, you want to have a more predictable experience for your mobile visitors. To accomplish this, you're going to selectively load CSS based on the type of visitor to your site.

Browser detection approaches have matured over time. Next we explore two basic methods to accomplish this task, keeping in mind that your site is meant to be universal and expecting browsers of different shapes, sizes, and capabilities.

16.2.4 Interrogating the user agent

The first approach involves the examination of the user agent string. Every time a browser connects to a web server, one of the pieces of information it provides in the HTTP request is the browser's user agent. The user agent is a string value representing the name of the browser, the platform, the version number, and other characteristics of the browser. For example, the user agent of a Nexus One running Android 2.2 looks like this:

```
Mozilla/5.0 (Linux; U; Android 2.2; en-us;Nexus One Build/FRF91) AppleWebKit/
     533.1 KHTML, like Gecko) Version/4.0 Mobile Safari/533.1
```

The contents of this string may be examined and subsequently used to decide which content and or CSS files to load into the browser. The following listing demonstrates the use of the user agent string to selectively choose a CSS file.

Listing 16.3 Interrogating the user agent

```
<html>
<head>
<meta name="viewport" content="width=device-width" />
<link rel="stylesheet" href="corestuff.css" type="text/css" />      Load core CSS  1
<script type="text/javascript" src="jquery.js"></script>

<script type="text/javascript">
   if (navigator.userAgent.indexOf('Android') != -1) {         2  Look for Android userAgent
      document.write('<link rel="stylesheet" href="android.css"
type="text/css" />');
   } else {
      document.write('<link rel="stylesheet" href="desktop.css"
type="text/css" />');
   }
</script>
</head>
<body>
...
</body>
</html>
```

This HTML snippet includes a viewport meta tag, specifying that the viewport's width should match the width of the device. A CSS file is included named corestuff.css ❶. Regardless of the platform, this file contains required classes and styles related to the application. Using this approach, the web application includes a style sheet aimed at more than one target platform. This enables you to have a more deterministic impact on the behavior on a particular device, leaving less to chance. Your primary content hasn't changed—remember, you're still targeting a universal website but keeping mobile users in mind. Clearly, more work is being done here, as various stylistic elements have been extracted into platform-specific files. As such you're taking taken a measured step down the continuum toward a made-for-mobile site. If the user agent string contains the word "Android" ❷, the code loads the user-supplied android.css style sheet. If the user agent isn't from an Android device, the code loads a style sheet named desktop.css. Additional conditional statements may be included here for other mobile platforms, such as the iPhone or BlackBerry.

User agent strings contain a considerable amount of information, though just how much of it is useful and trustworthy is a topic of debate. For example, it's not uncommon for a hacker to write code to programmatically bombard a target website and in the process craft a user agent that masquerades as a particular kind of browser. Some versions of websites are more secure than others. A user agent value is easy to forge, and although the security implications aren't expressly of concern to us in this discussion, it's something to keep in mind.

The user string has so much data that you have to do a bit of homework to interpret it properly. Some JavaScript libraries can aid in this process, but ultimately it may not be the best approach. There's another way: the media query.

16.2.5 The media query

Early web styling included inline markup such as font and bold tags. The best practice in web design today is to separate styling information from the content itself. This involves adding class attributes to elements of content and relying on a style sheet to dictate the specific colors, sizes, font face, and so on. Providing multiple CSS files for a given page enables flexible management of the numerous styles needed to deliver a crisp, clean web page. This approach also permits flexibility in terms of which styles are used for a particular device.

Professional content sites have leveraged this approach through the use of multiple, targeted style sheets along with extensive use of the `link` statement's `media` attribute. The `media` attribute acts as a filter for determining which style sheets should be loaded. For example, consider the familiar scenario where you purchase something online and you're shown your receipt. The page may have fancy graphics and multiple elements organized in a creative manner. But when the page is printed, the output is relatively benign and thankfully easy to read. This is accomplished through the use of the `media` attribute applying a print-oriented style sheet. You can leverage this same approach to build mobile-in-mind web pages.

The following listing presents a snippet from a web page with support for many style sheets, including two for mobile-specific use.

Listing 16.4 Sample link statements

```
<link href="//sitename/css/all.css" media="all"
     rel="stylesheet" type="text/css" />
<link href="//sitename/css/screen.css"
     media="screen,projection"
     rel="stylesheet" type="text/css" />
<link href="//sitename/css/screenfonts.css"
     media="screen,projection" rel="stylesheet"
     type="text/css" />
<link href="//sitename/css/print.css"
     media="print" rel="stylesheet" type="text/css" />
<link href="//sitename/css/handheld.css"
     media="handheld" rel="stylesheet" type="text/css" />
<link href="//sitename/css/handheld-small.css"
     media="only screen and (max-device-width:320px)"
     rel="stylesheet" type="text/css" />
```

❶ **all.css**

❷ **Screen filter**

❸ **Fonts only**

❹ **Print only**

❺ **Handheld device CSS**

❻ **Media query based on screen size**

The media query of `all` ❶ indicates that the associated style sheet (all.css) is appropriate for all devices. The screen.css file ❷ is filtered for screen or projectors only. An additional screen- or projector-targeted style sheet named screenfonts.css ❸ is included to organize all font-related styles. For hard-copy output, the media value of `print` is used ❹. The media value of `handheld` ❺ is intended for handheld devices, though when the media query specifications were first drafted, the capabilities of mobile browsers were quite limited—certainly much less feature rich than the Android browser is today. Therefore, a better approach is to use a media query related to specific attributes such as screen dimensions. For example, if you're targeting a particularly small device, you can use a specific attribute-oriented media query. The handheld-small.css file ❻ will be used when you have a screen width of no more than 320 pixels.

As with all things browser related, your mileage may vary over time with different releases. There's no substitute for regression testing.

The actual technique of employing CSS within your code is beyond our scope and interest here; you can find many good references on the topic. The takeaway from this discussion is to be prepared to employ the appropriate CSS for each visitor to the site based on their respective web browser capabilities. The media query is a powerful tool in your arsenal.

Of course, no matter the amount of effort you put into making your universal website mobile friendly, there are times when a site should simply be designed from the ground up for mobile users.

16.2.6 Considering a made-for-mobile application

Here we are, finally at the other end of the spectrum where we look at web applications that are designed explicitly for mobile devices. After all, the best mobile web applications are designed to be mobile applications from the start and aren't simply the full application versions crammed onto a smaller screen, relying on the browser to render them. The Android browser will display the pages, but the full-blown websites are often too heavy and busy for the typical mobile user. The reason is simple: using a mobile device is a different experience than sitting at your desk with a mouse and full keyboard.

More often than not the Android user is on the move and has little time or patience for dealing with data entry-intensive UIs or sifting through large result sets. Mobile transactions need to be thought out and targeted to the mobile user profile. Pretend you're standing on a train looking at a piece of content or making a status update to a service ticket. If you have to select multiple menus or enter too many fields to perform your work, it likely won't get done.

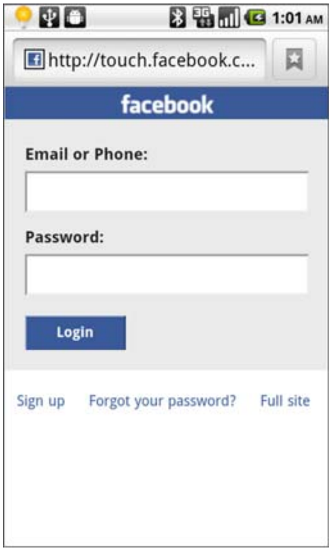


Figure 16.4 **Facebook mobile**

Consider two axioms for made-for-mobile applications.

The first is to simplify, reduce, and eliminate. Simplify the UI. Reduce the data entry required. Eliminate anything that's not needed. Seriously; pull out the scalpel and cut out things that don't matter to someone in the field. Consider figure 16.4, which shows the mobile version of the Facebook application. There's no nonsense here, just two fields: Email or Phone and Password, and a button to log in. Three links round out the page.

The second axiom is to provide a link to the full site and make sure that you don't reroute the user to the mobile version if they've explicitly requested to go to the main page. Note the Full Site link in the lower-right corner of figure 16.4. Sometimes people have the time and need to dig deeper than the mobile version permits. When this occurs, let the user go to the main page and do whatever you can through the viewport tag and the media queries to make the site as palatable as possible, but with the full features.

It's time to move beyond the browser-only visual aspects of web application development to consider more advanced techniques that Android developers have at their disposal. We look next at browser database management technology, which has the promise to take web applications, mobile and desktop, to unprecedented levels of functionality and utility.

16.3 Storing data directly in the browser

One of the historical challenges to web applications is the lack of locally stored data. When performing a frequent lookup, it's often too time- and bandwidth-intensive to constantly fetch reference rows from a server-side database. The availability of a local-to-the-browser SQL database brings new possibilities to web applications, mobile or otherwise. Support for SQL databases varies across browsers and versions, but fortunately for us, the Android browser supports this functionality. Once again the WebKit engine relationship pays dividends as we demonstrate using the desktop version of the browser to debug our application. It's mobile development, but that doesn't mean you're constrained to working exclusively on the device! The sample application used in this portion of the chapter illustrates the basics of working with a locally stored SQL database.

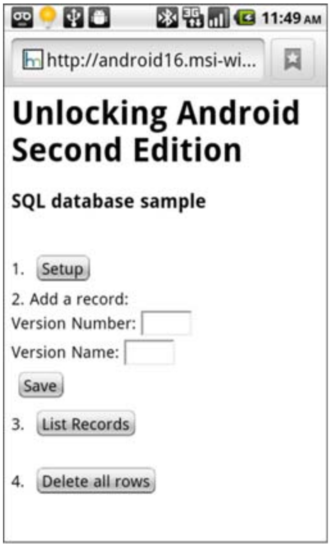
16.3.1 Setting things up

The local SQL database accessible through the Android browser is essentially a wrapper around SQLite. As such, any syntactical or data type questions can be satisfied by referring to the documentation for SQLite. To learn more about the underlying database technology, refer to the discussion in chapter 6 and or visit the SQLite website at http://sqlite.org.

For this application we're managing a single table of information with two columns. Each row represents a version of the Android operating system releases. A simple web application is used to exercise the database functionality. Figure 16.5 shows the application screen when first loaded.

The sample application, which we present in the next section, is accessible on the web at http://android16.msi-wireless.com/db.php. Before diving into the code, let's walk through the operation of the application.

Running the application is straightforward. The first thing to do is click the Setup button. This attempts to open the database. If the database doesn't exist, it's created. Once the database is opened, you can add records one at a time by populating the two text fields and clicking the Save button. Figure 16.6 shows the process of adding a record.



Unlocking Android Second Edition

SQL database sample

1. Setup

2. Add a record:

Version Number:

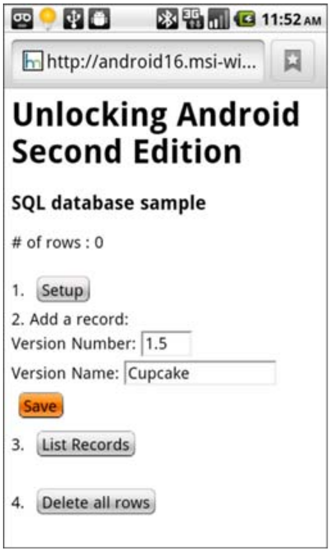
Version Name:

Save

3. List Records

4. Delete all rows

Figure 16.5 The sample SQL application



Unlocking Android Second Edition

SQL database sample

# of rows : 0

1. Setup

2. Add a record:

Version Number: 1.5

Version Name: Cupcake

Save

3. List Records

4. Delete all rows

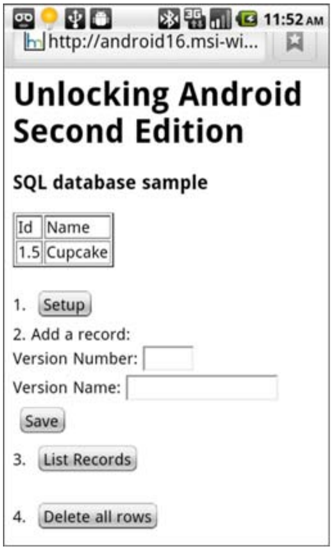
Figure 16.6 Saving a new record

The List Records button queries the database and displays the rows in a crudely formatted table. Figure 16.7 shows the result of our query after a single entry.

The final piece of demonstrable functionality is the option to remove all records. Clicking the Delete All Rows button opens the prompt shown in figure 16.8. If you confirm, the application proceeds to remove all the records from the database.

Remember, all of this is happening inside the browser without any interaction with the server side beyond the initial download of the page. In fact, there's no database on the server! If 10 people all hit the site, download this page, and add records, they'll be working independently with independently stored databases on their respective devices.

Let's look at the code for this application.



Unlocking Android Second Edition

SQL database sample

Id	Name
1.5	Cupcake

1. Setup

2. Add a record:

Version Number:

Version Name:

3. List Records

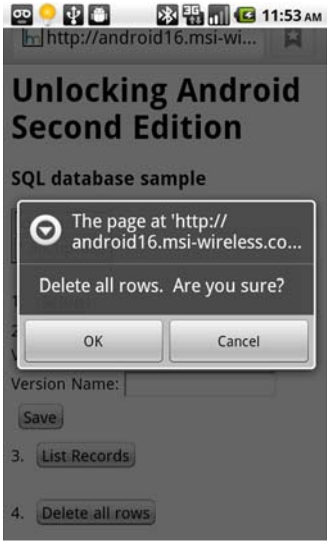
4. Delete all rows

Figure 16.7 Listing the records from the table

16.3.2 Examining the code

Working with a SQL database within the browser environment involves the use of some nontrivial JavaScript. If you're not comfortable working in the flexible JavaScript[3] language, the code may be difficult to follow at first. Stick with it—the language becomes easier as you let the code sink in over time. One helpful hint is to work with a text editor that highlights opening and closing braces and brackets. Unlike a compiled Android SDK application where the compiler points out coding problems during the development process, JavaScript errors are found at runtime. Anomalies occur and you have to track down the offending areas of your code through an iterative process of cat and mouse.

Let's begin by examining the UI elements of this application.



Unlocking Android Second Edition

SQL database sample

The page at 'http:// android16.msi-wireless.co...

Delete all rows. Are you sure?

Version Name:

3. List Records

4. Delete all rows

Figure 16.8 Confirming deletion of records

16.3.3 The user interface

We break down the code into two sections. The following listing contains the code for the UI of the application, stored in db.html.

[3] For more on JavaScript, take a look at *Secrets of the JavaScript Ninja* at http://www.manning.com/resig. The book, by John Resig, is to be published by Manning in June 2011.

Listing 16.5 User interface elements of the SQL sample page in db.html

```
<html>                                                        Viewport meta tag ❶
<head>
<meta name="viewport" content="width=device-width" />
<script src="db.js" type="text/javascript" ></script>              Reference
</head>                                                          ❷ JavaScript file
<body>
<h1>Unlocking Android Second Edition</h1>
<h3>SQL database sample</h3>                               ❸ Output div
<div id="outputarea"></div><br/>

1.&nbsp;<button onclick="setup();">Setup</button>               ❹ Call setup()
<br />
2.&nbsp;Add a record:<br/>
Version Number:&nbsp;<input id="id-field"                        ❺ Gather
     maxlength="50" style="width:50px" /><br/>                      required
Version Name:&nbsp;<input id="name-field"                           data
     maxlength="50" style="width:150px" /><br/>
<button onClick="saveRecord(document.getElementById(              ❻ Save
     'id-field').value,document.getElementById(                     record
     'name-field').value);">Save</button>
<br/>
3.&nbsp;<button onclick="document.getElementById(                 ❼ List all
     'outputarea').innerHTML = listRecords();">                      records
     List Records</button><br/>
<br />
4.&nbsp;<button onclick="if (confirm('Delete all rows.  Are you sure?') )
     {deleteAllRecords();}">Delete all rows</button>
<br />                                                             Delete
</body>                                                          all rows ❽
</html>
```

The db.html file presents a simple GUI. This page runs equally well in either the Android browser or the desktop WebKit browser. It's coded with mobile in mind, and as such includes the viewport meta tag ❶. All of the database interaction JavaScript is stored externally in a file named db.js. A script tag ❷ includes that code in the page. A `div` element with an ID of `outputarea` ❸ is used for displaying information to the user. In a production-ready application, this area would be more granularly defined and styled. Clicking the Setup button ❹ calls the JavaScript function named `setup` found in db.js. Ordinarily this kind of setup or initialization function would be called from an HTML page's `onload` handler. This step was left as an explicit operation to aid in bringing attention to all the moving pieces related to this code sample. We look more deeply at these JavaScript functions in the next section, so sit tight while we finish up the GUI aspects.

Two text fields are used to gather information when adding a new record ❺. When the Save button is clicked ❻, the `saveRecord` function is invoked. Listing the records is accomplished by clicking the List Records button ❼. Deleting all the records in the database is initiated by clicking the Delete All Records button ❽, which in turn invokes the `deleteAllRecords` function found in db.js.

With the basic GUI explanation behind us, let's examine the functions found in db.js, which provide all of the heavy lifting for the application.

16.3.4 Opening the database

Now it's time to jump into the db.js file to see how the interactions take place with the browser-based SQL database. The code for opening the database and creating the table is found in the following listing.

Listing 16.6 Code that opens the database and creates the table

```
var db;                                          ❶ db handle
var entryCount;                                           entryCount
var ret;                                                ❷ variable
entryCount = 0;

function setup()  {
try {                                                          Open database ❸
    db = window.openDatabase('ua2',1.0,'unlocking android 2E',1024);
    db.transaction(function (tx) {
      tx.executeSql("select count(*) as howmany from versions",   Execute SQL
      [],                                                       ❹ transaction
        function(tx,result) {                             ❺ Result handler
        entryCount = result.rows.item(0)['howmany'];
        document.getElementById('outputarea').innerHTML = "# of rows : " +
     entryCount;
        },                                              ❻ Error handler
      function(tx,error) {
        alert("No database exists?  Let's create it.");
        createTable();                                         Handle error by
      });});                                                 ❼ creating table
    } catch (e) {alert (e);}
}

function createTable() {                                           Execute ❽
    try {                                                     create table
        db.transaction(function (tx) {
          tx.executeSql("create table versions(id TEXT,codename TEXT)",
          [],
          function(tx,result) {
          },
          function(tx,error) {
            alert("Error attempting to create the database" + error.message);
          });});

    } catch (e) { alert (e); }
}
```

All interactions with the database require a *handle*, or variable, representing an open database. In this code, the variable named db ❶ is defined and used for this purpose. A variable named `entryCount` ❷ is used to keep track of and display the number of records currently in the database. This variable isn't essential to the operation of the code, but it's a helpful tool during development. In the setup function, the variable db is initialized with a call to the `openDatabase` function ❸. The arguments to the openDatabase function include a name, version, description, and initial size allocation of the database. If the database exists, a valid handle is returned. If the database isn't yet in existence, it's created and a handle returned to the newly created database. Calling the transaction ❹ method of the database object invokes a piece of SQL code.

The mechanics of the transaction method are nontrivial and are described in detail in section 16.3.5. For now understand that the argument to the transaction method is a function that has four arguments: a SQL statement, parameters, a callback function for handling a successful invocation of the SQL statement, and an error-handling function invoked when an error occurs processing the SQL. The SQL statement invoked here attempts to get a count of the rows in the table named versions. This value is stored in the entryCount variable ❺. If the table doesn't exist, an error is thrown ❻. This is your cue to go ahead and create the table with a call to a user-supplied function named createTable ❼. The createTable function executes a single piece of SQL to create a table ❽. This method could be used to do any number of database initialization activities, such as creating multiple tables and populating each with default values.

Before we go through the balance of the transactions, it's important to grasp how the transaction method of the database object is wired.

16.3.5 Unpacking the transaction function

All interactions with the database involve using the transaction method of the database object, so it's important to understand how to interact with each of the four arguments introduced previously.

The first argument is a parameterized SQL statement. This is simply a string with any parameterized values replaced with a question mark (?). For example, consider a SQL statement that selectively deletes the iPhone from a table named smartphones:

```
delete from smartphones where devicename = ?
```

The second argument is an array of JavaScript objects, each element representing the corresponding parameter in the SQL statement. Keeping with our example, you need to provide the value needed for the where clause of the delete statement within the array:

```
['iPhone']
```

The net effect of these two lines together results in this SQL statement:

```
delete form smartphones where devicename = 'iPhone'
```

This approach keeps you from worrying about delimiters and reduces your exposure to SQL injection attacks, which are a concern when working with dynamically constructed SQL statements.

The third argument is a function that's invoked when the SQL statement is successfully executed. The arguments to this callback function are a handle to the database transaction identifier along with an object representing the result of the statement.

For example, when you perform a select query against a database table, the rows are returned as part of the result, as you can see in listing 16.7, which shows the `listRecords` function from our sample application. In this listing, you use the returned rows to construct a rudimentary HTML table to dynamically populate the screen. There are other ways of accomplishing this task, but we kept it simple because our primary focus is on the interaction with the returned results set.

Listing 16.7 Processing returned rows from a query

```
function listRecords() {
    ret = "<table border='1'><tr><td>Id</td><td>Name</td></tr>";

    try {
      db.transaction(function(tx) {                        ❶ SQL statement
        tx.executeSql("select id,codename from versions",
        [],                                   ❷ Optional parameters
        function (tx,result) {                             ❸ Result function
        try {
          for (var i=0;i<result.rows.length;i++) {         ❹ Process result rows
            var row = result.rows.item(i);                      ❺ Work with one row
            ret += "<tr><td>" + row['id'] +               ❻ Append formatted string
              "</td><td>" + row['codename'] +
              "</td></tr>";
          }
          ret += "</table>";
          document.getElementById('outputarea').innerHTML = ret;
        } catch (e) {alert(e);}
        },
        function (tx,error) {
          alert("error fetching rows: " + error.message);
        });});
    }
    catch (e) { alert("Error fetching rows: " + e);}
}
```

The SQL statement is passed as the first argument ❶. In this case we're pulling two columns from the table called versions. The second parameter ❷ is the Java array holding any available parameterized values. In this sample, there are no parameterized values to pass along to the transaction, but you'll see one in the next section. Upon a successful execution of the select query, the `results` function is invoked ❸. The second parameter to the `results` function, which is named `result` in this code, provides access to the returned record set. The `result` object contains a collection of rows. Looping over the result set is as easy as walking through an array ❹. Each row is pulled out of the result set ❺ and individual columns are extracted by name ❻.

The fourth and final argument to the transaction method is the error handler. Like the success handler, this is also a callback function that takes two parameters. The first parameter is again the transaction identifier and the second is an object representing the trapped error.

With this basic understanding of the database object's transaction method, let's review the remaining functions contained in the db.js file.

16.3.6 Inserting and deleting rows

Thus far you've seen how to open a database, create a table, and select rows from the table. Let's round this example out with an examination of the code required to insert a row and to remove all the rows from the table. The following listing shows the saveRecord and deleteAllRecords functions.

Listing 16.8 Data handling functions

```
function saveRecord(id,name) {
    try {
        db.transaction(function (tx) {
          tx.executeSql("insert into versions (id,codename) values (?,?)",
          [id,name],
          function(tx,result) {
            entryCount++;
            document.getElementById('outputarea').innerHTML = "# of rows : "
     + entryCount;
          },
          function(tx,error) {
            alert("Error attempting to insert row" + error.message);
          });});

    } catch (e) { alert (e); }
}

function deleteAllRecords() {
    try {
        db.transaction(function (tx) {
          tx.executeSql("delete from versions",
          [],
          function(tx,result) {
            entryCount = 0;
            document.getElementById('outputarea').innerHTML = "# of rows : "
     + entryCount;
          },
          function(tx,error) {
            alert("Error attempting to delete all rows" + error.message);
          });});
    } catch (e) { alert (e); }
}
```

❶ Save record

❷ Insert SQL statement

❸ Define parameters

❹ Update count

❺ Define error handler

❻ Delete SQL statement

Inserting a row into our sample database takes place in the saveRecord method ❶. This method takes two arguments: id and name. A parameterized SQL insert statement ❷ is crafted providing a placeholder for each of the values required for the versions table. The parameters themselves are provided in an array ❸. The success handler ❹ is invoked after a successful insertion. When an error occurs during the SQL statement's execution, the user is notified via a simple JavaScript alert ❺. Of course, more sophisticated error responses can be crafted as desired. In the deleteAllRecords function you see a delete statement executed ❻.

If you're starting to get the feeling that this is just plain old SQL like you hoped, you're correct. And remember, this is running in the client side of your browser on your Android device!

Though the code runs happily in the Android browser, your phone isn't necessarily the most expedient way of testing the core functionality outside of the visual appearance. Testing on either a real Android device or the emulator both provide an acceptable experience, but for web applications such as this one, there's a better way to test: WebKit on the desktop.

16.3.7 Testing the application with WebKit tools

The good news about SQL development is that you can do it; the bad news is that the tools are limited compared to other SQL environments you're accustomed to in the desktop world. Fortunately, you can leverage the WebKit desktop browser, and its built-in development tools aid in your database work.

The Web Inspector[4] and Error Console found beneath the Develop menu in WebKit on the desktop provide helpful tools. When you're working with JavaScript, one of the challenges is that code tends to die silently. This can happen because something is misspelled or a function doesn't get properly defined thanks to a parsing error. When working in Eclipse with an Android SDK application, this kind of problem doesn't occur at runtime because the compiler tells you long before the application ever runs. With WebKit you can leverage the Error Console, which provides helpful pointers to parsing problems in JavaScript code. This is one of those "don't leave home without it" kind of tools.

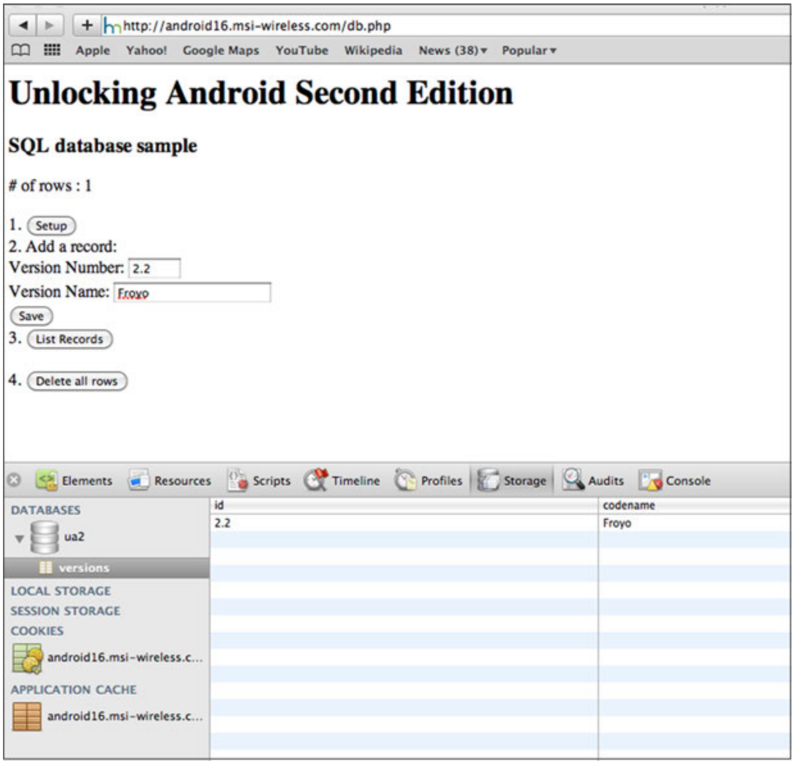
When you're working explicitly with a SQL database, the Web Inspector provides a helpful database tool that permits you to peer into the database and browse each of the defined tables. Although this tool isn't nearly as powerful as tools for commercial databases, there's something particularly reassuring about seeing your data in the table. Figure 16.9 shows a row in our versions table along with the web application running within the WebKit desktop browser.

The ability to move between the powerful WebKit desktop environment and the Android-based browser is a tremendous advantage to the Android developer looking to create a mobile web application.

As you've seen, the ability to store and manage relational data in a persistent fashion directly in the browser opens up new possibilities for application creation, deployment, and life-cycle management.

Storing data locally is a tremendous capability, but there may be times when you simply need more raw power and flexibility—and that calls for an SDK-based application. How do you get the best that HTML, CSS, and JavaScript have to offer but still go deeper? The answer is to build a hybrid application, which we cover next.

[4] For more details on the Web Inspector, try: http://trac.webkit.org/wiki/WebInspector.



Unlocking Android Second Edition

SQL database sample

# of rows : 1

1. Setup
2. Add a record:
Version Number: 2.2
Version Name: Froyo
Save
3. List Records

4. Delete all rows

id	codename
2.2	Froyo

Figure 16.9 Testing in the WebKit browser

16.4 Building a hybrid application

So far we've explored some of the capabilities of the Android browser and the flexibility it can provide to Android web developers. Fortunately those capabilities aren't constrained to pure web application developers only—you can bring the power of WebKit directly into your Android SDK applications. Including the browser control into an application is much more than a web page on a form, though of course if that's what you need, it's simple to implement. This section walks through the basics of building a hybrid application and demonstrates some features of the hybrid application model. It all starts with putting a browser on the screen, so let's begin there.

16.4.1 Examining the browser control

The browser control, found in the `android.webkit` package, may be added to any UI layout in the same way that you add a `TextView` or a `Button` to an `Activity` in your application. From there you can programmatically direct the browser to load content from a remote web page, from a file shipped with your application, or even content generated on the fly. This control's behavior may be overridden at any number of points and is a topic worthy of an entire book itself!

In its default condition, the control behaves just like the Android browser—minus the menus for navigating forward and back through your history and other typical browser actions. A common use for the control is for displaying a help screen for an application. Help written in HTML and downloaded from the vendor's website is the most convenient means of keeping information up-to-date, particularly if there's a user-community aspect to an application.

Things become more interesting as you consider the desired behavior of the web control. For example, have you ever wished you could provide a different message box for your application? Or how about implementing a feature in Java rather than JavaScript? All these things and more can be accomplished with the browser control—or you can just use it to browse your application's help docs. You decide when you configure the control in your `Activity`. Let's start there.

16.4.2 Wiring up the control

An application may override significant portions of functionality of the WebView browser control, including the `WebChromeClient`, the `WebViewClient`, and one or more JavaScript handlers, as described in table 16.2.

Table 16.2 Overriding browser behavior

Handler	Description
`WebChromeClient`	`WebChromeClient` controls visual aspects of the browser, including everyday tools such as the alert, confirm, and prompt functions. You can override `WebChromeClient` if you want to make a unique browsing user interface experience.
`WebViewClient`	`WebViewClient` modifies how the browser controls content navigation and lets you change the pages accessible to the user. Override `WebViewClient` if you want to filter the content in some fashion.
`JavaScriptInterface`	Custom JavaScript "libraries" are mapped into the namespace of the browser's JavaScript environment. `JavaScriptInterface` is the mechanism by which you can bridge the JavaScript/Java programming environments.

The following listing demonstrates setting up the browser control when the `Activity` starts.

Listing 16.9 Setting up a browser control

```
package com.msi.manning.webtutorial;

import android.app.Activity;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;
import android.webkit.WebView;
import android.webkit.WebSettings;
import android.webkit.WebChromeClient;
```

(1) **WebView imports**

```
public class WebTutorial extends Activity {

    private WebView browser = null;

   public static final String STARTING_PAGE =
     "file:///android_asset/index.html";

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        setupBrowser();
    }
    private void setupBrowser() {
      browser = (WebView) findViewById(R.id.browser);
      WebSettings browserSettings = browser.getSettings();
      browserSettings.setJavaScriptEnabled(true);
      browser.clearCache(true);
      browser.setWebChromeClient(new UAChrome(this));
      browser.addJavascriptInterface(new UAJscriptHandler(this),
     "unlockingandroid");
      browser.setWebViewClient(new
     UAWebViewClient(this.getApplicationContext()));
      browser.loadUrl(STARTING_PAGE);
    }
}
```

❷ WebView instance variable

❸ Starting web page

❹ Inflate layout

❺ Set up browser control

❻ Get reference to control

❼ Get browser settings object

❽ enable Javascript

❾ Set up Chrome client

❿ Set up JavaScript handler

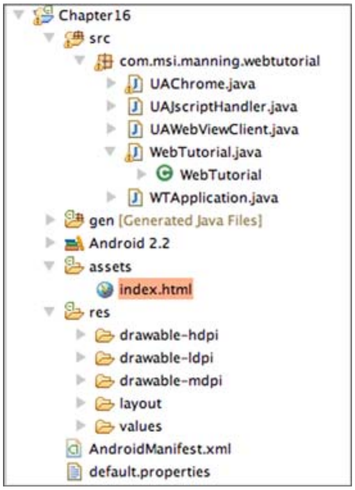
⓫ Set up WebView client

⓬ Load content into control

As with any aspect of Java programming, you have to create the required import statements. In this case, the android.webkit package ❶ provides functionality for the browser control. An Activity-scoped variable ❷ is defined for accessing the WebView browser control. A constant ❸ is defined as the starting page for the Activity. The layout is inflated in the same way virtually every Android Activity operates ❹; the interesting point here is that when this layout inflates, it instantiates a browser control that's subsequently set up with a call to the method setupBrowser ❺. findViewById is used to obtain a reference to the browser control ❻. With this reference, a number of characteristics of the browser may now be defined. Some of those characteristics are managed by the WebSettings object associated with the browser, so it's important to get a reference to the settings ❼. With the settings object available, you enable JavaScript ❽. Next you customize the behavior for this browser instance by installing your own handlers for UI events with the UAChrome class ❾, a JavaScript extension ❿, and a WebViewClient to handle navigation events ⓫. Finally, you get things started with the browser by loading a page of content into the control ⓬.

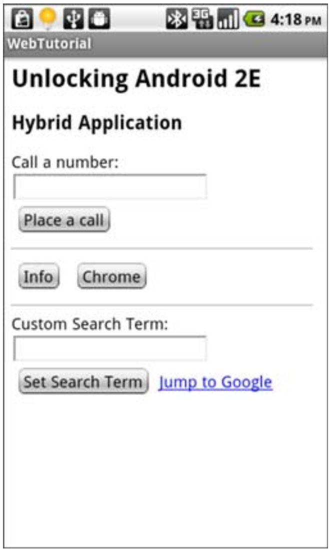
As we mentioned earlier, the browser control may load content from a remote web page, a locally stored page, or even from content generated dynamically. For the sample application built for this chapter, the initial page is stored under a folder named *assets*. The assets folder isn't created automatically and must be created by the developer. Figure 16.10 shows the index.html file in relation to the project.

Android maps this folder at runtime to the logical path name of android_asset, as in file:///android_asset/index.html. Figure 16.11 shows the index.html page running in the application.



```
Chapter16
  src
    com.msi.manning.webtutorial
      UAChrome.java
      UAJscriptHandler.java
      UAWebViewClient.java
      WebTutorial.java
        WebTutorial
      WTApplication.java
  gen [Generated Java Files]
  Android 2.2
  assets
    index.html
  res
    drawable-hdpi
    drawable-ldpi
    drawable-mdpi
    layout
    values
  AndroidManifest.xml
  default.properties
```

Figure 16.10 index.html used in our sample application



4:18 PM

WebTutorial

Unlocking Android 2E

Hybrid Application

Call a number:

Custom Search Term:

Figure 16.11 index.html in our browser control

The sample application is broken up into three demos, each highlighting a slightly different aspect of working with a browser control in an SDK application (a hybrid application). This approach is typical of hybrid applications where the browser control provides the UI and the Java code of the application provides much of the functionality and program logic. The next few sections break down each of these pieces of functionality along with the respective demonstration. Let's begin by looking at the technique of adding a JavaScript handler to our code.

16.4.3 Implementing the JavaScript handler

There are three pillars to the hybrid application, each supporting one of the major areas of functionality that the WebView browser provides. The most straightforward of these is the JavaScript handler. The browser control permits an application to "install" one or more JavaScript handlers into the JavaScript namespace of the browser. You've already seen in listing 16.9 how the browser is set up. To emphasize the JavaScript handler initiation, it's repeated here:

```
browser.addJavascriptInterface(new UAJscriptHandler(this),
 "unlockingandroid");
```

The arguments to the `addJavascriptInterface` mechanism are simple. The first is an application-provided handler and the second is a textual string. Listing 16.10 presents the class for the handler. Note that this kind of class may also be implemented as an inner class to the `Activity`, but we broke it out for clarity in this example. One consequence of breaking the class out on its own is the need to provide a `Context` for the code in order to access various elements of the SDK. You accomplish this by passing the context into the constructor, as in this listing.

Listing 16.10 JavaScript interface code implementation

```
package com.msi.manning.webtutorial;

import android.content.Intent;
import android.content.Context;
import android.net.Uri;
import android.util.Log;
import android.widget.Toast;

public class UAJscriptHandler {
    private String tag = "UAJscriptHandler";
    private Context context = null;

    public UAJscriptHandler(Context context) {
      Log.i(tag,"script handler created");
      this.context = context;
    }

    public void Log(String s) {
      Log.i(tag,s);
    }

    public void Info(String s) {
      Toast.makeText(context,s,Toast.LENGTH_LONG).show();
    }

    public void PlaceCall(String number) {
      Log.i(tag,"Placing a phone call to [" + number + "]");
      String url = "tel:" + number;
      Intent callIntent = new Intent(Intent.ACTION_DIAL,Uri.parse(url));
      context.startActivity(callIntent);
    }

    public void SetSearchTerm(String searchTerm) {

      WTApplication app = (WTApplication) context.getApplicationContext();
      app.setSearchTerm(searchTerm);
    }

}
```

❶ JavaScript interface

❷ Context reference

❸ Constructor

❹ Log function

❺ Custom notification method

❻ PlaceCall function

❼ Intent to initiate call

❽

❾ String assigned as search term

The UAJscriptHandler class ❶ implements various functions that are mapped to the browser control's JavaScript environment. Because this class is implemented outside of the browser control's containing Activity, you store a reference to the Context ❷, which is passed in with the constructor ❸. A custom function named Log ❹ takes a String parameter and adds an entry to the LogCat file when invoked. Troubleshooting JavaScript code is a nontrivial exercise; the ability to add entries to the LogCat is a big help during development. A custom function named Info ❺ takes a provided String and presents it as a Toast notification. This is helpful for providing information to the user while browsing without the annoyance of an alert pop-up box.

An excellent example of bridging the gap between web page and SDK programming is found in the PlaceCall function ❻. When this function is invoked, the code initiates a Dial to call a phone number that was provided by the user on the HTML page. When the number is received and passed to this function, an Intent is crafted to initiate a dial action ❼. The final portion of the JavaScript interface is a function named SetSearchTerm. We'll discuss this further when we examine the WebViewClient. For now, note that this function obtains a reference to the application's global class named WTApplication ❽. With this reference the passed-in string is assigned as a search term ❾.

Remember that some functionality implemented in the JavaScript handler may require you to add entries to the application's manifest file. For example, the PlaceCall method relies on the application having the android.permission.CALL_PHONE entry in the AndroidManifest.xml file.

Now that you've defined the JavaScript implementation class, let's look at how this code is invoked. To do that, we need to jump over to the index.html file.

16.4.4 Accessing the code from JavaScript

When the JavaScript interface code was installed, two arguments were passed to the addJavaScriptInterface method. The first was the implementation class, which we just introduced. The second argument is a string representing the namespace of the code as known to the calling JavaScript code. In our example, you passed in a string named unlockingandroid as the second value. This means that in order to access the functions within that JavaScript interface, you need to prefix the functions in the JavaScript code with the string unlockingandroid. So, to call the Info function you use code such as `window.unlockingandroid.Info('Hi Android');`.

Let's delve further into the code and examine how the JavaScript interface is utilized.

16.4.5 Digging into the JavaScript

The following listing presents the index.html file, which contains the UI for this application. For each of the buttons on the HTML page, there exists a simple onclick handler that invokes the respective JavaScript functionality.

Listing 16.11 index.html

```
<html>
<head>
<meta name="viewport" content="width=device-width" />
</head>
<body>
<h2>Unlocking Android 2E</h2>
<h3>Hybrid Application</h3>
Call a number:&nbsp;<input type="text" id="phone"/>&nbsp;
<button onclick=
     "window.unlockingandroid.PlaceCall(
     document.getElementById('phone').value);">
     Place a call</button>
<hr/>
<button onclick=
     "window.unlockingandroid.Info('Hi Android');">
     Info</button>&nbsp;
```

❶ Set up viewport

❷ Request phone number

❸ Invoke PlaceCall

❹ Invoke Info

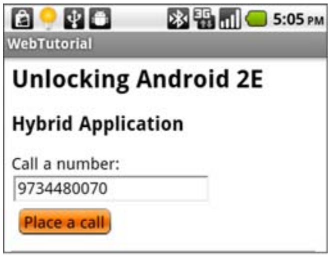
```
<button onclick="alert('Hey chrome!');">Chrome</button>
<hr/>
Custom Search Term:&nbsp;<input type="text" id="term"/><br/>
<button onclick=
     "window.unlockingandroid.SetSearchTerm(
     document.getElementById('term').value);">
     Set Search Term</button>
<a href="http://google.com">Jump to Google</a>
</body>
</html>
```

❺ Invoke alert function

❻ Invoke SetSearchTerm

❼ Define simple anchor tag

You include a viewport meta tag reference ❶ to this page, making it scale nicely to the Android device window. The first input text box ❷ accepts a phone number to be passed to the PlaceCall function ❸. Note the fully qualified name of the function: window.unlockingandroid.PlaceCall. Likewise, the Info function ❹ is called, passing in a static string. The alert function ❺ is invoked; this is the same alert function web developers have been using for over a decade. You'll see how this function is handled later when we review the WebChromeClient implementation. Also, for a later demonstration of the WebViewClient, the SetSearchTerm function ❻ passes a user-supplied textual string to the code, which is subsequently managed by the JavaScript interface in the Java code. A link to Google's home search page ❼ is provided to demonstrate traditional links and to provide a launching point for the WebViewClient demo.



WebTutorial

Unlocking Android 2E

Hybrid Application

Call a number:

9734480070

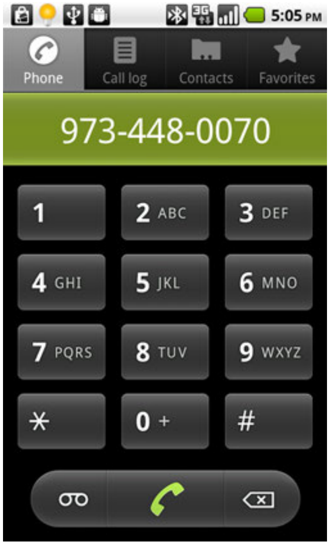
Place a call

Figure 16.12 Entering a phone number in the web page

As you can see, the JavaScript side of things is simple. All you have to do is prefix the function name with the namespace used when the code was registered in the SDK code side of our hybrid application. Note that the structure of the JavaScript interface technique permits adding multiple JavaScript handlers, each registered independently and uniquely identified with their own namespace.

Figure 16.12 shows the collection of a phone number in the browser window.

Once the user clicks the button, the window.unlockingandroid.PlaceCall function is invoked, passing in the value of the input text field. From here, the Java code behind the PlaceCall function creates the Intent to start the dial action. Figure 16.13 shows the number ready to be dialed.



973-448-0070

Figure 16.13 Ready to dial!

When the user clicks the Info button on the web page, the button's `onclick` handler invokes `window.unlockingandroi.Info`, passing in a string of "Hi Android". Figure 16.14 demonstrates the result of that action.

Clearly, the JavaScript implementation is powerful—so powerful that many commercial applications are written with an open source project named PhoneGap that leverages this WebKit browser control technique. PhoneGap provides a framework for developers to create their applications in HTML, CSS, and JavaScript, where the device-specific features are provided in Android SDK Java code. For more information about PhoneGap, visit http://phonegap.com.

With power comes responsibility. There are some security concerns with this interfacing technique.

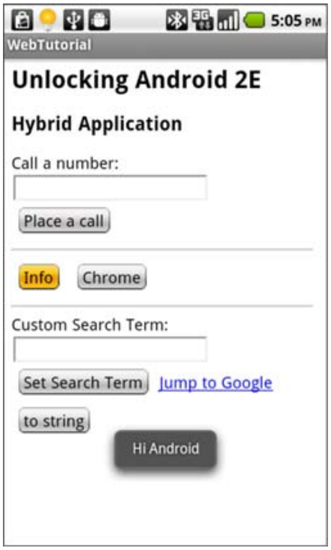


Figure 16.14 The Info button shows a `Toast` notification.

16.4.6 Security matters

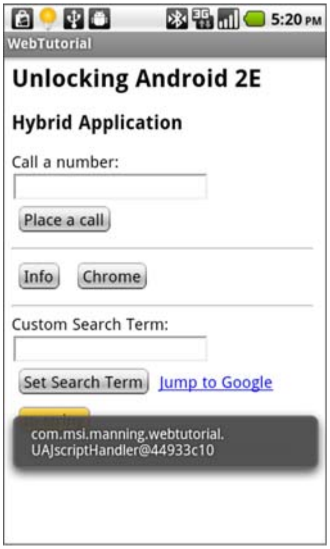
Using the JavaScript interfacing capabilities opens your application to potential security risks because anyone who knows your custom "API" implemented in your JavaScript handler can exploit those features. It may be wise to permit this functionality only for HTML that you've written yourself. To demonstrate just how wired-in this JavaScript interface actually is to your application, consider this JavaScript code, which may be implemented by a malicious piece of JavaScript:

```
<button onclick=
     "alert(window.unlockingandroid.toString());">
     toString</button>
```

The result? Look at figure 16.15.

A simple method call identifies the name of the class behind the interface!

If you determine that a page isn't of your own creation and want to disable the connection between your JavaScript code and Java code, you can re-register the JavaScript interface with a class that contains limited functionality, a class with zero methods (like that in the following listing), or even a null object.



```
com.msi.manning.webtutorial.
UAJscriptHandler@44933c10
```

Figure 16.15 The `toString` method called from JavaScript

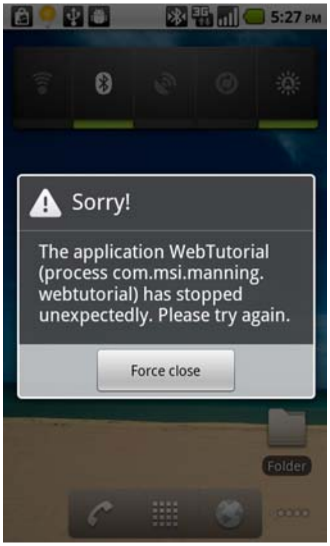
Listing 16.12 A no-op class

```
private class UANOOP {
}
```

Even a class with no methods isn't immune from trouble. Consider this JavaScript code run against our no-op class: `window.unlockingandroid.wait()`. Guess what this does to our application? Figure 16.16 demonstrates the destructive power of a malicious code calling methods of the root Java `Object` class.

If you plan on taking this approach of putting in a "safe" class implementation, a better move is to install null as the class to handle the Javascript interface: `browser.addJavascriptInterface(null,"unlockingandroid");`. Or better yet, don't allow navigation away at all.

Okay, enough of the drama. For now, let's assume that the web is a safe place to navigate and you can lock things down as you see fit. If you want to exercise more control over navigating the web with your in-the-app browser, you can implement your own `WebViewClient`. We'll show you how.



Sorry!

The application WebTutorial (process com.msi.manning.webtutorial) has stopped unexpectedly. Please try again.

Figure 16.16 A crashed application thanks to the `wait` function

16.4.7 Implementing a WebViewClient

The basic approach to controlling a browser control's navigation is to subclass the `WebViewClient`, which is also part of the `android.webkit` package. This class provides methods such as `onPageStarted()`, `onPageFinished()`, and numerous other methods related to browser navigation. For example, the class also has a method named `onReceivedSslError()`, which is invoked when a visited site has a digital certificate that either has expired or is invalid for some other reason. The range of functionality here is significant and beyond the scope of this section, so we'll focus on a narrower and more practical example to demonstrate the use of this class.

16.4.8 Augmenting the browser

As mentioned earlier, the application associated with this chapter, WebTutorial, contains a few demos. In this portion of the code, we demonstrate the `WebViewClient` functionality by monitoring the loading of pages in the browser control, searching for a predefined term. It's designed to be a simple browsing augmentation tool: you could take this basic framework and build a browser with customized functionality.

Here's how the application works. Starting in our application's home page, enter a search term, as shown in figure 16.17. In our case we're interested in the term *HTC*, which is a company that manufactures a number of Android phones. We're interested in how many times the term HTC shows up on any page that we load in the browser.

When we click the Set Search Term button, our Java code is invoked, which stores this search term in the application globals. Listing 16.13 shows portions of both the JavaScript handler code and the `WTApplication` class. The `WTApplication` code manages the application's global variables.

Listing 16.13 Managing the search term

```
import android.app.Application;

public class WTApplication extends Application {

    private String searchTerm = "";

    public void setSearchTerm(String searchTerm) {

      this.searchTerm = searchTerm;
    }

    public String getSearchTerm() {
      return this.searchTerm;
    }
}

public class UAJscriptHandler {

   public void SetSearchTerm(String searchTerm) {

    WTApplication app = (WTApplication) context.getApplicationContext();
    app.setSearchTerm(searchTerm);
   }
}
```

❶ Implement android.app.Application

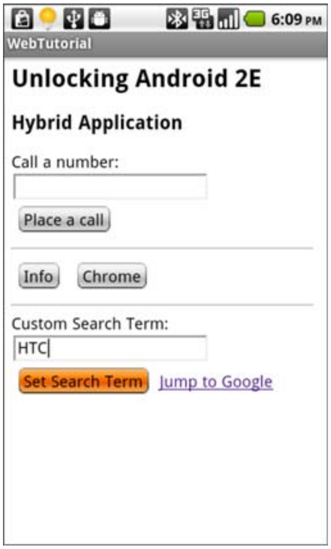
❷ Manage global search term

❸ Get reference to Application

❹ Set search term

A strategy to manage application globals in an Android application is to provide an implementation of the `android.app.Application` class ❶. This class is accessible to every `Activity`, `Service`, `BroadcastReceiver`, and `ContentProvider` within a given application, and as such is a convenient means of sharing information. In this case you implement a simple pair of setter/getter methods for managing the search term ❷. When a user clicks the Set Search Term button in the browser control, the code gets a reference to the `android.app.Application` implementation ❸ and then updates the search term ❹.

With the search term safely stored, it's time to navigate the web.



Unlocking Android 2E

Hybrid Application

Call a number:

Custom Search Term: HTC

Figure 16.17 Setting up a search term

16.4.9 *Detecting navigation events*

Now you need to begin browsing the internet. For lack of a better place to start, jump to the Google home page and enter the search term *android*, as shown in figure 16.18.

As soon as the page is launched, we want to remind our users that we're searching for the term HTC, and when the page is fully loaded we want users to know whether the term was found, as shown in figure 16.19.

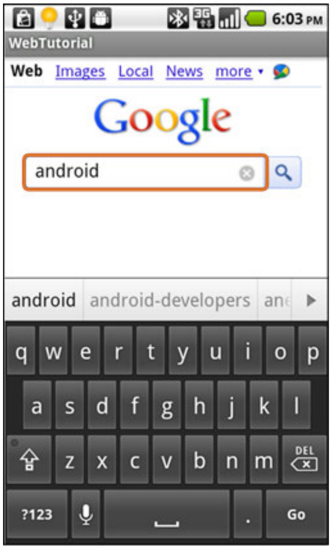
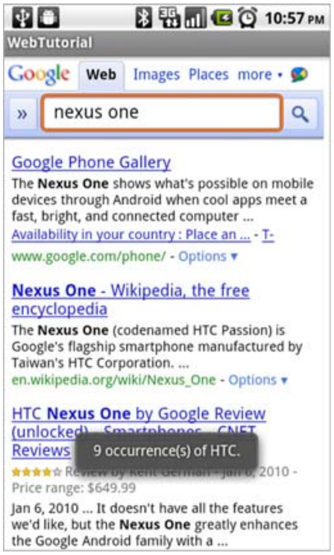


Figure 16.18 Searching the web via Google



Google Phone Gallery
The **Nexus One** shows what's possible on mobile devices through Android when cool apps meet a fast, bright, and connected computer ...
Availability in your country : Place an ... - T-
www.google.com/phone/ - Options ▾

Nexus One - Wikipedia, the free encyclopedia
The **Nexus One** (codenamed HTC Passion) is Google's flagship smartphone manufactured by Taiwan's HTC Corporation. ...
en.wikipedia.org/wiki/Nexus_One - Options ▾

HTC **Nexus One** by Google Review (unlocked) - Smartphones - CNET Reviews
★★★★☆ Review by Kent German - Jan 6, 2010 -
Price range: $649.99
Jan 6, 2010 ... It doesn't have all the features we'd like, but the **Nexus One** greatly enhances the Google Android family with a ...

9 occurrence(s) of HTC.

Figure 16.19 Hit indicator

As you can see, searching the web for *android* + *HTC* is a different operation compared to searching for android and then HTC. Yes, this functionality can be accomplished with some search engine magic. The point is that you want to be able to have this browsing assistant follow you wherever you go on the internet—even beyond the search engine launching point. Any time you browse to a new page, you receive a Toast notification letting you know how many times your search term appears on the page. To show you how this is implemented, let's examine the following listing, which shows the implementation of the UAWebViewClient class.

Listing 16.14 UAWebViewClient.java

```
package com.msi.manning.webtutorial;

import android.content.Context;
import android.util.Log;
import android.graphics.Bitmap;
import android.webkit.*;
import android.widget.Toast;
                                                          1 Extend
public class UAWebViewClient extends WebViewClient{         WebViewClient
    private String tag = "UAWebViewClient";
    private Context context;
    public UAWebViewClient(Context context) {       2 Call super
      super();                                        methods
      this.context = context;
    }                                                                    2
    public void onPageStarted(WebView wv,String url,Bitmap favicon) {
      super.onPageStarted(wv,url,favicon);
      if (!url.equals(WebTutorial.STARTING_PAGE)) {      3 Selective operation
```

```
        WTApplication app = (WTApplication) context;
        String toSearch = app.getSearchTerm();
        if (toSearch != null && toSearch.trim().length() > 0) {
          Toast.makeText(context,"Searching for " +
     toSearch,Toast.LENGTH_SHORT).show();
        }
      }
    }
    public void onPageFinished(WebView wv,String url) {
      super.onPageFinished(wv,url);
      Log.i(tag,"onPageFinished");
      if (!url.equals(WebTutorial.STARTING_PAGE)) {
        WTApplication app = (WTApplication) context;
        String toSearch = app.getSearchTerm();
        if (toSearch != null && toSearch.trim().length() > 0) {
          int count = wv.findAll(app.getSearchTerm());
          Toast.makeText(app, count + " occurrences of " + toSearch +
     ".",Toast.LENGTH_SHORT).show();

        }
      }
    }
}
```

❹ Get globals

❺ Get search

❻ Find all occurrences of term

❷ Call super methods

❹ Get globals

❺ Get search

❻ Find all occurrences of term

Extending the WebViewClient ❶ for our purposes involves a custom constructor plus two subclassed methods. For each of the three methods, you call the respective super methods ❷ to make sure the parent class has an opportunity to perform necessary housekeeping. When filtering the page loads, you don't want your augmented behavior to fire if you're on the starting index.html page shipped with the application, so you do a basic string comparison ❸ to selectively ignore the additional functionality if you're back on the starting page within your own content. Assuming you're on a subsequently loaded web page, you need to access the android.app.Application instance ❹ to gain access to the globally stored searchTerm ❺. In the onPageStarted method, you remind the user of the term you're searching for through the use of a Toast notification. In onPageFinished, you call the findAll method to search the newly downloaded page for the search term ❻. When the search is completed, you again let the user know via a Toast notification.

This WebViewClient extension is installed with a call to

```
browser.setWebViewClient(new
 UAWebViewClient(this.getApplicationContext()));
```

This simple application is begging for enhancements, so feel free to download the sample code and extend the WebViewClient yourself and enjoy the feeling of adding functionality to the world's most popular application—the browser!

If you have a taste for improving additional behavior modification of the browser, you'll want to try your hand at updating the chrome by subclassing WebChromeClient.

16.4.10 Implementing the WebChromeClient

The `WebChromeClient` class is responsible for handling client-side events such as JavaScript alerts, prompts, and the like. Additionally, the `WebChromeClient` is notified when a window is being closed or a new window is attempting to load. Like the `WebViewClient`, this class is a topic unto itself. For our purposes here, we demonstrate its use by overriding the `alert()` JavaScript function.

Referring to the index.html file from listing 16.11, you see the code for handling the Chrome button: `<button onclick="alert('Hey chrome!');">Chrome</button>`. There's no special namespace qualifier on this function—it's the same JavaScript alert function used since client-side programming first littered our world with pop-up messages. This function's sole purpose is to inform the user of information that's hopefully relevant to their browsing activity. Listing 16.15 demonstrates subclassing the `WebChromeClient` and implementing the `onJsAlert` method. Again, this method is invoked when the JavaScript alert function is encountered in a running web page.

Listing 16.15 UAChrome.java

```
package com.msi.manning.webtutorial;

import android.content.Context;
import android.widget.Toast;
import android.webkit.JsResult;
import android.webkit.WebChromeClient;
import android.webkit.WebView;
import android.widget.Toast;

public class UAChrome extends WebChromeClient {

    private Context context;
    public UAChrome(Context context) {
      super();
      this.context = context;

    }
    public boolean onJsAlert(WebView wv,String url,
     String message,JsResult result) {
      Toast.makeText(wv.getContext(),message,Toast.LENGTH_SHORT).show();
      result.confirm();
      return true;
    }
}
```

❶ Extending WebChromeClient

❷ Call super method

❸ Override onJsAlert

❹ Toast notification

❺ Confirm result

❻ Return true

The `UAChrome` class extends the `WebChromeClient` ❶. In the constructor, you call the constructor of the super class with a call to `super` ❷. In order to process alert statements, you need to implement the `onJsAlert` method ❸. This method takes four arguments: the `WebView` instance hosting the JavaScript code, the URL of the page, the message to be displayed, and a `JsResult` argument. The `JsResult` object has methods for indicating to the caller whether the user confirmed or canceled the prompt. In this sample implementation, you replace the pop-up box with a tidy `Toast` notification ❹ and because the `Toast` notification clears on its own, you need to inform the caller that the notification was "confirmed." This is accomplished with a call to the `confirm` method of the `JsResult` parameter named `result` ❺. Finally you return a Boolean `true` ❻ to indicate that this overridden method has processed the event.

16.5 Summary

This chapter covered a fair amount of material related to web development for Android. As with most topics related to programming, we've only scratched the surface—and hopefully provided you with inspiration and guidance on where to dig deeper on your own.

Starting with the idea of making pure web applications as friendly as possible on the Android standalone browser, we explored the viewport meta tag along with the technique of loading platform-specific CSS files. Style sheet management is an important and recommended skill for any kind of web development today.

Moving beyond the look and feel aspects of the in-the-browser application model, we explored the technique of SQL database access directly from the client-side JavaScript. After a basic demonstration of the data access methods, we looked at a convenient testing platform in the desktop version of WebKit.

We then moved on to cover perhaps the most exciting aspect of web programming in the mobile world today: creating hybrid applications. Hybrid applications can enhance an Android SDK application with highly capable formatting of selective areas such as help screens or tutorials. Or, as demonstrated in this chapter's sample code, a hybrid application can enable a purely HTML and CSS user interface while permitting the underlying Java code to perform the heavy lifting. Regardless of your approach, having an understanding of the basics of Hybrid application development is a benefit to the Android programmer.

In the next chapter, we'll look at one of the "game changing" features of the Android platform: the `AppWidget`.

17 AppWidgets

This chapter covers

- Introducing the AppWidget
- Introducing the sample application: SiteMonitor
- Implementing the AppWidgetProvider
- Configuring an AppWidget instance
- Updating the AppWidgets

Continuing with the theme of exploring the maturing Android platform, this chapter examines the AppWidget, which brings functionality to the *phone-top*, or home screen, of an Android device. In this chapter, you construct an application named SiteMonitor that ties together much of the fundamental application development skills from prior chapters and adds an interactive presence on the home screen.

By placing content directly on the Android device's home screen, you empower the user to fully leverage the platform by making powerful tools available for use quickly and conveniently. Think about it—the deeper a user has to tap into an application to find information of value, the less likely the application will become an everyday staple. The goal is to create a mobile experience that brings value without becoming a black hole of time, attention, or worse, annoyance. A key ingredient to meeting this objective is the effective use of the AppWidget. This chapter equips you with an understanding of the uses and architecture of `AppWidgets`. It walks step by step through the creation of a nontrivial `AppWidget` example application, including important tasks such as configuration, data, and GUI updates, and wraps up with a discussion of the elements required within the application's AndroidManifest.xml file.

17.1 Introducing the AppWidget

When a user picks up an Android device (or any smartphone for that matter), their first impression is often defined by their experience interacting with the phone's home screen. On the home screen, a user interacts with applications, initiates a phone call, searches the device, launches a browser, and more. By the time the user begins to browse for an application among the sea of icons in the launcher window, a certain percentage of users will be "lost" and will conclude that the device just isn't "user friendly." This can be likened to burying important web content deep within a website—you need to be wary of hiding the value of your applications. One solution to this challenge for Android is to employ an `AppWidget`.

17.1.1 What's an AppWidget?

An `AppWidget` is code that runs on the home screen of an Android device. The visual size of an `AppWidget` instance can vary and is designated by the programmer at design time. The home screen is broken up into 16 usable spaces, each of which is approximately 74 pixels square. An `AppWidget` can span a rectangular area ranging from 1x1 spaces to 4 x 4 spaces, provided there's room on the current home screen "page." (A typical phone has around five to nine home screen pages.)

An `AppWidget` is typically deployed as a read-only interface providing a view into application data. The UI is implemented in a manner similar to a `layout` for a traditional Android `Activity`. Unlike the `Activity`, the `AppWidget` is much more constrained in that the only user action permitted is a click. An `AppWidget` can't present a scrollable list, an `EditText`, or any other user input mechanism beyond something that can react to a click. When anything beyond a click is required as user interaction, it's prudent to load an `Activity` for the heavy lifting—an `AppWidget` just isn't designed for significant user interactions. In this case, the `AppWidget` acts as a storefront for the underlying `Activity`. It accepts a click and then hands control off to the back office, implemented typically by an `Activity`.

Despite this apparent shortcoming, not all `AppWidgets` require anything beyond a basic user interface. For example, consider the Power Control Widget shown in figure 17.1. The Power Control Widget is an

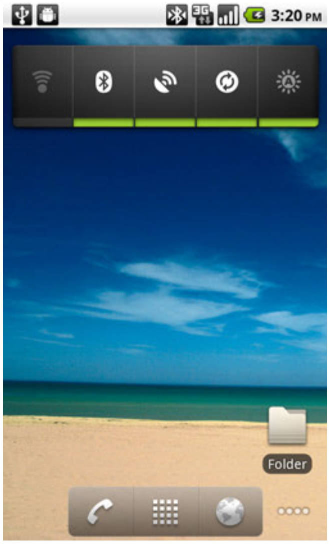


Figure 17.1 Power Control Widget on the home screen

excellent demonstration of simplicity and value. This widget is used to enable and disable various system services such as Bluetooth, Wi-Fi, GPS, and other battery-impacting functions. GPS services are a significant drain on the battery—the fact that the Power Control Widget exposes this on/off feature so easily makes the use of location-based services a more realistic option for Android users. On other phone platforms, this kind of functionality is generally "hidden" under system or option menus.

You add AppWidgets to the home screen by pressing and holding an empty area until a menu launches, as shown in figure 17.2.

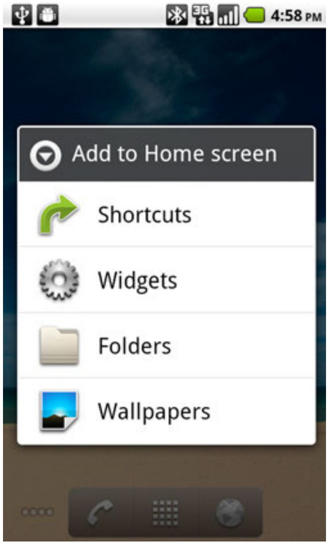


Figure 17.2 **Add to home screen**

From this menu, select Widgets and available AppWidgets are displayed in a scrollable list, as shown in figure 17.3. Tap on the desired widget to add an instance to your home screen.

An AppWidget runs under another application, namely an AppWidgetHost, which is typically the device's home screen. The AppWidget code is implemented in an instance of an AppWidgetProvider, which is an extension of the BroadcastReceiver class. Recall from prior chapters that a BroadcastReceiver is defined as a "receiver" in the AndroidManifest.xml file. The AppWidgetProvider is a BroadcastReceiver with a special IntentFilter and a metadata tag that further defines the AppWidgetProvider's characteristics. An AppWidget may be implemented in code as a BroadcastReceiver alone, yet the AppWidgetProvider provides some convenience wrapper functionality and is the recommended means of coding an AppWidget.

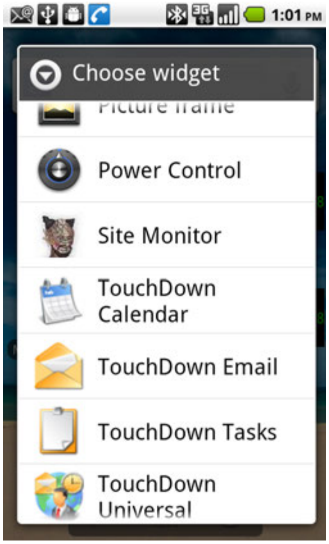


Figure 17.3 **Choose a widget, any widget.**

AppWidgets are designed to be updated periodically. The stock implementation of an AppWidget automatically updates at an interval defined by the developer at design time. In general, this update is kept to a low frequency to conserve battery power. There are other mechanisms for updating an AppWidget on an as-needed basis through the use of Intents. The AppWidgetProvider extends BroadcastReceiver and therefore can receive different Intent Actions based on the defined IntentFilters. The common practice is to define an application-specific IntentFilter action and use the sendBroadcast method to trigger an AppWidget update on an as-needed basis.

The details of the AppWidgetProvider, the special metadata in the AndroidManifest, IntentFilters, RemoteViews, and much more are all discussed in this chapter. Before we delve into the details of constructing an AppWidget, let's consider the various design patterns an AppWidget can satisfy.

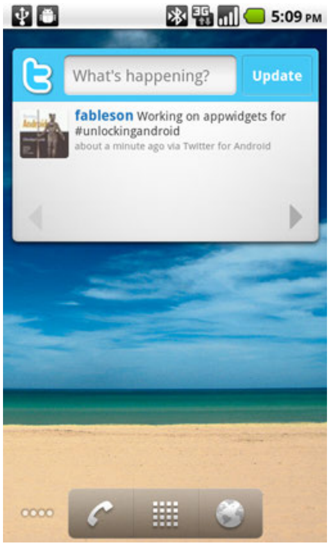
17.1.2 AppWidget deployment strategies

In its most basic implementation, an AppWidget can be considered a dashboard of sorts. The Power Control Widget shown in figure 17.1 is a good example of this flavor of AppWidget. This AppWidget has no other user interface to which it's tied and any actions taken directly invoke an underlying request to enable or disable a system feature. In any normal scenario, there'd be at most one Power Control Widget deployed to the home screen. A user is free to add multiple copies of the Power Control Widget to their home screen, but there's no additional utility or benefit from doing so.

Now consider an AppWidget for Facebook or Twitter, as shown in figure 17.4.

Some people have multiple social media accounts and may desire multiple AppWidgets instantiated for making updates to specific accounts. In this scenario, each AppWidget instance is tied to its own set of data. For the purposes of this chapter, we'll call this data the AppWidget instance model. Each instance of the AppWidget has its own set of data—which in the case of a Twitter account widget would look like username/password information plus any cached data related to the specific account. We go into significant detail later in this chapter on how to manage this per-widget data.

One role an AppWidget can play is as a "smart shortcut." Rather than simply displaying a static shortcut to an application, the AppWidget provides a means of displaying pertinent and timely information to the user. When clicked, the AppWidget loads the relevant Activity or launches a relevant web page. Consider, for example, a calendar widget that displays upcoming events on the home screen. Tapping on the widget causes the calendar application to load, jumping to the specific event of interest.



fableson Working on appwidgets for #unlockingandroid
about a minute ago via Twitter for Android

Figure 17.4 Tweeting about this chapter!

Due to the variable nature of an AppWidget's deployment strategy, it's important to give some consideration to how an AppWidget interacts with the other Android-based application components. What may seem like a simple AppWidget application may in fact require the collaboration of multiple components. Table 17.1 presents a nonexhaustive list of options for how an AppWidget may interact with other components within a suite of applications.

What makes AppWidgets specifically (and Android generally) so appealing to a developer is the ability to distribute code that provides a small, focused piece of

Table 17.1 Various AppWidget deployment patterns

Description	Additional application components	Example
AppWidget standalone	Singleton. No configuration options. No Activity required. May use a Service to process a longer-running request in some implementations.	Power Control.
AppWidget as smart shortcut	Used to present information and upon a click loads an Activity.	Email widget showing the number of messages in an inbox or upcoming events in a calendar.
AppWidget with specific static data	Configuration required. Variable amounts of data associated with each widget instance.	PictureFrame widget showing a user-selected image from the gallery.
AppWidget with dynamic data	Configuration required. Variable amounts of data associated with each widget. Service used to update data. BroadcastReceiver used to respond to alarm invocations or other triggers that prompt updates at various times.	News application widgets. The sample application from this chapter, SiteMonitor, demonstrates this pattern.

functionality. For example, let's say you have a great idea to make interacting with the calendar much easier: the "Killer Calendar App." In traditional mobile development, implementing your idea would require replacing the entire calendar application. With Android's architecture, a developer can distribute their application as simply an AppWidget that provides a better interface, yet not have to replace the mountains of work already shipped in the form of built-in applications! What's more, portions of your code could be called by third-party applications, provided they have the appropriate permissions.

Creating an AppWidget isn't something to be taken lightly. There are decisions to be made and pitfalls to avoid—all of which we cover in this chapter. Before we dive into the weeds of constructing an AppWidget of our own, let's take a step back and review the design objectives of SiteMonitor, our sample application for this chapter that provides the context for learning about AppWidgets.

17.2 Introducing SiteMonitor

To demonstrate the AppWidget, this chapter presents an application named SiteMonitor. SiteMonitor is a simple utility used to help monitor hosted applications such as blogs, commerce sites, and the like—web-based applications.

17.2.1 Benefits of SiteMonitor

The premise of SiteMonitor is that an Android device is highly connected and highly capable, and therefore should be leveraged to provide more value to the user than simply functioning as a fancy email device. Although this application may not be appealing to the mass consumer market, there's a nontrivial population of individuals (entrepreneurs, system admins, service providers, and so on) who have an interest in keeping one or more websites running.

The implications of a website/hosted application not running properly range from annoyance, to embarrassment, to loss of revenue when a commerce site isn't accepting transactions. Being able to keep an eye on a collection of online assets from anywhere is empowering and can even make things like a day out of the office seem more realistic. Mobile devices today often feel like long leashes—perhaps this application can lend a few more feet to the leash?

Of course it's possible to receive SMS alerts or emails when a site is no longer available with server-side tools and third-party services—which is fine and applicable in many instances. But keep in mind that SiteMonitor works entirely without server-side integration, another desirable feature that's in line with the trend of today's mobile applications.

SiteMonitor is an `AppWidget` designed to meet the objective of monitoring website health. Let's start with the high-level user experience.

17.2.2 *The user experience*

Checking on websites is possible today with an Android device—just bookmark a bunch of application URLs in the WebKit browser and check on them. And then check on them again. And again. You get the picture; it'd be desirable to get some automated assistance for this basic and repetitive task. The logical answer is to code an application to do this for you. In this case, SiteMonitor employs an `AppWidget` to bring useful information directly to the phone's home screen. Let's look at the user interface.

Figure 17.5 shows the home screen of an Android device with four instances of the SiteMonitor widget configured to monitor four different applications.

The UI is admittedly lackluster, but the information is precisely what's needed in this situation. At a glance you can see that of the four sites being monitored, three are up (green) and one is in an error condition (red). You can see the name of each site and the date/time of the most recent update. Tapping on one of the SiteMonitor widget instances brings up an `Activity` where you can see detailed information about the site.

The widget-specific data includes not only the configuration values for the hosted application but also the most recent data retrieved from the hosted application. For example, you see in figure 17.6 that this particular

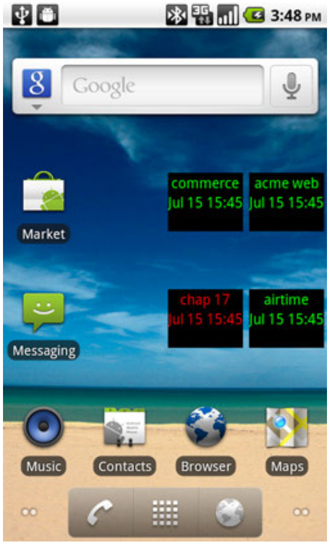
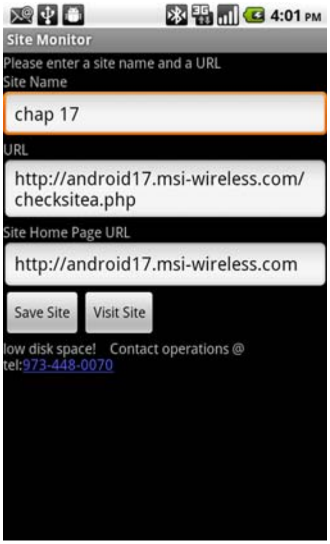


Figure 17.5 Four instances of the SiteMonitor `AppWidget` on the home screen

site, named “chap 17,” has a low disk space warning. The application can easily support multiple conditions; for now let’s keep things simple with a *good* condition and a *bad* condition. Good sites are shown in green and bad are shown in red on the home screen.


4:01 PM

Site Monitor

Please enter a site name and a URL

Site Name

chap 17

URL

http://android17.msi-wireless.com/checksitea.php

Site Home Page URL

http://android17.msi-wireless.com

low disk space! Contact operations @ tel:973-448-0070

Figure 17.6 Site details, including hot-linked text

Still referring to figure 17.6, note that the screen has support for maintaining three different fields, each implemented as an EditText control. The first EditText instance manages the name of the site. Space is limited on the widget, so it’s best to keep the length of this name limited. Note that the widget can be expanded to take more space and therefore permit more descriptive names, and most users will be able to use a short descriptor such as a server name or a client’s nickname. More importantly, you want to fit multiple instances of the SiteMonitor widget on the home screen, so they shouldn’t take up any more space than absolutely necessary.

The next EditText field holds the URL that the application periodically pings to check the status of the site. When it’s time to update the status of a particular hosted application, SiteMonitor performs an HTTP GET against this URL and expects a pipe-delimited return string. For example, a response from a commerce site might look like the following:

```
GOOD|There are 10 orders today, totaling $1,000.00
```

Or perhaps a site that’s down might have a response that looks like this:

```
BAD|Cannot reach payment processor.  Contact fakepaymentprocessor.com
at 555-123-4567
```

The third EditText field stores the site’s home page. When the Visit Site button is clicked, the application opens a browser and navigates to this URL. Why is this feature included? The answer is simple: the most common reaction to receiving a “the site is down” notification is to check out the site firsthand to see what kind of errors may be presenting themselves to site visitors. This approach is much easier than firing up the browser and tapping in a complete URL—particularly under duress!

Referring again to figure 17.6, note the highlighted phone number contained in the TextView field. Selecting that link causes the phone to launch the dialer as shown in figure 17.7. This TextView has the autoLink attribute enabled. When this attribute is enabled, the application is requesting that Android scan the textual information and attempt to turn any data that looks like a link into a clickable *hotspot*. Android can optionally look for websites, email addresses, phone numbers, and so forth. This feature is one more step in making things as streamlined as possible for a mobile user to support a hosted application.

Let's say the user is out to dinner and equipped only with an Android device when a site goes down or experiences an urgent condition. It may not be convenient or feasible to look up detailed support information about a particular hosted application, particularly if this site is one of many. When an error condition is observed, it's useful to have key actionable data right at your fingertips. This is akin to a "break glass in case of emergency" situation. An emergency has occurred, so let's be as prepared as possible.

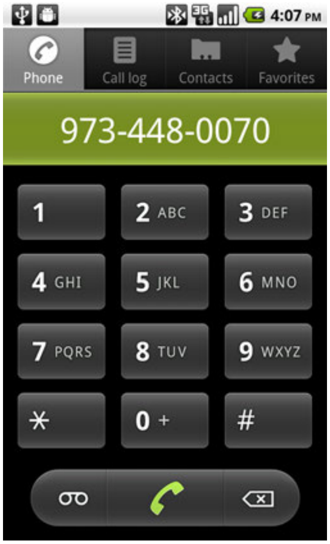
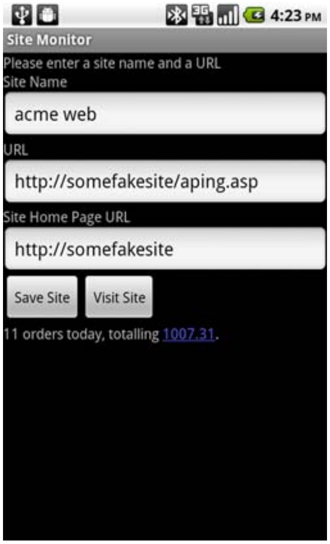


Figure 17.7 Easy dialing to an affected user

The application isn't limited to returning only bad news—it's feasible to have the application monitor good news as well. For example, figure 17.8 shows that the Acme site has received over $1,000 worth of business today! Note the highlighted numbers: one of the side effects of the auto-linking is that it sometimes returns false positives.



4:23 PM

Site Monitor

Please enter a site name and a URL

Site Name

acme web

URL

http://somefakesite/aping.asp

Site Home Page URL

http://somefakesite

Save Site | Visit Site

11 orders today, totalling 1007.31.

Figure 17.8 Monitor the good news also—revenue!

The SiteMonitor application is `AppWidget`-centric. Although you can configure and examine various aspects of a specific site through the configuration `Activity`, there's no other means of interacting with the application. This application doesn't appear in the main launcher screen. As a natural consequence of relying solely on instances of an `AppWidget` for the UI, the number of sites monitored by SiteMonitor is limited by the amount of home-screen real estate available upon which SiteMonitor widget instances can be placed. Remember the motivation to keep an `AppWidget` design as conservative as possible with respect to screen real estate. It's impossible to have an `AppWidget` that takes up less than one of the 16 available spaces on a home screen page, but easy to have one that's larger than a 1 x 1 space.

Please keep in mind that this choice of making SiteMonitor available exclusively as an `AppWidget` is an arbitrary design decision to demonstrate working with an `AppWidget`. A simple extension of this application could remove this intentional limitation. To add this application to the home screen, you could add an `Activity` that presents each of the monitored sites and then add the `Activity` to the home screen launcher.

Now that you have a basic understanding of what SiteMonitor is attempting to do, it's time to look at how the application is constructed.

17.3 SiteMonitor application architecture

Let's start with a high-level architectural view and then drill down so that you can better understand each major aspect of the SiteMonitor application. Some of the code examples are provided in this chapter in an abbreviated form due to their length. You're encouraged to download the full project to walk through the code as you move through the balance of this chapter.

Let's begin with a pictorial representation of the application.

17.3.1 Bird's-eye view of the application

The SiteMonitor application relies on multiple Android application components to deliver the desired functionality. Figure 17.9 depicts the major components of the application.

Examining in more detail the elements depicted in figure 17.9, you see an Android device ❶ hosting multiple instances of `AppWidgets` ❷. In this diagram, each of the four `AppWidgets` represents an instance of the `SiteMonitorWidget`. This `AppWidget` class is implemented in the class named `SiteMonitorWidgetImpl` that extends an Android-provided class named `AppWidgetProvider`.

Each instance of the SiteMonitor widget is configured by the user to monitor a distinct hosted application/website in the collection of sites ❸. The relationship

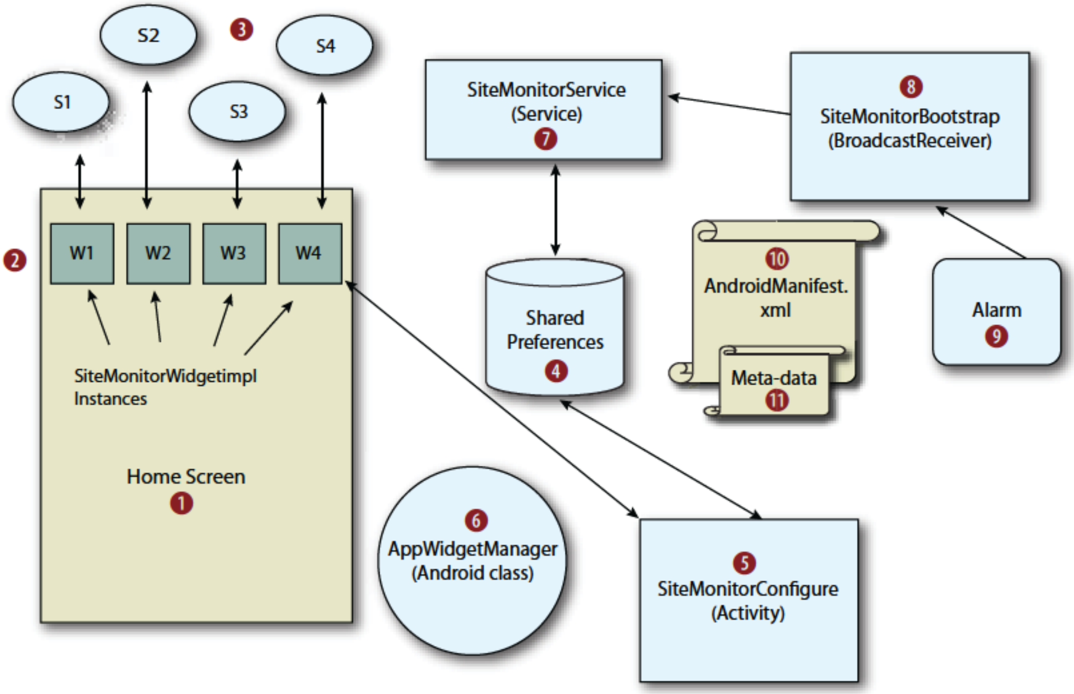


Figure 17.9 **Architectural diagram of the SiteMonitor application**

between a specific widget instance and its data is stored in the application's SharedPreferences ❹ persistent store. The class SiteMonitorModel provides methods for managing and manipulating these data model instances. An Activity ❺ is employed to permit a user to configure the widget instance data. This Activity is implemented as class SiteMonitorConfigure.

Each time a new SiteMonitor widget is added to the home screen, the SiteMonitorConfigure Activity is invoked to query the user for site-specific data: name, URL, home page. This auto-launching of an Activity happens thanks to a special relationship between an AppWidget class and its *configuration activity*. This relationship is defined in the metadata file ⓫. This metadata is referenced by the <receiver> entry of the SiteMonitorWidgetImpl class in the manifest file ❿. Note that the practice of having a configuration Activity for an AppWidget is optional.

When a SiteMonitor widget instance is added or removed, a corresponding method in the SiteMonitorWidgetImpl class is invoked. The specific methods of this class are described in section 17.3.2. The Android provided AppWidgetManager class ❻ acts as a helper to provide a list of AppWidget identifiers related to the SiteMonitorWidgetImpl class. At this point, there are multiple widget instances on the home screen, updating their visual display according to an update-refresh interval setting defined in the metadata file ⓫.

As mentioned earlier, it's common practice to perform an AppWidget update outside of its normally defined update interval. This update is generally accomplished through the use of a Service, often triggered by an Alarm. It's not a recommended practice to have long-running Services in the background, so you use the preferred method of employing an Alarm ❾ that propagates a PendingIntent.

The PendingIntent contains an Intent that's received by the SiteMonitorBootstrap class ❽. This class also contains static methods for managing the alarm, called at various points in the application, each of which is discussed further in this chapter. Consider the condition where there are no SiteMonitor widgets instantiated on the home screen. If there are no hosted applications to monitor, there's no need to have a periodic alarm activated. Likewise, when a new SiteMonitor widget is added to the home screen, it's desirable for an alarm to be set to ensure that the widget is updated periodically. The relationship between the Intent Action triggered by the alarm and consumed by the SiteMonitorBootstrap class is defined in the manifest ❿. The Action field of the Intent set in the Alarm's PendingIntent matches the IntentFilter Action for the SiteMonitorBootstrap.

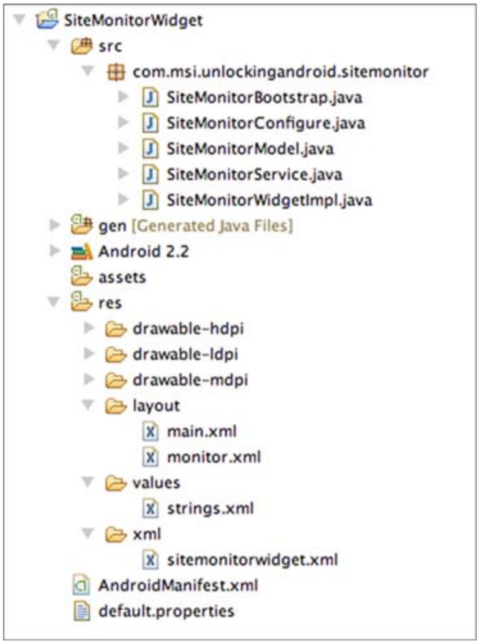
When the SiteMonitorBootstrap ❽ receives the Intent in its onReceive method, the resulting step is the starting of SiteMonitorService ❼. The SiteMonitorService is responsible for carrying out the updates and checking on the hosted applications.

When it's started, the SiteMonitorService iterates through the available widgets thanks again to assistance from the AppWidgetManager ❻, which provides a list of widgets. For each active widget instance, the Service extracts the hosted application's URL and performs an HTTP GET to retrieve the most up-to-date status information. The data model for each active AppWidget is updated with the information retrieved from the hosted application. When all the site data has been updated, the `Service` sends an `Intent` broadcast, in effect asking the `SiteMonitorWidgetImpl` class to update the visual status of the widgets themselves.

As you can see, there are quite a few moving pieces here in this prototypical `AppWidget` application. It may be helpful to refer back to this section as you consider each of the ensuing code descriptions. Let's now take a tour of the files in the project.

17.3.2 File by file

We'll be looking at code snippets soon, but first let's tour the project from a high level, discussing the purpose of each significant file in the project. Figure 17.10 shows the project in the Eclipse IDE, and table 17.2 provides a brief comment for each file.



```
SiteMonitorWidget
  src
    com.msi.unlockingandroid.sitemonitor
      SiteMonitorBootstrap.java
      SiteMonitorConfigure.java
      SiteMonitorModel.java
      SiteMonitorService.java
      SiteMonitorWidgetImpl.java
  gen [Generated Java Files]
  Android 2.2
  assets
  res
    drawable-hdpi
    drawable-ldpi
    drawable-mdpi
    layout
      main.xml
      monitor.xml
    values
      strings.xml
    xml
      sitemonitorwidget.xml
  AndroidManifest.xml
  default.properties
```

Figure 17.10 **SiteMonitorWidget in Eclipse**

Table 17.2 File listing for this project

Filename	Comment
AndroidManifest.xml	Contains definitions of each `Application` component in the application along with `IntentFilter`s and required permissions.
sitemonitorwidget.xml	Defines `AppWidgetProvider`-specific attributes, including dimensions, configuration activity, icon, and initial UI layout.
SiteMonitorWidgetImpl.java	Contains the `AppWidgetProvider` implementation.
SiteMonitorConfigure.java	Contains the `Activity` used to manipulate a specific entry's data and to view data received from a remote hosted application.
SiteMonitorModel.java	Contains the methods for managing the `SharedPreference`s that store widget-specific data elements.
SiteMonitorService.java	Service responsible for performing the actual monitoring of remote sites. Network communications take place on a background thread.
SiteMonitorBootstrap.java	Contains code related to alarm management and is responsible for triggering the `SiteMonitorService` under various conditions, including alarm firing.
monitor.xml	Defines the user interface elements used by the `AppWidget` on the home screen.

Table 17.2 File listing for this project *(continued)*

Filename	Comment
main.xml	Defines the user interface elements used in the `SiteMonitorConfigure Activity`.
strings.xml	Contains externalized strings; useful for easy management of textual data and for potential localization.

With this foundational understanding of how the various pieces relate to one another, it's time to start looking at the code behind this application. Although it may be tempting to jump into the `AppWidgetProvider` implementation, we first need to look at the code for handling the `AppWidget`-specific data.

17.4 AppWidget data handling

As mentioned earlier, each instantiated `AppWidget` has a unique numeric identifier represented as an integer primitive (`int`). Any time the application is asked to work on a particular `AppWidget`, this identifier value is available to the code. Sometimes it's provided, as in an `Intent`'s extras bundle; in other circumstances a collection of widget identifiers is retrieved from the `AppWidgetManager` as an array of integers (`int []`). Regardless of its source, managing the relationship between this identifier and the `AppWidget` instance-specific data defined by your own applications is crucial for success.

For the SiteMonitor application, all data management is performed by the `SiteMonitorModel` class, contained in the SiteMonitorModel.java source file. The `SiteMonitorModel` class can be broken down into two logical sections: the instance data and methods, and the static method. The instance portion of the class includes a number of `String` member variables, their respective getter and setter methods, and helpful bundling and unbundling methods.

The underlying data storage persistence method is the application's `SharedPreferences`, which we introduced in chapter 5. To keep things simple, every data element is stored as a `java.lang.String`. When the entire "record" needs to be stored, the data elements are combined into a composite delimited `String`. When the data is read out of the `SharedPreferences`, the retrieved `String` is parsed and stored into respective members based on ordinal position in the string. Although this approach is perhaps pedestrian, it's perfectly adequate for our purposes. Alternatively, we could've employed an SQLite database or constructed our own `ContentProvider`, but both of those mechanisms are overkill for this purpose at present. A `ContentProvider` is often only justified if the data needs to be shared with components outside of a single application suite.

The data elements managed for each `AppWidget` include:

- Site name
- Site update URL

- Site home page URL
- Status
- Last status date
- Message/comments

The class also includes four static methods used as helpers to manipulate instances of widget data throughout the application. Three of these methods are related to persistence of SiteMonitorModel data, and a fourth provides date formatting. This date formatting method was included to help standardize and centralize Date string representation.

The following listing presents the implementation of the SiteMonitorModel class, minus a few setter/getters, which are omitted here but are included in the full source listing available for download.

Listing 17.1 SiteMonitorModel class

```
package com.msi.unlockingandroid.sitemonitor;

import java.text.SimpleDateFormat;
import android.content.Context;
import android.content.SharedPreferences;
import android.util.Log;

public class SiteMonitorModel {

    private static final String tag = "SiteMonitorModel";

    private static final String PREFS_NAME
"com.msi.unlockingandroid.SiteMonitor";
    private static final String PREFS_PREFIX = "sitemonitor_";
```

1 Contain constants for SharedPreferences persistence

```
    private String name;
    private String url;
    private String homepageUrl;
    private String status;
    private String statusDate;
    private String message;
```

2 Contain per-widget data elements

```
    public SiteMonitorModel(String name,String url,String homepageUrl,
String status,String statusDate,String message) {
      this.name = name;
      this.url = url;
      this.homepageUrl = homepageUrl;
      this.status = status;
      this.statusDate = statusDate;
      this.message = message;
    }
```

3 Define constructor

```
    public SiteMonitorModel(String instring) {
      Log.i(SiteMonitorModel.tag,"SiteMonitorModel(" + instring + ")");
      String[] data = instring.split("[|]");
      if (data.length == 6) {
        this.name = data[0];
        this.url = data[1];
        this.homepageUrl = data[2];
        this.status = data[3];
        this.statusDate = data[4];
        this.message = data[5];
      } else {
        this.name = "?";
        this.url = "?";
        this.homepageUrl = "?";
        this.status = "WARNING";
        this.statusDate =
java.util.Calendar.getInstance().getTime().toString();
        this.message = "";
      }
    }
```

4 **Define constructor/parser**

```
    public String getName() {
      return this.name;
    }
    public void setName(String name) {
      this.name = name;
    }
```

5 **Define getter/setter**

```
    // see full source code for remaining getter/setter methods

    public String storageString() {
      return this.name + "|" + this.url + "|" + this.homepageUrl + "|" +
this.status + "|" + this.statusDate + "|" + message;
    }
```

6 **Prepare data for storage**

```
    public String toString() {
      return this.storageString();    }
```

7 **Override toString()**

```
    public static void saveWidgetData(Context context,int
widgetId,SiteMonitorModel model) {
      Log.i(SiteMonitorModel.tag,"saveWidgetData(" + widgetId + "," +
model.storageString() + ")");
      SharedPreferences.Editor prefsEditor =
context.getSharedPreferences(PREFS_NAME, 0).edit();
      prefsEditor.putString(PREFS_PREFIX +
widgetId,model.storageString());
      prefsEditor.commit();
    }
```

8 **Save widget data**

```
    public static SiteMonitorModel
getWidgetData(Context context,int widget) {
      Log.i(SiteMonitorModel.tag,"getWidgetData(" + widget + ")");

      SharedPreferences prefs =
      context.getSharedPreferences (PREFS_NAME, 0);
      String ret = prefs.getString(PREFS_PREFIX + widget,"BAD");
      if (ret.equals("BAD")) return null;
      return new SiteMonitorModel(ret);
    }
```

9 **Retrieve widget data**

10 **Return SiteMonitorModel instance**

```
    public static void deleteWidgetData(Context context,int widgetId) {
      Log.i(SiteMonitorModel.tag,"deleteWidgetData(" + widgetId + ")");
      SharedPreferences.Editor prefsEditor =                        ⓫ Remove
context.getSharedPreferences(PREFS_NAME, 0).edit();                    widget
      prefsEditor.remove(PREFS_PREFIX + widgetId);                     data
      prefsEditor.commit();
    }

    public static String getFormattedDate() {
      SimpleDateFormat sdf = new                                    Format
     SimpleDateFormat                                           status date ⓬
("MMM dd HH:mm");
      return sdf.format(java.util.Calendar.getInstance().getTime());
    }
}
```

The SiteMonitorModel class meets the data management needs of the SiteMonitor AppWidget. The underlying data persistence method is the application Shared-Preferences and as such a couple of constant String values are employed ❶ to identify the SharedPreferences data. Each data element is defined as a String member variable ❷. Two distinct constructors are employed. The first constructor ❸ is for creating a new instance from distinct String values, and the second constructor ❹ is used to parse out data for an existing widget that has been retrieved from a Shared-Preference.

Only one set of getter/setter methods ❺ is shown in this listing, but they all employ the same basic bean pattern.

When preparing data to be stored in the SharedPreferences, the widget instance data is reduced to a single delimited String with the assistance of the storageString method ❻. The toString() method ❼ is overridden and invokes the storageString method.

Data storage, retrieval, and deletion are handled in statically defined methods. The saveWidgetData ❽ method stores the widget data with a key of PREFIX_NAME + widgetIdentifier. This means that the data for a widget with an ID of 200 would look like this:

```
sitemonitor_200 = "sitename|url|homepageurl|status|statusDate|message"
```

For more specifics on using SharedPreferences, refer to chapter 5.

Widget data is retrieved from SharedPreferences in the getWidgetData method ❾. This method returns a SiteMonitorModel ❿ by employing the parsing version of the constructor ❹.

When a widget is removed from the device, you delete the associated data with a call to deleteWidgetData ⓫.

Finally, the getFormattedDate method ⓬ is responsible for formatting a java.util.Date string into a String representation with the help of a SimpleData-Format class.

At this point you should have a good feel for what data is managed and where it lives. Let's get to the code to actually implement an AppWidget!

17.5 Implementing the AppWidgetProvider

The AppWidgetProvider for the SiteMonitor application is implemented in the file SiteMonitorWidgetImpl.java. The AppWidgetProvider is responsible for handling updates to the UI as well as responding to housekeeping events related to the AppWidget lifecycle, and is arguably the most important aspect of AppWidget programming to understand. Because of its centrality and importance to working with AppWidgets, we're going to look at the code from two perspectives.

17.5.1 AppWidgetProvider method inventory

The methods presented in table 17.3 represent the core AppWidgetProvider functionality. Although these methods are common to AppWidgetProviders, the comments are made in the context of the SiteMonitor application. Also, the final two methods (denoted with *) are custom to the SiteMonitor application.

Table 17.3 Inventory of AppWidgetProvider methods

Method name	Comment
onReceive	This is the same method found in all BroadcastReceiver classes. It's used to detect ad hoc update requests, which it then hands off to the onUpdate method.
onUpdate	This method is responsible for updating one or more widget instances. The method receives an array of widget identifiers to be updated.
onDeleted	This method is invoked when one or more widgets are deleted. Like the onUpdate method, this method receives an array of widget identifiers—in this case each of these widgets has just been deleted. This method is responsible for cleaning up any data stored on a per-widget basis.
onEnabled	This method is invoked when the first AppWidget instance is placed on the home screen. In SiteMonitor this method initiates an Alarm sequence, which forces an update on a specific interval as defined within the SiteMonitorBootstrap class.
onDisabled	This method is invoked when the final AppWidget instance is removed from the home screen. When there are no instances of the SiteMonitor widget, there's no need for updating them. Therefore the alarm is cleared. This method doesn't reliably get called when you think it ought to be invoked.
*UpdateOneWidget	This static method is responsible for performing the update on a specific widget. Because there are multiple scenarios for interacting with the AppWidgets in our class, it was desirable to consolidate all widget UI impacting code into a single method.
*checkForZombies	The AppWidget subsystem has a nasty habit of leaving widgets behind without an effective means of cleaning them up short of a reboot. Consequently our AppWidgetProvider instance is consistently being asked to perform operations on widgets that don't exist any longer. This method is used as a helper to the onDisabled method. Every time the onDelete method is invoked, call this method to perform an additional cleanup step. When no legitimate widgets are detected, clear the update alarm, performing the job that the onDisabled method can't reliably perform.

You now know what the method names are and the responsibility of each. It's time to examine the code. Let's begin with the implementation of an `AppWidgetProvider` as we look at SiteMonitorWidgetImpl.

17.5.2 Implementing SiteMonitorWidgetImpl

There's a lot of code to examine in this class, so we're going to break it into a couple of sections. In listing 17.2 you can see the basic callbacks or hooks that respond to the `AppWidget` events. As you know, every widget in the system has an integer identifier. When you're working with `AppWidgets`, it's common to manipulate an array of these identifiers, as shown in the upcoming listings, so keep an eye out for those identifiers as you review the code.

Listing 17.2 `SiteMonitorWidgetImpl`, which implements `AppWidget` functionality

```
package com.msi.unlockingandroid.sitemonitor;

import android.content.Context;
import android.content.ComponentName;
import android.content.Intent;
import android.app.PendingIntent;
import android.appwidget.AppWidgetProvider;
import android.appwidget.AppWidgetManager;
import android.widget.RemoteViews;
import android.net.Uri;
import android.util.Log;
import android.graphics.Color;

public class SiteMonitorWidgetImpl extends AppWidgetProvider {
    private static final String tag = "SiteMonitor";
    public static final String UPDATE_WIDGETS =
"com.msi.unlockingandroid.sitemonitor.UPDATE_WIDGETS";

    @Override
    public void onUpdate(Context context,AppWidgetManager
appWidgetManager,int[] appWidgetIds ) {

      super.onUpdate(context, appWidgetManager, appWidgetIds);
      int count = appWidgetIds.length;
      Log.i(SiteMonitorWidgetImpl.tag,"onUpdate::" + count);
      // we may have multiple instances of this widget ... make
sure we hit each one ...
      for (int i=0;i<count;i++) {
        SiteMonitorWidgetImpl.UpdateOneWidget
(context, appWidgetIds[i]);
      }
    }

    public void onDeleted(Context context,int[] appWidgetIds) {
      super.onDeleted(context, appWidgetIds);
      Log.i(SiteMonitorWidgetImpl.tag,"onDeleted()" + appWidgetIds.length);
      for (int i = 0;i<appWidgetIds.length;i++) {
        SiteMonitorModel.deleteWidgetData(context, appWidgetIds[i]);
```

1 AppWidget imports

2 String used for updates

3 onUpdate method

4 Update of single widget

5 onDeleted method

```
      }
      checkForZombies(context);            ❻ checkForZombies
    }

    public void onEnabled(Context context) {                         Provider
      Log.i(SiteMonitorWidgetImpl.tag,"onEnabled");                  enabled
      super.onEnabled(context);

      // set up the recurring alarm that drives our refresh process  ❼
      SiteMonitorBootstrap.SetAlarm(context);
    }
    public void onDisabled(Context context) {
      Log.i(SiteMonitorWidgetImpl.tag,"onDisabled()");
      super.onDisabled(context);                                 Provider
      // kill the recurring alarm that drives our refresh process ❽ disabled
      SiteMonitorBootstrap.ClearAlarm(context);
    }

    public void onReceive(Context context,Intent intent) {
      super.onReceive(context, intent);            ❾ onReceive override
      Log.i(SiteMonitorWidgetImpl.tag,"onReceive()::" +
 intent.getAction());

      if (intent.getAction().equals                      ❿ Check for
(SiteMonitorWidgetImpl.UPDATE_WIDGETS)) {                   update request

        Log.i(SiteMonitorWidgetImpl.tag,
"Updating widget basedon intent”);AppWidgetManager appWidgetManager =
AppWidgetManager.getInstance(context);
        int [] ids = appWidgetManager.getAppWidgetIds
(new ComponentName(context,SiteMonitorWidgetImpl.class));   List of ⓫
        onUpdate(context,appWidgetManager,ids);             widgets
      } // trace me

    }
public static void UpdateOneWidget(Context context,int widgetNumber) {
    // shown in Listing 17.5
}

private void checkForZombies(Context context) {
    // shown in Listing 17.3
}
```

The first thing to observe in listing 17.2 is the presence of some imports ❶ that provide resolution for the AppWidget-related classes. A String constant ❷ is defined for comparing against Intents received by this class in the onReceive method ❾. Note that the protection level of this constant is public. It's declared as a public member because this String is used by other classes in the application to trigger an update request.

The onUpdate method ❸ is invoked both periodically based on the widget's update frequency as well as on an ad hoc basis. Note that when this update occurs, it's simply performing a refresh of the AppWidget UI. The actual refreshing of the underlying data model is a separate and distinct operation, which is discussed in detail in section 17.8.

Once you have a list of widgets that require updating, each is updated in turn with a call to `SiteMonitorWidgetImpl.UpdateOneWidget` ❹. This method is defined as a static method, as it's also called from the `SiteMonitorConfigure` `Activity`.

The `onDeleted` method ❺ handles the scenario where a widget is removed from the home screen. When this method is invoked, it in turn calls the super class's `onDeleted` method. Next it removes the data related to each deleted widget with a call to `SiteMonitorModel.deleteWidgetData`. Finally, this method wraps up with a call to check for zombie widgets (which we describe in a moment) by calling `checkForZombies` ❻. It's not uncommon to have a widget identifier allocated but the widget itself not actually created. An example of this is when a configuration activity is launched but then canceled by the user—no widget data gets created so you wind up with widget identifiers not attached to meaningful widget data. The reason you want to track this situation is to disable the update alarm when no legitimate widgets remain. Also, note that the arguments to this method include an array of integers representing the list of deleted widget instances. This array will usually consist of only a single widget.

When the `AppWidgetProvider` is enabled ❼, the update alarm is "set." Note that the `onEnabled` method can be used for other housekeeping setup tasks as well. This method is triggered when the first `AppWidget` is created.

The mirror image of the `onEnabled` method is the `onDisabled` method ❽. This method cancels the alarm set previously in the `onEnabled` method. This method isn't called when you think it ought to be! Why? Because of the "zombie" widgets that lurk about in the ether. It's for this reason that the `checkForZombies` method was added to this class, accommodating for the scenario where there are no active widgets but the operating system believes they still exist. These widgets will persist until the device is rebooted. The moral of the story here is that although these callback methods are nice to have, it's ultimately up to the developer to manage around the system. This will in all likelihood be rectified in future releases but as of Android version 2.2, this "feature" remains.

Rounding out this code listing, you see the `onReceive` method ❾. This is the same method required of all `BroadcastReceiver` implementations—recall that `AppWidgetProvider` extends the `BroadcastReceiver`. The super class's `onReceive` method is invoked and then the `Intent` is examined ❿. If the `Intent` matches the special update constant defined in this class, `SiteMonitorWidgetImpl.UPDATE_WIDGETS`, the code gathers a list of relevant widget identifiers ⓫ and passes them to the `onUpdate` method to be refreshed visually.

17.5.3 Handling zombie widgets

We've discussed at length the nature of the relationship between the widget identifier and the widget data as defined and managed in the `SiteMonitorModel` class. There are a number of places in the application where a widget identifier is available and the code needs to check for the presence of legitimate data. One example of this is the processing that occurs after a widget instance is deleted. Listing 17.3 demonstrates a technique for keeping track of legitimate versus zombie widget identifiers. If no legitimate identifiers are found, the code disables the alarm—there's no need to take up more system resources than necessary.

Listing 17.3 Dealing with `AppWidgets` that won't die off

```
private void checkForZombies(Context context) {
      Log.i(SiteMonitorWidgetImpl.tag,"checkForZombies");
      AppWidgetManager appWidgetManager =
AppWidgetManager.getInstance(context);
      int [] ids = appWidgetManager.getAppWidgetIds
(new ComponentName(context,SiteMonitorWidgetImpl.class));
      int goodCount = 0;
      for (int i=0;i<ids.length;i++) {
        SiteMonitorModel smm =
SiteMonitorModel.getWidgetData(context,ids[i]);
        if (smm != null) goodCount++;
      }
      if (goodCount == 0) {
        Log.i(SiteMonitorWidgetImpl.tag,
"There are no good widgets left! Kill alarm!");
     SiteMonitorBootstrap.clearAlarm(context);
      }
    }
```

❶ Reference AppWidgetManager

❷ Get list of widgets

❸ Attempt to load data

❹ Clear alarm

All interactions with widgets rely on the availability of a valid widget identifier. To obtain this information, the code must have access to the `AppWidgetManager`, which is obtained with a call to that class's static `getInstance` method ❶. The `AppWidgetManager` has a method named `getAppWidgetIds` ❷ that takes a `ComponentName` argument. For each widget identifier, you attempt to load widget-specific data ❸. If no valid widget identifier-to-data relationships are found, you can clear the alarm with a call to `SiteMonitorBootstrap.clearAlarm` ❹.

We have one method remaining to review: `UpdateOneWidget`. This method covers such a broad range of topics that it's discussed in its own section, coming up next.

17.6 Displaying an AppWidget with RemoteViews

An `AppWidget` runs in the process space of another application, typically the home screen. Running in the space of another application has an impact on what can and can't be accomplished when interacting with UI elements. This section demonstrates how an `AppWidget`'s user interface is managed through the use of the `RemoteViews` class.

17.6.1 Working with RemoteViews

The `RemoteViews` class is used to permit a `View` to be displayed and managed from a separate process. Unlike a traditional `ViewGroup` layout, which may be readily managed via direct methods, the access to the view hierarchy inflated under the `RemoteViews` class is limited and rigid. For example, in a typical `Activity` the code can inflate a layout by simply referencing it as `R.layout.main` and passing it to the method `setContentView`. `AppWidgets` require more effort than this.

The `RemoteViews` class offers two constructors. The one of most interest to us is defined as

```
RemoteViews(String packageName, int layoutId)
```

The `packageName` can be obtained with a call to `context.getPackageName()` and the `layoutId` is a layout defined in the normal manner using a subset of the familiar widgets such as `TextView` and `Button`.

```
RemoteViews rv = new RemoteViews(context.getPackageName(),R.id.monitor);
```

Once a reference to a "remote view" is available, a proxy-like mechanism is available for setting and getting basic properties of views contained within the view hierarchy loaded by the `RemoteViews` instance. Here's an example of changing the text in a `TextView` identified by `R.id.someTextView`:

```
rv.setTextViewText(R.id.someTextView,"Fat cats buy ice");
```

Another available method is `setInt`, which passes an integer value to a named method on a specified view. For example, to change the height of the view to 200 pixels, you can use the following code:

```
rv.setInt(R.id.someTextView,"setHeight",200);
```

When you're working with `RemoteViews`, the only user interaction you can trap is a click. You do so by passing a `PendingIntent` to the `setOnClickListener` method. The following listing demonstrates this procedure.

Listing 17.4 Setting up a `PendingIntent` to handle user interactions

```
Intent onClickedIntent = new Intent(context,SomeClass.class);
PendingIntent pi = PendingIntent.getActivity(context, 0, onClickedIntent, 0);
rv.setOnClickPendingIntent(R.id.someButton, pi);
```

To make a `RemoteViews`-based click handler, the first step is to create a new `Intent`. Initialize the `Intent` in any manner appropriate for your task. In this example, you define an `Intent` to launch a specific `Activity`. Next, create a `PendingIntent` using the `getActivity` static method. One of the arguments to this method is the previously created `Intent`. Then a call to the `setOnClickPendingIntent` method passing the `PendingIntent` will wire up the desired behavior.

Now that you have a feel for how `RemoteViews` operate, let's finish up the discussion of the `SiteMonitorWidgetImpl` class's code.

17.6.2 *UpdateOneWidget explained*

All of the code presented thus far for the `SiteMonitorWidgetImpl` class has taken care of the plumbing and scaffolding of the operation of our `AppWidget`. It'd be nice to get something onto the screen of the phone! That's where the `UpdateOneWidget` method comes into play. Recall that the onUpdate method delegated the responsibility of updating the screen to this method. The following listing demonstrates updating the widget with the help of RemoteViews.

Listing 17.5 Updating the widget with RemoteViews in SiteMonitorWidgetImpl

```
  public static void UpdateOneWidget(Context context,int widgetNumber) {
   Log.i(SiteMonitorWidgetImpl.tag,"Update one widget!");
   AppWidgetManager appWidgetManager =
 AppWidgetManager.getInstance(context);
   SiteMonitorModel smm = SiteMonitorModel.getWidgetData
(context, widgetNumber);

   if (smm != null) {
    Log.i(SiteMonitorWidgetImpl.tag,"Processing widget " +
smm.toString());

    RemoteViews views = new
RemoteViews(context.getPackageName(),R.layout.monitor);
    if (smm.getStatus().equals("GOOD")) {
     views.setTextColor(R.id.siteName, Color.rgb(0,255,0));
     views.setTextColor(R.id.updateTime, Color.rgb(0,255,0));
     views.setTextColor(R.id.siteMessage, Color.rgb(0,255,0));
    } else if (smm.getStatus().equals("UNKNOWN")){
     views.setTextColor(R.id.siteName, Color.rgb(255,255,0));
     views.setTextColor(R.id.updateTime, Color.rgb(255,255,0));
     views.setTextColor(R.id.siteMessage, Color.rgb(255,255,0));
    } else {
     views.setTextColor(R.id.siteName, Color.rgb(255,0,0));
     views.setTextColor(R.id.updateTime, Color.rgb(255,0,0));
     views.setTextColor(R.id.siteMessage, Color.rgb(255,0,0));
    }
    views.setTextViewText(R.id.siteName, smm.getName());
    views.setTextViewText(R.id.updateTime, smm.getStatusDate());

    // make this thing clickable!
    Intent intWidgetClicked = new
Intent(context,SiteMonitorConfigure.class);
    intWidgetClicked.setData(Uri.parse("file:///bogus" +
widgetNumber));
    intWidgetClicked.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,
 widgetNumber);
    PendingIntent pi = PendingIntent.getActivity(context, 0,
intWidgetClicked, 0);
    views.setOnClickPendingIntent(R.id.widgetLayout, pi);
    appWidgetManager.updateAppWidget(widgetNumber,views);
   }
   else {
    Log.i(SiteMonitorWidgetImpl.tag,"Ignore this widget # " +
widgetNumber + ".  Must be a zombie widget.");
   }
  }
```

❶ **Acquire AppWidgetManager reference**

❷ **Load widget data**

❸ **Create RemoteViews instance**

❹ **Format TextViews**

❺ **Assign text values**

❻ **Assign unique data**

❼ **Create PendingIntent**

Like virtually everything related to AppWidget programming, the first thing to do is acquire a reference to the AppWidgetManager ❶. Next, you load the widget-specific data associated with this widget identifier with `SiteMonitorModel.getWidgetData` ❷ and confirm that the data is valid. Assuming you have a good widget to work with, you next create an instance of the `RemoteViews` class ❸, passing in an identifier for your preferred layout `R.id.monitor`. Based on the status, you assign different `TextColor` values ❹ to each of the visible `TextViews` within the layout, as well as populate the controls with the actual textual values for display with calls to `setTextViewText` ❺.

At this point, your widget is now ready for display. You'd also like the user to be able to tap on the widget and bring up related information. To do this, you must assign a `PendingIntent` to a view within the view hierarchy represented by the `RemoteViews` instance you previously instantiated.

To begin, you create an `Intent` referencing your configuration activity `SiteMonitorConfigure`. You next assign data related to this `Intent` with a call to the `setData` method ❻. Note that the data here isn't particularly important, as long as it's unique. The reason for this is related to the manner in which `PendingIntents` are resolved. Without this uniqueness, each subsequent `PendingIntent` assignment would replace the previously assigned `Intent`. By adding this custom and unique data to the `Intent`, your `PendingIntent` becomes unique per widget. If you doubt this, just comment out this line and find out what happens!

Next you assign the `widgetNumber` to the key `AppWidget.EXTRA_APPWIDGET_ID`. This is used to make things a bit easier in the `SiteMonitorConfigure Activity`, which is discussed in the next section. A `PendingIntent` is created, requesting an `Activity` ❼, and finally this `PendingIntent` is assigned to your widget via the `setOnClickPendingIntent` method. One piece of trivia to note here is that you've passed in an ID for the `LinearLayout` of the user interface. Layouts often don't have ID attributes associated with them. Layouts are instances of `ViewGroups`, which are extensions of the `View` class, so there's no reason why you can't assign an ID to the layout itself. The net effect is that your entire widget is clickable. Considering the fact that it's a mere 74 pixels square, this is a reasonable approach.

At this point, much of the heavy lifting for our `AppWidget` is behind you. Let's look at some of the details associated with configuring a specific instance of an `AppWidget` next, as we examine the `SiteMonitorConfigure Activity`.

17.7 Configuring an instance of the AppWidget

There are two scenarios where our `AppWidget` may be configured. The first is right after the user requests its creation, and the second is when the user taps on an existing widget instance on the home screen.

Generally speaking, this `Activity` operates just as any other `Activity` you've experienced throughout the book. It inflates a layout, gets references to the various `Views`, and responds to user input. Only a couple of items are worthy of highlighting here, but they're important details that you must implement.

Let's start with how Android knows which `Activity` to launch after a new instance is created. To do that, we'll take a brief side trip to look at the metadata related to this `AppWidget`.

17.7.1 AppWidget metadata

Earlier we alluded to a special metadata file that defines attributes for an AppWidget. This file is associated with a specific receiver entry in the AndroidManifest.xml file. Within the metadata file, you can associate a specific Activity as the preferred configuration tool. The following listing presents the sitemonitorwidget.xml file. Even though our focus in this section is on the Activity, this is a good opportunity to tie together a couple of ideas you've learned to this point.

Listing 17.6 AppWidget metadata file defining widget characteristics

```
<?xml version="1.0" encoding="utf-8"?>
<appwidget-provider
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:minHeight="72dp"
  android:minWidth="72dp"
  android:initialLayout="@layout/monitor"
  android:updatePeriodMillis="0"
  android:configure=
"com.msi.unlockingandroid.sitemonitor.SiteMonitorConfigure"
  android:icon="@drawable/icon"
  >
</appwidget-provider>
```

Earlier in this chapter we described the screen real estate consumed by an AppWidget. The height and width are specified as minimum values. Each of the available "spaces" or *cells* in the screen is 74 pixels square. The formula for deriving the values here is the number of cells requested times 74 minus 2.

The initial layout used by the widget is defined in the initialLayout attribute. At runtime the application is free to change the layout, but you should consider the fact that by the time your widget is ready for updating, it's already been placed on the screen, so your expectations might be shattered if you were hoping to bump some other widget out of the way!

The updatePeriodMillis specifies the update interval. Based on the architecture of the SiteMonitor, this has little importance, so set it to 0 to tell the widget not to bother waking itself up to update. Setting this attribute to a nonzero value causes the device to wake up periodically and call the onUpdate method in the AppWidgetProvider implementation.

Finally, you see the configure attribute, which permits you to specify the fully qualified class name for the Activity to be launched when the user selects this widget from the list of available widgets on the home screen. When the user is selecting from the list of widgets, the icon displayed in the list is defined by the icon attribute.

Now that the Activity is associated with our AppWidget, it's time to examine the key elements of the Activity. The full code is available for download. The snippets shown here are only the portions particularly relevant to AppWidget interactions.

17.7.2 Working with Intent data

When the AppWidget's configuration Activity is launched, the most important piece of information is the associated widget identifier. This value is stored as an extra in the Intent and should be extracted during the onCreate method. The following listing demonstrates this technique.

Listing 17.7 Setting up the configuration Activity to manage a widget instance

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

        etSiteName = (EditText) findViewById(R.id.etSiteName);
        etSiteURL = (EditText) findViewById(R.id.etSiteURL);
        etSiteHomePageURL = (EditText)
 findViewById(R.id.etSiteHomePageURL);
        tvSiteMessage = (TextView) findViewById(R.id.tvSiteMessage);

        final Button btnSaveSite =
(Button) findViewById(R.id.btnSaveSite);
        btnSaveSite.setOnClickListener(this);
        final Button btnVisitSite =
(Button) findViewById(R.id.btnVisitSite);
        btnVisitSite.setOnClickListener(this);

        widgetId =
getIntent().getIntExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,widgetId);

        // lookup to see if we have any info on this widget
        smm = SiteMonitorModel.getWidgetData(this, widgetId);
        if (smm != null) {
        etSiteName.setText(smm.getName());
          etSiteURL.setText(smm.getUrl());
          etSiteHomePageURL.setText(smm.getHomepageUrl());
          tvSiteMessage.setText(smm.getMessage());
        }
    }
```

Wire up GUI ❶

Extract widget identifier ❷

❸ Look up widget data

❹ Populate GUI

The Activity looks like boilerplate code, as it begins with wiring up the various view elements in the layout to class-level variables ❶. The widgetId is extracted from the startingIntent ❷. Again you see the relationship between widgetId and widget-specific data managed by the SiteMonitorModel class ❸. If data is available, the GUI elements are prepopulated with the values ❹. This scenario would only come into play after the widget has been successfully created and subsequently clicked for managing it.

At this point, the Activity operates as expected, permitting the user to update the details of the widget data as well as visit the associated website.

17.7.3 Confirming widget creation

When the user has populated the required fields and hits the Save button, you need to not only save the data via the SiteMonitorModel class but also let the AppWidget infrastructure know that you've affirmed the creation of this widget instance. This takes place by using the Activity's setResult method along with an Intent containing an extra indicating the widget number. In addition you want to ensure that the alarm is enabled for future updates. Finally, you really don't want to wait until the next alarm interval elapses; you want to get an update now. The following listing demonstrates how to accomplish each of these tasks.

Listing 17.8 Handling button clicks in the configuration Activity

```
public void onClick(View v) {
    switch (v.getId()) {
      case R.id.btnSaveSite: {                               1 Save data
        saveInfo();                                                          Update
        // update the widget's display                                   widget's UI 2
        SiteMonitorWidgetImpl.UpdateOneWidget(v.getContext(), widgetId);

        // let the widget provider know we're done and happy
        Intent ret = new Intent();
        ret.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,widgetId);
          setResult(Activity.RESULT_OK,ret);                                     3

        // let's ask for an update and also enable the alarm
        Intent askForUpdate =                                  5 Set alarm
new Intent(SiteMonitorBootstrap.ALARM_ACTION);
        this.sendBroadcast(askForUpdate);
        SiteMonitorBootstrap.setAlarm(this);
        finish();
      }
      break;
      case R.id.btnVisitSite: {                              1 Save data
        saveInfo();
        Intent visitSite = new Intent(Intent.ACTION_VIEW);        6 Visit site
        visitSite.setData(Uri.parse(smm.getHomepageUrl()));         home page
        startActivity(visitSite);
      }
      break;
    }
    }
```

When the user clicks the Save button (or the Visit Site button), the widget-specific data is saved 1. There's nothing fancy there—just a call to SiteMonitorModel.saveWidgetData(). The AppWidget subsystem is supposed to update the UI of the widget after the Configuration dialog box completes successfully, but experience shows that this isn't always the case. Therefore a call is made to SiteMonitorWidgetImpl.UpdateOneWidget with the newly created widgetId 2.

An important step in the life of a new AppWidget is to be sure to set the Activity result to RESULT_OK ❸, passing along an Intent extra that identifies the new widget by number.

At this point our new widget is populated with a name and no meaningful status information. To force an update, you broadcast an Intent that simulates the condition where the alarm has just triggered ❹. You also want to ensure that the alarm is armed for a subsequent operation, so you call SiteMonitorBootstrap.setAlarm ❺.

In the event that the Visit Site button is clicked, you want to take the user to the defined home page of the currently active site being monitored by the widget ❻.

The last condition to handle is the case where the widget has been selected from the Add New Widget list on the home screen when the user cancels out of the configuration activity. In this case the widget shouldn't be created. To achieve this result, you set the result of the Activity to RESULT_CANCELED as shown in this listing.

Listing 17.9 Checking for a cancel

```
public void onDestroy() {
 super.onDestroy();
 if (!isFinishing()) {
   Intent ret = new Intent();
   ret.putExtra(AppWidgetManager.EXTRA_APPWIDGET_ID,widgetId);
   setResult(Activity.RESULT_CANCELED,ret);
 }
```

This code overrides the onDestroy method. If this method is invoked for any reason other than finish() was called by the Activity itself, you want to "cancel" the Activity. This is accomplished with a call to setResult with the inclusion of the Intent extra to pass along the widget identifier. Note that this cancel step is only required when the widget instance is first created. There's no harm in setting the result for future invocations of the Activity.

At this point our AppWidget is created, you know how to store data, and you know how to configure a particular instance of the SiteMonitor widget. What's needed next is to update the data. For that, we'll look at the SiteMonitorService.

17.8 Updating the AppWidget

Our AppWidget instances on the home screen are only useful if they're actually keeping data up-to-date. One means for performing this update is to solely rely on the AppWidgetProvider's onUpdate method. The onUpdate method is invoked periodically according to the schedule specified in the metadata file.

One reason for updating the widget outside of the AppWidgetProvider is to provide more granularity than afforded by the built-in scheduling mechanism. For example, you may wish to update the data more frequently than once per hour depending on the conditions. Imagine an AppWidget that tracks stock prices. It makes little sense to have the widget update when the market is closed, and it wouldn't be unreasonable to update once every 15 minutes during market hours.

Fortunately, the `SiteMonitorWidgetImpl` can process ad hoc updates. You're already set up for this because you're overriding the `onReceive` method. When the `onReceive` method receives an `Intent` with an `Action` equaling `SiteMonitorWidgetImpl.UPDATE_WIDGETS`, the widgets are updated on the home screen. So, now all you need to do is sort out how (and when) to update the underlying data.

Beyond the `AppWidget` built-in scheduler, there are basically two mechanisms for periodic updates. The first approach is to create a `Service` that periodically performs an update. The other approach is to set an `Alarm` that triggers the update on a recurring basis. Let's take a brief look at both strategies.

17.8.1 Comparing services to alarms

A constantly running `Service` is capable of performing this periodic update; it's the ideal place to perform operations that are to be carried out in the background. An Android `Service` is the appropriate vehicle for performing the update work—talking to a remote hosted application—but what about all the idle time between updates? Does the application need to be running, consuming resources but adding no additional value? Also, what happens if the `Service` stops running? Do you configure it to be restarted automatically? Perhaps it should be kept running at all times, but at what impact on the device's overall performance? You need to conserve battery resources as well as keep the load on the CPU as low as feasible to improve overall device responsiveness. This approach isn't ideal because a long-running `Service` that's idle the majority of the time consumes resources unnecessarily and brings little overall benefit.

What about an alarm? An alarm can be set to trigger once or periodically. Additionally, an alarm may be configured to trigger only when the device is awake—if the widget doesn't need to be updated while you and your device are sleeping, there's no need to unnecessarily expend battery resources to perform an update you may never see! Also, the Alarm Manager can group a number of periodic alarms so that the device wakes up for short windows of activity rather than every time an alarm may request it. By giving Android some latitude on when an alarm can be fired, you can further manage battery resources for longer life.

As much as we like the alarm approach, there's one thing you need to keep in mind: an alarm has no ability in and of itself to perform any real activity beyond signaling. The alarm can signal and the `Service` can perform the actual update, so the correct answer here is to use both an alarm and a `Service`. And for good measure we're going to use a `BroadcastReceiver` to add more flexibility into the mix. Looking back at figure 17.9, you see that the preferred architecture is to have an alarm send an `Intent` via broadcast to a `BroadcastReceiver`, which in turn initiates a `Service` to perform the update process—fetching data over the internet from the various data sources. Once the data is fetched and the `SiteMonitorModel` updated, you can request that the widgets themselves be refreshed visually and the `Service` can shut down. It'll be started again for the next periodic update.

Let's see how the alarm is managed.

17.8.2 Triggering the update

Services have been covered in various portions of this book already, so they'll receive relatively little coverage here. Of more interest in this section is the relationship between the alarm, the `BroadcastReceiver`, and the `Service`. Let's look first at the code in SiteMonitorBootstrap.java.

Listing 17.10 SiteMonitorBootstrap.java

```
package com.msi.unlockingandroid.sitemonitor;

import android.app.AlarmManager;
import android.app.PendingIntent;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.SystemClock;
import android.util.Log;

public class SiteMonitorBootstrap extends BroadcastReceiver {
    private static final String tag = "SiteMonitorBootstrap";
    public static final String ALARM_ACTION =                    1 Set up
"com.msi.unlockingandroid.sitemonitor.ALARM_ACTION";               Intent action

    private static final long UPDATE_FREQUENCY =              2 Update
(1000 * 60 * 60);                                               frequency
// default to one hour

    @Override
    public void onReceive(Context context, Intent intent) {
      String action = intent.getAction();
                                                                  3 Check for
      Log.i(SiteMonitorBootstrap.tag,"onReceive");                  update
                                                                    trigger
      if (action.equals(SiteMonitorBootstrap.ALARM_ACTION)) {
        Log.i(SiteMonitorBootstrap.tag,
"Alarm fired -- start the service to perform the updates");
        Intent startSvcIntent = new                                 4 Start
Intent(context,SiteMonitorService.class);                             service
        startSvcIntent.putExtra("ALARMTRIGGERED", "YES");
        context.startService(startSvcIntent);
      }
    }

    public static void setAlarm(Context context) {          Get AlarmManager 5
      Log.i(SiteMonitorBootstrap.tag,"setAlarm");                 reference
      AlarmManager alarmManager =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);

      // setup pending intent
      Intent alarmIntent = new Intent(SiteMonitorBootstrap.ALARM_ACTION);
      PendingIntent pIntent = PendingIntent.getBroadcast(context, 0,
 alarmIntent, PendingIntent.FLAG_UPDATE_CURRENT);
                                                       Create PendingIntent 6
      // now go ahead and set the alarm
```

```
   alarmManager.setRepeating(AlarmManager.ELAPSED_REALTIME_WAKEUP,
 SystemClock.elapsedRealtime() + SiteMonitorBootstrap.UPDATE_FREQUENCY,
SiteMonitorBootstrap.UPDATE_FREQUENCY, pIntent);
  }                                                       Arm alarm ❼

  public static void clearAlarm(Context context) {
   Log.i(SiteMonitorBootstrap.tag,"clearAlarm");
   AlarmManager alarmManager =
(AlarmManager)context.getSystemService(Context.ALARM_SERVICE);

   // cancel the pending intent!
   Intent alarmIntent = new Intent(SiteMonitorBootstrap.ALARM_ACTION);
   PendingIntent pIntent = PendingIntent.getBroadcast(context, 0,
alarmIntent, PendingIntent.FLAG_UPDATE_CURRENT);

   alarmManager.cancel(pIntent);                 ❽ Cancel alarm

  }

}
```

The SiteMonitorBootstrap class is responsible for managing alarms within the application as well as the starting the SiteMonitorService as needed. This class is a BroadcastReceiver and as such overrides the onReceive method, looking for Intents ❸ that it can process. The only IntentFilter set up for this class is for the SiteMonitorBootstrap.ALARM_ACTION ❶. This constant is declared as a public static member so it can be accessed from other components as well as the local alarm. The update frequency is fixed and set in the UDPATE_FREQUENCY constant ❷. An enhancement for this type of application would be to make this setting user configurable in some fashion. When an ALARM_ACTION Intent is encountered, an Intent is created to start the SiteMonitorService ❹ It's the responsibility of the SiteMonitorService to perform the update of the configured SiteMonitor widgets.

This class can readily be extended to detect other events such as the device booting or power events. For example, if it's detected that the device is being charged—a nonbattery power source is detected—the frequency of updates could be increased. Similarly, if it's detected that the device is now roaming, this could suspend the update process to minimize any data roaming charges.

Beyond acting as a BroadcastReceiver to react to events of interest, this code also contains two static methods for setting and clearing the application-defined alarm. Let's walk through the two routines, starting with the setAlarm method. The first step in working with an alarm is to obtain a reference to the AlarmManager ❺. When setting an alarm, you must create a PendingIntent that contains the Intent to be dispatched when the alarm triggers. In this case you create a PendingIntent ❻ that represents a subsequent Broadcast of the SiteMonitorBootstrap.ALARM_ACTION Intent. In essence, you're setting an alarm to send an Intent to this BroadcastReceiver, which in turn initiates the update process. Once the PendingIntent is set up, it's passed as an argument to the setRepeating method ❼. When canceling the alarm, the steps are the same, except you call cancel ❽.

You've done a lot of setup, and now it's time to look at the code that performs the update of the underlying widget-specific hosted application status data.

17.8.3 *Updating the widgets, finally!*

Once the SiteMonitorService code is in control, updating the widgets is rather easy. Updating the SiteMonitorModel data is the sole responsibility of the SiteMonitorService. To accomplish this task, the SiteMonitorService performs these basic operations:

1. Starts a thread to perform the update
2. Updates each of the sites in turn
3. Shuts itself down to conserve resources

The Service creates a separate Thread for performing the update, because by default the Service attempts to run on the main GUI thread of the device. Running in a separate Thread of execution inside a Service is arguably the best place to perform a potentially blocking operation such as the operation required of the SiteMonitorService.

Let's look at the code structure of the SiteMonitorService class.

Listing 17.11 SiteMonitorService class

```
package com.msi.unlockingandroid.sitemonitor;

import android.app.Service;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.os.IBinder;
import android.util.Log;
import android.appwidget.AppWidgetManager;


import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;

public class SiteMonitorService extends Service {

    private static final String tag = "SiteMonitorService";


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


    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
      Log.i(SiteMonitorService.tag,"onStartCommand");

      Thread smu = new Thread(new
SiteMonitorUpdater(this.getBaseContext()));
      smu.start();
```

1 Create, start new Thread

```
      return Service.START_NOT_STICKY;                  ❷ Don't restart service
    }

    @Override
    public void onDestroy() {
      super.onDestroy();
      Log.i(SiteMonitorService.tag,"onDestroy");
    }

    class SiteMonitorUpdater implements Runnable {      ❸ Define SiteMonitorUpdater
      private static final String tag = "SiteMonitorUpdater";

      private Context context;

      public SiteMonitorUpdater(Context context) {
        this.context = context;
      }

      public void run() {
        Log.i(SiteMonitorUpdater.tag,"Running update code");
        updateAllSites();                               ❹ Update all sites
        stopSelf();                ❺ Stop service
      }

      private void updateAllSites() {
        // discussed in Listing 17.12
      }

      private void updateOneSite(SiteMonitorModel smm,int widgetId) {
        // discussed in Listing 17.12
      }

      private String getDataFromSite(String siteUrl) {
        // discussed in Listing 17.12
      }

    }
}
```

The first step this service takes is the creation of a new `Thread` ❶ based on the `SiteMonitorUpdater` class ❸. Once created, the thread is started. The `Service` then returns `Service.START_NON_STICKY` ❷. This tells Android not to restart the `Service` if it either crashes or is killed by the operating system. Because our `Service` will be started periodically, there's no need to have the operating system restart it.

All the update code resides in the `SiteMonitorUpdater` class ❸. In the run method of this class, you perform two steps. First you call a method called `updateAllSites` ❹, which, as the name implies, performs the various steps to update the widget data. Once that operation is complete, the service calls `stopSelf`, which cleanly terminates the service ❺.

At this point our `Service` is starting and has created an instance of the private `SiteMonitorUpdater` class. Let's look at the update operations.

Listing 17.12 Iterating through each of the sites and request updates

```
    private void updateAllSites() {
      Log.i(SiteMonitorUpdater.tag,"updateAllSites");

       try {
        AppWidgetManager appWidgetManager =
AppWidgetManager.getInstance(context);
        ComponentName widgetComponentName = new
ComponentName(context,SiteMonitorWidgetImpl.class);
        int [] widgetIds =
appWidgetManager.getAppWidgetIds(widgetComponentName);
        for (int i=0 ; i< widgetIds.length; i++) {
          SiteMonitorModel smm =
SiteMonitorModel.getWidgetData(context, widgetIds[i]);
          if (smm != null) {
            updateOneSite(smm,widgetIds[i]);
          } else {
            Log.i(SiteMonitorUpdater.tag,"Ignore this zombie widget!");
          }
        }

        Intent updateWidgetsIntent = new
Intent(SiteMonitorWidgetImpl.UPDATE_WIDGETS);
        context.sendBroadcast(updateWidgetsIntent);

        Log.i(SiteMonitorUpdater.tag,"Complete!");
      } catch (Exception e) {
        Log.e(SiteMonitorUpdater.tag,"updateAlLSites::caught exception:" +
 e.getMessage());
        e.printStackTrace();
      }
    }


    private void updateOneSite(SiteMonitorModel smm,int widgetId) {
      try {
        Log.i(SiteMonitorUpdater.tag,"updateOneSite: [" + smm.getName() +
"][" + widgetId + "]");

        // get update report from this site's url
        Log.i(SiteMonitorUpdater.tag,"url is [" + smm.getUrl() + "]");
        String dataFromSite = getDataFromSite(smm.getUrl());
        String[] data = dataFromSite.split("[|]");
        if (data.length == 2) {
          smm.setStatus(data[0]);
          smm.setMessage(data[1]);
        }
        smm.setStatusDate(SiteMonitorModel.getFormattedDate());
        SiteMonitorModel.saveWidgetData(context, widgetId,  smm);

      } catch (Exception e) {
        Log.e(SiteMonitorUpdater.tag,"updateOneSite::caught exception:" +
e.getMessage());
        e.printStackTrace();

      }
    }
```

1 Get widget list

2 Load widget data

3 Update widget

4 Refresh widget UI

5 Download, parse data

6 Update widget data

7 Save widget data

```
  private String getDataFromSite(String siteUrl) {
   String ret = "BAD|unable to reach site";
   URL url;

   try {
    url = new URL(siteUrl);
    HttpURLConnection urlConn = (HttpURLConnection)
url.openConnection();
    BufferedReader inBuf = new BufferedReader(new
InputStreamReader(urlConn.getInputStream()));
    String inputLine;
    String result = "";
    int lineCount = 0; // limit the lines for the example
    while ((lineCount < 10) && ((inputLine = inBuf.readLine()) !=
null)) {
     lineCount++;
     Log.v(SiteMonitorUpdater.tag,inputLine);
     result += inputLine;
    }

    inBuf.close();
    urlConn.disconnect();

    return result;

   } catch (Exception e) {
    Log.d(SiteMonitorUpdater.tag,"Error caught: " + e.getMessage());
    e.printStackTrace();
    return ret;
   }
  }
```

Talk to web server ❽

❾ **Return web data**

The `updateAllSites` method starts like so many other methods in this chapter: by obtaining a reference to the `AppWidgetManager` and creating a list of widgets ❶. For each widget identifier, the code attempts to load the associated widget data ❷ and perform an update by calling the `updateOneSite` method ❸.

The `updateOneSite` method invokes the `getDataFromSite` method ❺. The `getDataFromSite` method performs some basic HTTP `GET` code to retrieve a string from the remote site ❽. Once the data is retrieved from the remote site, it's returned ❾ to the `updateOneSite` method.

The returned data is parsed and stored in the `SiteMonitorModel` instance ❻. The date is updated and the widget data is saved with a call to `SiteMonitorModel.saveWidgetData` ❼.

After all the sites have been updated, an `Intent` is broadcast with the action `SiteMonitorWidgetImpl.UPDATE_WIDGETS` ❹. This causes the user widget UI to update, reflecting the most recent updated information.

That concludes the review of the code for this chapter. Let's now look at AndroidManifest.xml, which has a lot of important information tucked away.

17.9 Tying it all together with AndroidManifest.xml

If you're ever experiencing problems with an Android application, particularly during development, remember to check the manifest file. Chances are that you've forgotten to define an `Activity` or omitted a request for a required permission. The following listing presents the AndroidManifest.xml used for the SiteMonitor application. Looking at this should tie together any loose ends on how an AppWidget application can be constructed.

Listing 17.13 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.msi.unlockingandroid.sitemonitor"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
android:label="@string/app_name">
      <activity android:name=".SiteMonitorConfigure"
                  android:label="@string/app_name">
        <intent-filter>
            <action
android:name="android.appwidget.action.APPWIDGET_CONFIGURE" />
        </intent-filter>
      </activity>
      <receiver android:name=".SiteMonitorWidgetImpl">
        <intent-filter>
          <action android:name=
"android.appwidget.action.APPWIDGET_UPDATE" />
        </intent-filter>
        <intent-filter>
          <action android:name=
"com.msi.unlockingandroid.sitemonitor.UPDATE_WIDGETS" />
        </intent-filter>
        <meta-data android:name="android.appwidget.provider"
android:resource="@xml/sitemonitorwidget" />
      </receiver>
      <receiver android:name=".SiteMonitorBootstrap">
        <intent-filter>
          <action
     android:name="com.msi.unlockingandroid.sitemonitor.ALARM_ACTION" />
        </intent-filter>
      </receiver>
        <service
            android:name=".SiteMonitorService"
            android:enabled="true">
        </service>

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

1 Package declaration

2 Application tag

3 SiteConfigure Activity

4 Configuration IntentFilter

5 SiteMonitorWidgetImpl receiver

6 Auto Update IntentFilter

7 AdHoc Update IntentFilter

8 AppWidget metadata

9 SiteMonitorBootstrap receiver

10 SiteMonitorService

AndroidManifest.xml contains all the ingredients required to ensure that the application components know how to link to one another at runtime. All the components within the application share the same package name ❶. The `Application` tag defines the name of the `application` ❷ label, which in this case is taken from a string resource named app_name.

The `SiteMonitorConfigure Activity` is declared ❸. Note the presence of the `IntentFilter` named `android.appwidget.action.APPWIDGET_CONFIGURE` ❹. The presence of this `IntentFilter`, along with the contents of the `AppWidget`'s metadata file ❽, serve to provide the necessary elements to support the use of an `AppWidgetProvider`. Note that this metadata file is stored in the xml folder beneath the res folder. This xml folder must be created manually, as it's not part of the stock project folders when a new Android project is created in Eclipse.

The class `SiteMonitorWidgetImpl` ❺ is defined with a `receiver` tag and additional pieces of information that are essential for this application's proper operation. This receiver contains the `android.appwidget.action.APPWIDGET_UPDATE IntentFilter` ❻ along with the custom `IntentFilter` with an action of `com.msi.unlockingandroid.sitemonitor.UPDATE_WIDGETS` ❼. This is the `Intent Action` used when the application wants to update the visual representation of the `AppWidgets`. This `receiver` tag also refers to the metadata entry that defines the attributes for an `android.appwidget.provider`.

The `SiteMonitorBootstrap` is defined ❾ along with an action of `com.msi.unlockingandroid.sitemonitor.ALARM_ACTION`. This action triggers the launching of the `SiteMonitorService` ❿. Finally, the `SiteMonitorService` can't retrieve data over the Internet without the `uses-permission` of `android.permission.INTERNET`.

17.10 Summary

In this chapter we covered `AppWidgets`; you learned not only what they are but how they can be used and what it takes to make them operate at runtime. You learned that there can be more to the `AppWidget` than meets the eye. A smart phone's home screen is where users spend most of their time, so learning how to add value to the home screen is an essential skill for Android developers.

We presented some usage scenarios for `AppWidgets` and introduced a sample application named SiteMonitor that served as the context for discussing `AppWidgets`. This application demonstrated the techniques required to manage widget instances and refresh remote data in a nontrivial application. We explained the architecture required to support SiteMonitor and then presented it in a step-by-step fashion.

The major building blocks of the sample application presented important topics such as managing widget data through the versatile `SiteMonitorModel` class. Handling widget-specific data is critical to a successful `AppWidget` application. This chapter also covered some of the "undocumented features" of Android's `AppWidgets` and you learned how to code around those features.

This chapter explored the use of many of the other skills you've learned throughout this book. The chapter covered `AppWidgets` but also integrating with `SharedPreferences`, managing alarms, `Services`, `RemoteViews`, `Threads`, and more. After building an AppWidget with moving pieces behind the scenes, you now have an appreciation of how and where AppWidgets can be deployed and how Android may be leveraged to deliver intuitive and high-value applications to your customers.

In the next chapter we look at another important feature set of Android devices: the numerous sensors that make these devices much more than a mere telephony tool.

18 Localization

This chapter covers

- The need for localization
- Strategies for localizing an application
- Dynamic localization in Java
- Obstacles to late localization

Android is a worldwide open platform gaining market share at a rapid pace. As Android reaches into new markets globally, the opportunity for mobile developers to distribute applications is reaching a level previously enjoyed by only the most successful of software products and projects. You can deploy an application across the globe, have the Google Marketplace handle the commercial transaction, and receive a royalty check. And you can accomplish all this without ever leaving your home office or dorm room!

To sell an application worldwide successfully, you must target it to a broad and diverse audience. English enjoys a broad acceptance and practice, and therefore you can expect that an English language application will sell well globally, but why give up on sales opportunities to the broader non-English-speaking Android market? In its simplest approach, this means translation into multiple languages. But there's more! Beyond language translation, an application needs to properly handle dates and times, number and currency formats, and for some applications unit of measure. This chapter is an introduction to the localization capabilities of the Android platform.

In this chapter we cover the topics required for localizing an application. You'll learn high-level strategies for localizing an application, and you'll take a hands-on approach to converting an existing application to another language. Looking back to chapter 12's Field Service application, you'll refactor that code to be localization ready and then translate it to Spanish. We demonstrate the localized version of the application through screenshots throughout the chapter.

We conclude the chapter with a discussion of challenges of localizing an application as an afterthought rather than designing for localization from the start.

We think it's only fair to note that the author of this chapter is a native English speaker and his knowledge of any languages beyond English involves computer programming: Java, C, C#, Objective-C, and so on. If you're one of the talented and fortunate among us who can speak numerous languages, please bear with the rest of us as we broaden our horizons to the global scene.

18.1 The need for localization

For many programmers there's only one language: Java. Of course the world is larger than one programming language, and it's much larger than a single spoken and written language. The majority of computer users choose a language when they set up their computer and never think twice about the decision. In fact, after the initial setup many of the resources related to other languages may even be deleted from the computer or phone, to save storage space. If this describes your experience, you're not alone! The aim of this chapter is to equip you to navigate the task of localizing an Android application and in the process reach a broader audience. As a side benefit, you'll likely find that you look at application development differently, even if you never pursue localization of your own applications.

The reasons for localizing an application are manifold. For a commercial application, there are numerous markets to reach; there's no need to limit your sales to a single marketplace. Your application may be bound for some cultural reasons to a specific region, but many applications such as games, utilities, and productivity tools are of universal interest and appeal. Games in particular are often relatively light in textual components, with the majority of the text constrained to settings screens and help files. It's to the developer's advantage to access additional markets to increase sales volume. Volume is important, considering the low price point of most mobile applications.

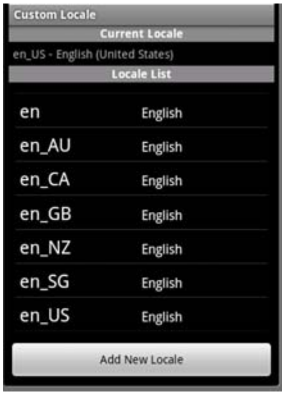
Even if your application isn't designed to generate sales revenue directly, its purpose may be to help build your brand and expand your influence. If you're part of an organization with a presence beyond your home country, localization is important to you as well.

Incredibly, the market reach of the mobile phone is greater worldwide than that of the personal computer. People even have cell phones ahead of running water in some parts of the globe. Many of these cell phones run a version of the Android OS, so as an Android developer, you can reach a large and diverse market. A small amount of revenue across numerous transactions can really add up!

Whether your purpose in localizing an application is to increase sales or to reach a new market for noncommercial reasons, you need a strategy and some practical skills, both of which you'll learn in this chapter. Before jumping into the details of localizing an application, let's examine the concept of a locale.

18.2 Exploring locales

A *locale* is generally referenced as a short character code including both the language and the geographic region. The origin and meaning of these values stem from political actions and boundaries, both of which change over time. Our discussion should be considered practical and hopefully useful rather than being a treatise on the history, meaning, and minutia of the "official" definition of locale, which is elusive.



Custom Locale

Current Locale

en_US - English (United States)

Locale List

en	English
en_AU	English
en_CA	English
en_GB	English
en_NZ	English
en_SG	English
en_US	English

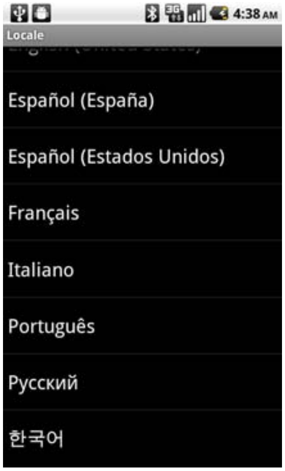
Figure 18.1 English locales on Android emulator

As language and regional barriers are blurred thanks to technology, the concept of a locale has been employed imperfectly to aid electronic communication. The ISO format for a locale identifier is a short character code of the following syntax:

```
language_REGION
```

where the language is represented first as two lowercase characters and followed by the region, which is represented as two uppercase characters. For example, the locale setting most Android phones employ in the United Kingdom is `en_GB` for "English, Great Britain." In Australia, the value is `en_AU`. `en_US` is the locale for English in the United States. Figure 18.1 shows an Android emulator instance indicating the various flavors of English locales in the Customize Locale application.

To access the available locales on an Android device, select Language & Keyboard from the Settings app, then tap on Select Language. This presents the installed locale options, as shown in figure 18.2.



Locale

Español (España)

Español (Estados Unidos)

Français

Italiano

Português

Русский

한국어

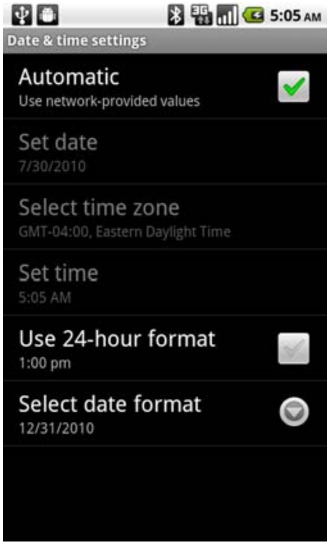
Figure 18.2 On-device locale options

When you speak of a locale, you often think of language: English, Spanish, Chinese, Polish, and so forth. Locale is more about orthography, or the rules for the written language. But the topic of localization encompasses more than just words. For example, Great Britain and the United States share a common language: English. The differences extend beyond how the language is used and how certain words are spelled. Consider the formatting of dates, which differs on either side of the Atlantic Ocean. In the United States, it's common to format a date with the month preceding the day: MM/DD/YYYY. In the UK, it's more common to see the day of the month listed first: DD/MM/YYYY.

As an Android developer, you must keep not only written language in mind, but also the formatting of dates, times, and numbers that users can customize. These differences are more than trivia; they have a direct impact on how an application is coded, how it behaves, and importantly, how it's tested prior to release. The Date & Time settings are shown in a separate preferences screen, as you can see in figure 18.3.

As the use of speech technology improves and becomes more heavily adopted, the spoken word will also be germane to the topic of localization.

Localization in practice entails more than just picking a language, so let's talk strategy on localizing an Android application.



5:05 AM

Date & time settings

Automatic ☑
Use network-provided values

Set date
7/30/2010

Select time zone
GMT-04:00, Eastern Daylight Time

Set time
5:05 AM

Use 24-hour format ☐
1:00 pm

Select date format
12/31/2010

Figure 18.3 Date and time settings

18.3 Strategies for localizing an application

Ideally an application is built from the "top down" with localization[1] in mind from day one. If you always take this approach from the start, congratulations, this topic will be a breeze for you. But if you've ever written an application that has some hard-coded strings or perhaps some code that makes specific assumptions about status codes or date formatting, you have some work to do to make your applications play nicely across multiple locales.

There are a number of perspectives on localization. We won't cover all of them, but the discussion that follows should give you a good foundation for localizing your application. Throughout this chapter, most of the code examples we use are from a localized variant of the Field Service application you met in chapter 12. The code in chapter 12 is *not* localized. The chapter 20 code is both localized and is additionally translated into Spanish. Let's get started!

18.3.1 Identifying target locales and data

In all likelihood, you'll develop your Android application in your native language and your initial target deployment may be in that same language. Your application may be aimed at a specific people group and a specific language. Whether you're targeting a

[1] Localization is the how to, and the why. Frank Ableson's post on Linux Magazine scratches the surface: http://www.linux-mag.com/id/7794.

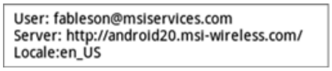
broader audience or a specific single market, it's a good idea to always keep your target market in mind.

For example, in the Field Service application, you may have users, customers, and dispatchers distributed anywhere in the world. This means that you need to keep the entire infrastructure in mind, not just the mobile application. You might congratulate yourself for making your application play nicely in the fr_FR locale settings, but if all the application data is still shown in English, you're missing a vital element of the bigger objective. So not only does your mobile code need to be localized, but the entire infrastructure needs to keep locale in focus.

Data generated on the server may require on-the-fly translation. Perhaps a simpler approach is to filter query results based on the specified language. The device's locale can be programmatically obtained at runtime using the `toString()` method of the `Locale` class. This is shown in the following listing and demonstrated in figure 18.4 and figure 18.5.

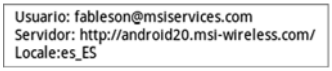
Listing 18.1 Getting current locale at runtime

```
private void RefreshUserInfo() {
 final TextView emaillabel = (TextView) findViewById(R.id.emailaddresslabel);
            emaillabel.setText(this.getString(R.string.user) +": " +
this.myprefs.getEmail() + "\n" + this.getString(R.string.server)+ ": " +
 this.myprefs.getServer() + "\n" +
   this.getString(R.string.locale) + ":" + Locale.getDefault().toString()

            );
    }
```



User: fableson@msiservices.com
Server: http://android20.msi-wireless.com/
Locale:en_US

Figure 18.4 en_US locale



Usuario: fableson@msiservices.com
Servidor: http://android20.msi-wireless.com/
Locale:es_ES

Figure 18.5 es_ES locale

You obtain a reference to a `TextView` widget for displaying textual information at runtime. The `getDefault()` static method returns the currently selected locale. The `toString()` method displays the ISO format of the locale.

When submitting a request to a server-side application, this locale value can be passed along as a query parameter:

```
http://<servername>/somepage.php?a=b&c=d&locale=en_US
```

How the server side handles the query is application-specific and beyond our discussion here. Keep in mind that localization is more than the translation of strings within your application itself. Speaking of strings, they're up next in our discussion.

18.3.2 *Identifying and managing strings*

There's perhaps no more emblematic localization topic than the concept of translating and managing an application's "strings" into target languages. Textual strings are the most visible and obvious means to target an application for a particular locale.

The centerpiece of string management within an Android application is the strings.xml file stored in the /res/values folder. The values folder contains the default resources for the application. Values for additional locales are stored in folders with names identifying the attributes for a specific language or locale. For example, figure 18.6 shows string files for both the default locale and for Spanish translation of those strings.

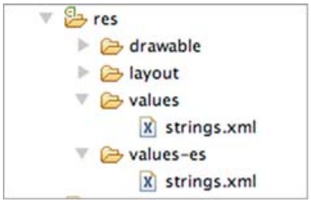


Figure 18.6
Multiple strings.xml files

A strings.xml file contains a list of strings, as shown in the following listing, which shows some of the strings used in the Field Service application.

Listing 18.2 Default strings.xml file

```
<?xml version="1.0" encoding="utf-8"?>                  ❶ XML file declaration
<resources>
    <string name="app_name">Unlocking Android</string>          ❷ String entry
    <string name="sign_and_close">Sign and Close</string>
    <string name="cancel">Cancel</string>
    <string name="refresh_job_list">Refresh Job List</string>
    <string name="manage_jobs">Manage Jobs</string>
    <string name="settings">Settings</string>
    <string name="user">User</string>
    <string name="server">Server</string>
    <string name="refreshing_job_list">Refreshing Job List</string>
    <string name="connecting">Connecting</string>
    <string name="there_are_count_jobs">There are %d jobs</string>
    <string name="jobid">Job ID</string>
    <string name="comments">Comments</string>
    <string name="product">Product</string>
    <string name="map_job_location">Map Job Location</string>
    <string name="get_product_info">Get Product Info</string>
    <string name="job_is_closed">Job is closed</string>
    <string name="view_signature">View Signature</string>
    <string name="email_address">Email Address</string>
    <string name="server_url">Server URL</string>
    <string name="save_settings">Save Settings</string>
    <string name="close_job">Close Job</string>
    <string name="locale">Locale</string>
</resources>
```

The strings.xml files are standard XML files ❶. To create one of these files, you can either create it "by hand" or you can select File > New: Android Xml File in Eclipse. Each string value ❷ has an attribute that uniquely identifies the string along with a value stored between the <string> and </string> tags.

When creating a strings.xml file for any target language beyond the default locale, you have the option of translating every string or just a subset of the strings. At runtime, your application will automatically load the correct string, looking first in the locale-specific file and working back to the default to find the required value. This cascading works in a manner similar to CSS. The most "precise" interpretation of the device locale is the first file to be searched. If the resource isn't found in this location, the platform works its way up the tree toward the default. Let's look at an example.

Let's say your application is written in a default language of English with 20 unique strings stored in /res/values/strings.xml. You anticipate that your application will be deployed around the globe, but you're specifically targeting English- and French-speaking users in the United States, France, and Canada.

To implement this strategy, your application contains four different strings.xml files, one in each of the directories listed in table 18.1.

Table 18.1 List of strings.xml files

Directory	Comment
/res/values/strings.xml	Default strings.xml stores values in English for this example
/res/values-fr/strings.xml	Complete translation of strings in French
/res/values-fr-rFR/strings.xml	A subset of strings with France-specific translations, spellings, etc.
/res/values-fr-rCA/strings.xml	A subset of strings with Canada-specific translations, spellings, etc.

When a string is looked up at runtime, Android uses the current locale as a filter to choose resources. Let's say our device is set for the French Canadian locale: fr_CA. If the string we require is found in the /res/values-fr-rCA/strings.xml file, it's used. If the string isn't found there, the file /res/values-fr/strings.xml is searched next because it's the general French strings file. If it's still not found, the string will be taken from the default strings file /res/values/strings.xml.

Not every string needs to be provided in each file. If only a handful of strings differ in Canada versus the general French strings, just provide the Canada-specific values in the fr_rCA file. Following this practice of managing a minimum number of strings can be helpful, because it can be labor- and testing-intensive to manage multiple string tables.

So far, our discussion has included only strings, language, and region-specific files. Strings aren't the only resources that may be localized. In the next section we take a brief look at other localized resources.

18.3.3 Drawables and layouts

Beyond strings, your application may need to provide locale-specific resources for *drawables* (images) and for user interface layouts. The process of managing drawables and layouts is identical to managing strings. If you require locale-specific versions of your images and layouts, they should be put in locale-specific folders in the /res folder.

Providing locale-specific images seems reasonable, because your application's images may have textual contents, or perhaps your application has images of region-specific currency. It'd make sense to show an appropriate image of a greenback dollar in the US or a Euro in most of Europe. But what about layouts: why would you want to localize a layout?

Most UI elements such as TextView widgets and Buttons display textual values. If those textual values vary in length from one language to the next—which they can and often do—it may be prudent to provide a locale-specific layout file in some instances. The idea here is to be intentional about your application's visual appearance rather than letting the user have a nondeterministic experience. Recall that many UI elements specify a width value wrap_content. This may not result in a visually appealing layout at runtime. If a particular string is going to distort the UI of your application, find out ahead of time and rearrange your widgets within a locale-specific layout as required.

In addition to your resources, you need to consider the data values your application uses, such as date and time, numbers, and currency. This is the topic of the next section.

18.3.4 Dates, times, numbers, and currencies

When working with data, keep in mind that users in various parts of the world manipulate dates and numbers differently. If you doubt that, just try to enter the thirteenth day of December 2010 into an application expecting input in the form of DD/MM/YYYY. If you enter 12/13/2010, the application will choke because there's no thirteenth month!

Manipulating these values isn't so much of an Android topic as it is a Java topic. As such, our discussion here is limited to a quick survey of commonly used Java classes for the purpose of handling data in a locale-specific fashion. Demonstrating each of these classes is beyond the scope of this chapter, and we encourage you to view the exhaustive Javadocs available for these classes. Table 18.2 enumerates some of the classes you're likely to employ when working with a localized application.

Table 18.2 Helpful classes for localized applications

Class name	Comment
java.util.GregorianCalendar	Subclass of the Calendar class, allowing for date and time manipulation specific to a locale.
java.text.SimpleDateFormat	Useful for formatting date and time according to custom developer-supplied formats.
java.text.DecimalFormat	Formats a decimal value according to a specific locale and string format.
java.text.DecimalFormatSymbols	Helper class to DecimalFormat. Use this class to retrieve currency symbols, grouping, and decimal symbols. Some locales use commas for grouping and period for decimal separation; others do the opposite. This class helps navigate those formatting distinctions.
java.util.Locale	Most of the previous classes rely on this class in one way or another.

Before examining the localized version of our Field Service application in more detail, there's one more topic to discuss that has less to do with code than it does with coding.

Most developers we know won't translate their applications to multiple languages and locales on their own—they'll employ a teammate or an outside service. Regardless of whom you work with, it's helpful to keep them in mind from the start of your project. The next section discusses things you can do to work successfully with your translation team.

18.3.5 Working with the translation team

Managing the strings within an application is straightforward on the surface, but it's not without some challenges, in part because you're working with others. The translation professionals may be part of your organization or they may be an outside service. If you have the good fortune of working closely with a teammate for translation, things may go easier for you, as you can rely on them for not only term translation but also context. An outside party can provide those services as well, but the cost in terms of dollars and time may be much greater.

Some of the challenges relate to the translation task itself, but there's another challenge with building a localized application: discipline. Unless you're working in a structured environment, some aspects of software creation are fluid. You may have an idea for an enhancement to a section of your code. You're excited about this feature, so you code the enhancement and add a menu item to enable this new aspect of your application. Terrific, your application is now more functional and your users love you! Hold on a moment. Did you use any string literals when you coded the new feature, including the menu? If so, did you get translations yet? A localized application may suffer from some latency and added expense when you factor in the translation and testing steps. When you send data out to your translation team, it's important not only to provide an exhaustive list of strings or terms that need to be translated, but also to provide as much context as possible. For example, when we had the Field Service application's terms translated to Spanish, we sent the list to our outside translation vendor. Included in that list was a brief description of each term and its use in the application. A further helpful step would be to storyboard the application with screenshots (or screencasts), thereby equipping the translation team with as much context as possible.

It's also a good idea to keep cultural references in mind. Remember: your objective is to translate user experience, not just textual terms!

At this point, you have enough information on why localization is important and some feel for what needs to be done to make it happen. The next section digs deeper into the capabilities built within the Android resource subsystem to aid in localization.

18.4 Leveraging Android resource capabilities

With the rapid pace of innovation bringing new and varied capabilities to the mobile market in general, and Android-powered devices in particular, the topic of localization for the Android platform extends beyond mere language- and number-formatting requirements. Because Android devices can vary in terms of their graphical capabilities and physical attributes, some applications will require multiple layout files and drawables targeted for a particular orientation and collection of display characteristics. In short, the mechanics of multiple resource files for supporting multiple locales can also provide Android applications with a flexibility previously unavailable in the mobile landscape.

18.4.1 More than locale

Locale is but one of a handful of attributes that can be specified within your resources. In addition to language and region (locale), the following attributes may be used as qualifiers for organizing and specifying resources to be employed at runtime:

- Mobile carrier country (mmc)
- Mobile network code (mnc)
- Orientation: portrait, landscape
- Screen size: small, normal, large
- Screen aspect: long (WQVGA, WVGA, FWVGA), notlong (QVGA, HVGA, VGA)
- Dock Mode: car, desk
- Screen pixel density: ldpi, mdpi, hdpi, nodpi (non scalable bitmaps)
- Screen type: no touch, stylus, finger
- Input method: qwerty keyboard, no keyboard
- Navigation: wheel, trackball, dpad, none (touchscreen only)
- API revision

Android is a moving target as it matures with new capabilities, so this list is regularly expanding and improving along with each Android release. For the most up-to-date version of this list, including the specific qualifiers for the resource folder definition, visit http://developer.android.com/guide/topics/resources/providing-resources.html.

At this point, you know how to make your applications provide locale- and device-specific resource files. The next section demonstrates how those resources relate to one another.

18.4.2 Assigning strings in resources

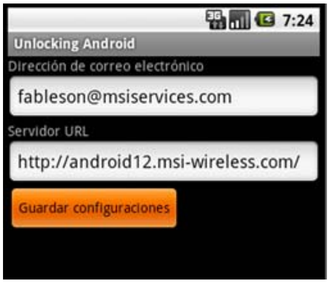
To effectively manage a localized application throughout its lifespan, you must leverage the strings.xml file to manage every string. The place where most (but not all) those strings are employed is within layout files. For example, this listing demonstrates a simple layout resource file that references string values.

Listing 18.3 Showsettings.xml, which references string constants

```
<?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"                    ❶ Linear layout
    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/email_address"                     ❷ Reference strings
    />
    <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="@string/server_url"                      ❷ Reference strings
    />
    <EditText
      android:id="@+id/serverurl"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:autoText="true"
    />
    <Button android:id="@+id/settingssave"                   ❸ Button with
      android:text="@string/save_settings"                     string reference
      android:layout_height="wrap_content"
      android:layout_width="wrap_content"
      android:enabled="true"
    />
</LinearLayout>
```

When defining a typical UI for an Android application, you define a layout ❶ containing multiple widgets. Regardless of the kind of layout you use, there will be widgets. Some of these widgets (namely `TextView` instances) will contain textual elements. In this case the `android:text` attribute is assigned a string value. For a properly localized application, this attribute will always refer to a string resource ❷ of the format *@string/<identifier>*. Likewise, the `Button` widget ❸ often displays textual strings and should also refer to a string constant.

Figure 18.7 shows a localized version of the configuration screen.



Unlocking Android

Dirección de correo electrónico

fableson@msiservices.com

Servidor URL

http://android12.msi-wireless.com/

Figure 18.7 Localized screen referencing strings directly in layout

Of course, not every string in an application is defined as an attribute of a layout resource. Some strings are provided at runtime within Java code. Fortunately, this approach too is straightforward and is presented in the next section.

18.5 Localizing in Java code

Many of the strings your application uses can be referenced in the application's layout files, but there's often a need for building a string dynamically at runtime to display to the user. These strings embedded into your code must be localized as well! In fact, our experience shows that it's these strings that flesh out the application and test your discipline as a developer committed to localization. These are also the strings that send you back to the translation team with further requests for translation services!

The first and fundamental use of localized strings in code is the simple string retrieved from the string table and directly displayed without further formatting. We'll start by looking at a snippet of the Field Service application's code prior to localization. The following listing shows the `onCreateOptionsMenu` method, which handles the creation of the presented menu options.

Listing 18.4 Menu creation code prior to localization

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    super.onCreateOptionsMenu(menu);

    menu.add(0, 0, 0, "Sign & Close");
    menu.add(0, 1, 1, "Cancel");
    return true;
}
```

❶ Menu with literal string

A literal string ❶ is used to add a menu option to the application in this prelocalization version of the code taken from chapter 12.

Let's now see how this code is converted to a localized form. The first step (see the following listing) is to ensure that we have these strings present in the string table, as shown in listing 18.2.

Listing 18.5 Subset of the string table

```
<string name="sign_and_close">Sign and Close</string>
<string name="cancel">Cancel</string>
```

With the strings defined in the string table, you can reference them from code. When using localized strings in Java code, your best friend is the `Context` class's `getString` method. This method takes a single integer argument representing the desired string, as defined by the `R` class. Recall that the `R` class is automatically generated by the Android Developer Tools whenever a resource is modified and saved within Eclipse. Consider the following code, which demonstrates using the `getString` method to retrieve localized strings.

Listing 18.6 Retrieving localized strings from the string table

```
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        super.onCreateOptionsMenu(menu);

        menu.add(0, 0, 0, this.getString(R.string.sign_and_close));
        menu.add(0, 1, 1, this.getString(R.string.cancel));
        return true;
    }
```

Retrieving a string is accomplished with a call to the `getString` method, passing in a reference to the string. The `R.string.`*`<identifier>`* value comes directly from the `name` attribute in the string table. Recall that the application-level `R` class is generated in the project as R.java. Never modify that file on your own, because it's regularly updated by the ADT and all your manual changes are lost!

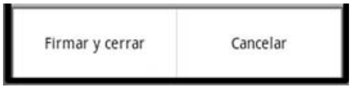


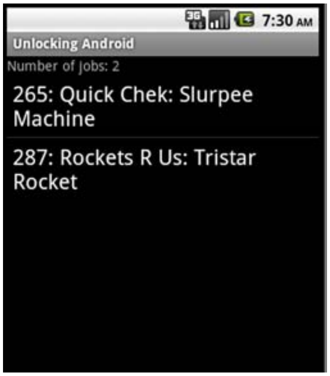
Figure 18.8 Localized menu

This localized menu is shown in figure 18.8.

Static strings are fine, but what about strings that have some formatting involved? What about when an application requires dynamic content to be localized? Let's look at using the `Formatter` class next.

18.6 Formatting localized strings

One of the screens used in the Field Service application lists all the jobs assigned to this user. Figure 18.9 shows this in the en_US locale.



Unlocking Android

Number of jobs: 2

265: Quick Chek: Slurpee Machine

287: Rockets R Us: Tristar Rocket

Figure 18.9 A job list in English

Note that the number of jobs phrase at the top of the screen is defined in the default string table as

```
<string name="there_are_count_jobs">Number of jobs: %d</string>
```

In the Spanish version of the string table, this is defined as

```
<string name="there_are_count_jobs">Hay %d trabajo(s)</string>
```

The `%d` placeholder is used to specify where the integer should be placed within the string. At runtime, this string is extracted and subsequently formatted with the help of the `java.util.Formatter` class, as shown here.

Listing 18.7 Formatting localized strings

```
if (this._joblist.getJobCount() == 0) {
    tv.setText(this.getString(R.string.there_are_no_jobs_available));
} else {
    Formatter f = new Formatter();
    tv.setText(f.format(this.getString(R.string.there_are_count_jobs),
    this._joblist.getJobCount()).toString());
}
```

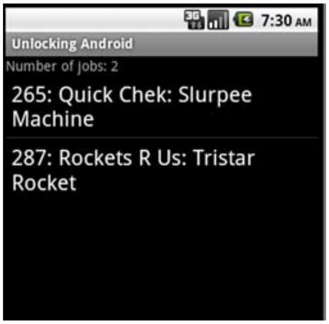
In this dynamic formatting code, the first step is to determine whether there are any jobs to display to the user. If not, a static string is retrieved and assigned to the TextView. In the case where there are jobs available for display, an instance of the Formatter class is created. The format method of the Formatter object is invoked, passing in the localized string pulled from the string table associated with the identifier of R.string.there_are_count_jobs, along with the integer value representing the number of jobs.

The benefit of using the Formatter class is that you have the option of presenting strings in different manners in different languages. Figure 18.10 shows the job listings screen in English.

Figure 18.11 shows the same screen but in Spanish. Note the different placement of the numeric value.

You've now seen examples of both statically and dynamically localized strings. There's no end to the combinations you might employ in your applications.

We conclude this chapter with a brief discussion of obstacles you should avoid when building a localized application.



7:30 AM

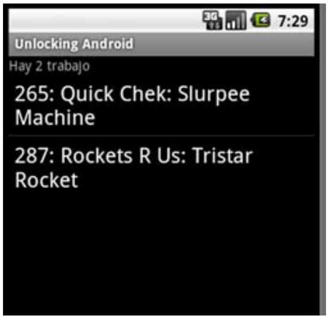
Unlocking Android

Number of jobs: 2

265: Quick Chek: Slurpee Machine

287: Rockets R Us: Tristar Rocket

Figure 18.10 Job listings screen in en_US locale



7:29

Unlocking Android

Hay 2 trabajo

265: Quick Chek: Slurpee Machine

287: Rockets R Us: Tristar Rocket

Figure 18.11 Job listings screen in es_ES locale

18.7 Obstacles to localization

Localizing an application shouldn't be an afterthought. Too much effort is required to properly rework your application when you consider all the supporting cast around your application. The Field Service application has a server-side component, and with it a whole set of other users and use cases to consider. In this final section, we examine a couple of these considerations.

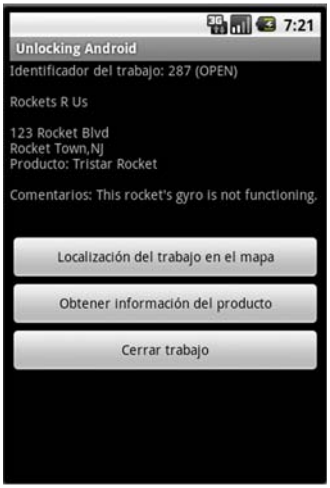
Anything that's shown to the user should ideally be put into a form that's readily consumed and is relevant to them. This means that certain elements of your coding approach might need to change—and this is often not easy to accomplish.

For example, one aspect of the Field Service application that should be treated differently is the use of the status code. The application uses the values OPEN and CLOSED transparently but without translation, as shown in figure 18.12.

This is a case where the same piece of text (OPEN, CLOSED) is used for not only display but also control of the application. In hindsight, this approach is something to avoid. The status code should be internal to the application and hidden from the user in its "raw" form. Instead, a locale-specific version should be rendered based on the underlying value. The status code ripples throughout the application, both on the device and on the server side, so modifying it involves a more comprehensive and costly effort than just translating some strings in the application.

Additionally, we discussed earlier in the chapter the idea of filtering out results that are potentially not usable for a particular user. Refer again to figure 18.12. Note that this screen is shown in Spanish but contains a "job" with English comments. Some elements of a job may be unable to be translated—for example, proper names of products or a physical address—but the comments themselves should be carefully distributed to only users who can be productive with the information.

The point in demonstrating some of the shortcomings of this (partially) localized application is to emphasize that localizing an application after it has been released is much more work than starting with localization in mind. Consequently, localizing an application down the road adds more cost than if you'd designed the application for localization from the start. In addition, the user experience may be compromised based on some of the steps required to make a localized application "fit" into an infrastructure that didn't contemplate the possibility beforehand.



Identificador del trabajo: 287 (OPEN)

Rockets R Us

123 Rocket Blvd
Rocket Town,NJ
Producto: Tristar Rocket

Comentarios: This rocket's gyro is not functioning.

Localización del trabajo en el mapa
Obtener información del producto
Cerrar trabajo

Figure 18.12 Speedbumps in localization: OPEN status

18.8 Summary

In this chapter we explored the topic of localizing an Android application. We reviewed high-level concepts, including motivation, strategy, and technique. The code used a variant of chapter 12's Field Service application as an example to illustrate both the techniques and challenges of localizing an application.

We looked at the resource structure and the services performed for you automatically by the Android platform. Beyond the definition and organization of the resources, we examined the various means of working with localized strings.

Although you may not have learned much Spanish in this chapter, we trust that you're now ready to add localization to your list of capabilities and in the process make your applications available to a much broader audience.

In addition to the basic mechanics of localizing an application, a key idea to take from this chapter is that localization isn't a casual exercise to be undertaken at some point in the future, but rather it should be designed into the DNA of your application from the start.

In the next and final chapter, you get to go under the hood of Android as we look at the Android Native Development Kit (NDK) and write C code for Android.

19

Android Native Development Kit

This chapter covers

- Introducing the Android Native Development Kit
- Exploring the Java Native Interface
- Building an application with the NDK
- Integrating NDK into Eclipse

The majority of the code in this book is written to employ the Android SDK using the Java programming language. Looking back to chapter 13, we explored creating native executable applications for Android by writing Linux-compatible applications. The code in that chapter was written in the C language, but it didn't produce applications that are easily executed on consumer hardware. The design approach in chapter 13 requires an unlocked developer, or rooted, device and is arguably only applicable for developers who are building custom Android builds—it's not for the typical developer looking to deploy applications to consumer-based handsets. This chapter presents the "approved" manner of writing C code for the Android platform with assistance from the *Android Native Developer Kit*, or simply the *NDK*.

This chapter presents the NDK as an aid to Android developers. The architecture of the NDK is presented and discussed in the context of a nontrivial, hands-on image-processing application. Image processing is a broad field encompassing many applications across almost every industry. Perhaps the most familiar example of an image-processing application is *optical character recognition (OCR)*. OCR can be implemented via a number of different algorithms. Many image-processing algorithms begin by attempting to identify the target object within a still image or frame of a video. One classic technique for separating an object from the background is known as *edge detection*. An edge detection algorithm analyzes the image looking for the outlines of any objects within the image, be they characters or any other object. The application that accompanies this chapter, named UA2E_FindEdges, implements a classic image-processing algorithm known as *Sobel Edge Detection*.[1] Using UA2E_FindEdges, you'll use the Android camera to acquire a photo and then find all of the edges within the image. The algorithm to be used has been ported from another platform in the C programming language, compiled into a native code library, and employed by an Android SDK Java application.

The chapter wraps up with a demonstration of integrating the NDK into the Eclipse build environment, permitting a nearly seamless experience for the developer.

19.1 Introducing the NDK

NDK is a bolt-on, complementary tool chain to the core Android SDK that permits developers to create application functionality in the C programming language. The NDK isn't meant to replace the SDK applications, but rather is designed to augment them. In fact, you can't create a standalone application with the NDK. The NDK compiles code written in C into libraries that are callable by SDK-based Java code. The role of these native libraries is to provide additional functionality to the Java application. The NDK also handles all of the application packaging steps to make sure that the resulting APK file contains not only the Java code but the native code libraries as well.

Considering the power and breadth of the Android SDK, why would you bother with the NDK?

19.1.1 Uses for the NDK

For most developers, there's no reason to use the NDK, as the combination of the Java programming language and the Android SDK are more than capable to meet the needs of their applications, and they'll never need to drop down to the "native" level to accomplish their tasks. But there are scenarios where C is better suited to the task than Java. Consider the case where an application needs to perform a large number of bitwise operations on 8-bit data elements for applications such as raw signal conditioning, image processing, or encryption. Operations such as these are ideally suited for the C language with its efficient use of memory and its agility when working with raw data. Likewise, accessing the OpenGL libraries from C may provide better performance for some applications where the developer is skilled in C.

[1] Here is an introductory tutorial covering the topic of Sobel Edge Detection: http://www.generation5.org/content/2002/im01.asp.

In addition to graphics programming and raw data processing needs where tight native code has an advantage, there's another scenario where the NDK may be worth considering. Taking into account the large body of code written for Linux in the C programming language, there may be functionality that a developer can "drop in" without the need to port the functionality to Java. The NDK enables the reuse of legacy C code.[2]

Interestingly, code written with the NDK isn't guaranteed to execute faster than Java code that implements the same algorithm, so the NDK shouldn't be viewed as an automatic choice when reviewing options to improve application performance. All things being equal, the NDK actually complicates matters, so it should be deployed with thoughtful consideration. The NDK is a good fit for self-contained and CPU-intensive operations where memory allocation is kept to a minimum.

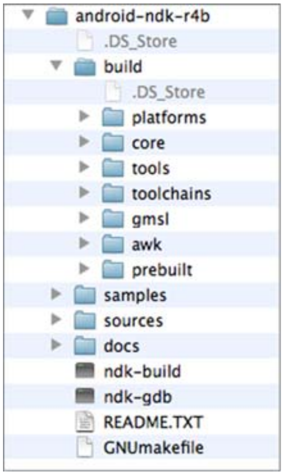
To build an NDK project, you need to understand the components of the NDK and how it relates to an Android application.

19.1.2 Looking at the NDK

The NDK is a freely available download from the Android developer website at http://developer.android.com/sdk/ndk. Like the Android SDK, there are versions available for each of the supported platforms: Windows, Mac, and Linux. Installing the NDK is as simple as downloading the archive file and unzipping it into an accessible place on your development machine. Placing the NDK in a folder parallel to the SDK is a good idea.

Creating a symbolic (or soft) link to the directory is also helpful. For example, you might put a link on your laptop, making the NDK easy to access:

```
ln -s /users/fableson/Software/android/android-ndk-r4b/ ndk
```



```
▼ android-ndk-r4b
    .DS_Store
  ▼ build
      .DS_Store
    ▶ platforms
    ▶ core
    ▶ tools
    ▶ toolchains
    ▶ gmsl
    ▶ awk
    ▶ prebuilt
  ▶ samples
  ▶ sources
  ▶ docs
    ndk-build
    ndk-gdb
    README.TXT
    GNUmakefile
```

Figure 19.1 NDK on the disk

The NDK is updated periodically just like the SDK, so using a soft link such as this can aid in managing build scripts, as you'll see at the end of this chapter.

At first glance, the NDK footprint seems pretty straightforward, but under the build folder lies a maze of make files. Figure 19.1 shows the NDK as it resides on the hard drive.

Fortunately for us, building a native library from C source code is surprisingly simple. Table 19.1 lists the high-level steps in building a native library. The balance of the chapter walks through the process of building a native library and a sample application that leverages the functionality of that library.

The code generated by the NDK is known as a *Java Native Interface (JNI) library*. A JNI library[3] exposes one or

[2] Start here to explore more about legacy C code: http://www.imagix.com/links/c_cplusplus_language.html.

[3] To access the JNI documentation: http://download.oracle.com/javase/6/docs/technotes/guides/jni/index.html.

Table 19.1 Build steps for an NDK library

Step	Comment
Create Android project	The starting point is to have an Android project to work with.
Create library source folder	Use the name jni (short for Java Native Interface). This folder contains C source plus project-specific make files. It should be at the same level as the src folder within a standard Android project folder structure.
Create C source file	This file contains the implementation of the native library. This code may be split among multiple C source files.
Create Android.mk	This is the configuration (or make) file for the native library. An example is provided later in this chapter.
Change directory to library source folder	The NDK must be run from within your jni folder.
Execute NDK	Type `ndk-build` from the command line to execute the build script. This processes a series of make files that perform all of the compilation and linking of the library. The result is a file ready for inclusion in the APK file.
Optional step: integrate NDK into project's build	Ideally modifying and saving the C source file will result in the complete build of the application, including both SDK and NDK aspects. Taking the time to do this makes the development process much more appealing.
Look for any compilation or linking errors	Any coding or configuration errors will become apparent as lines written to the standard output and standard error of the console where the script was executed. If run from within Eclipse, the output is shown in the console window.

more functions to a Java application through "exported" functions. The names of the exported functions follow a strict naming convention. Failure to comply with this naming convention results in runtime errors.

The resulting name of the JNI library is lib<*name of library*>.so, which is the naming convention for a shared library in the Linux environment. The JNI specification defines the interface between the Java and C environments. To illustrate the JNI interface with the NDK, let's build an image-processing application from start to finish.

19.2 Building an application with the NDK

This section presents a step-by-step guide to building an application that leverages the Android NDK. Before jumping into the code, let's walk through the high-level functionality of the application, which is named UA2EFindingEdges, including screenshots showing the application in action. After the demo of the application, we examine each of the pieces of code to construct the application from the ground up.

19.2.1 Demonstrating the completed application

The sole function of this application is to convert a photograph into a grayscale image, showing the edges of the object within the photograph. The application is written in Java using the Android SDK with a minimalist interface, as shown in figure 19.2.

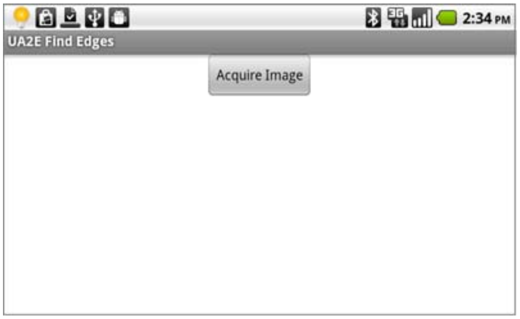


Figure 19.2 Application waiting to take photo

Selecting the Acquire Image button launches the built-in Camera application with default image settings. Take a photo. Figure 19.3 shows an image taken of a model race car body.

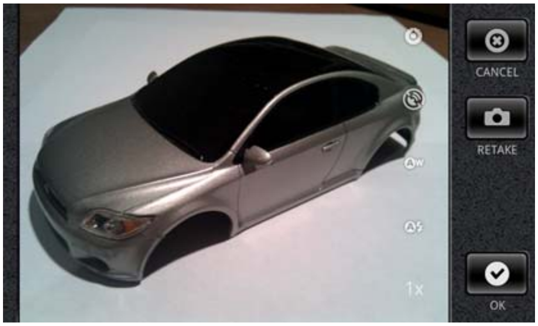


Figure 19.3 Take a photograph.

Hitting the OK button in the Camera application brings the photo back to our sample application and displays the image. The Find Edges button is now available, as shown in figure 19.4.

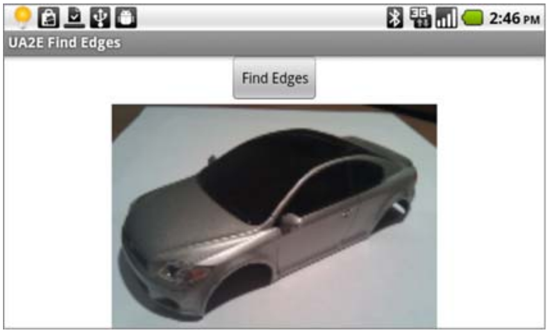


Figure 19.4 Captured image before image processing

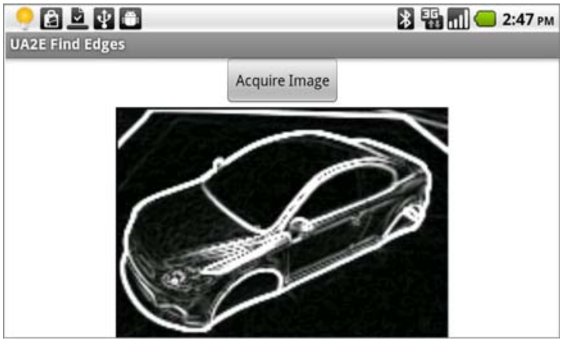


Figure 19.5 **Showing the edges of the car**

It's now time to exercise the primary function of this application: the edge-detection routine. See figure 19.5.

After the Find Edges button is selected, the application performs two consecutive image-processing routines, each of which is implemented in the C language JNI library. The first function converts the color image to grayscale, which is a common technique in image-processing algorithms. After the image has been converted, a transformation known as the *Sobel Edge Detection* algorithm is performed to highlight the edges in the photograph. Once the image processing is complete and the image updated, the application is ready to acquire a new image.

The image-processing prowess of this application is hardly groundbreaking, but the application is fun to play with and presents a sufficiently complex problem to solve with the NDK. You're encouraged to follow along in the next section and build this application for yourself. If you'd like to just use the application, it's available for download in the Android Market.

19.2.2 Examining the project structure

The application consists of two primary parts. The first is the Android SDK-based Java code, which contains the application structure, the UI, the click handlers, code to display the images, and all of the usual AndroidManifest.xml goodies required to make the application run on an Android device. The second portion of the application is the image-processing library built with the NDK. The library contains the two image-processing functions, written in C and exported for use by the user interface code. Figure 19.6 shows the project as it looks in Eclipse.

This project looks like every other Android project you've worked with to date, with the addition of the jni and libs folders.

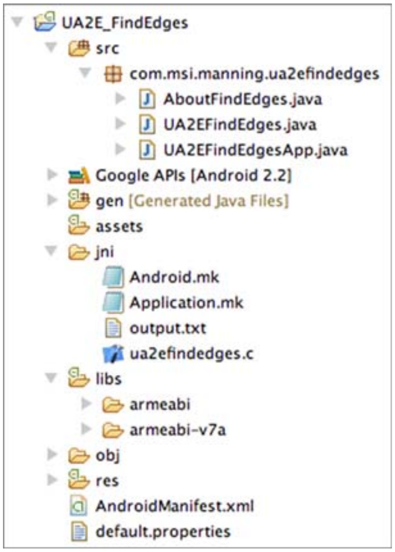


Figure 19.6
A project in the Eclipse GUI

The jni folder contains three files of interest: the C language file and two make files. The output.txt file is created by the NDK build subsystem.

The libs folder contains output files from the NDK build process targeted for different CPU architectures. The topic of which CPU target to use is beyond our objectives in this chapter—you can learn more about processor-specific settings in the readme files in the NDK's docs folder.

Let's start with a look at the JNI code.

19.3 Building the JNI library

Building a JNI library requires a basic understanding of JNI, a C source file, and the appropriate entries in the Android.mk file. The next few sections break down these requirements, step by step. We begin with a brief discussion of how code is mapped between the Java and C language environments.

19.3.1 Understanding JNI

As mentioned earlier, a JNI library exposes one or more functions to a Java-based application through a specific naming convention. A C language function is named according to the following guideline:

```
Java_fully_qualified_class_name_method_name
```

For example, a method named `SomeMethod` in the class named `DoSomething`, which takes an integer and a string argument, would be defined as shown in the following listing.

Listing 19.1 Sample JNI function signature

```
JNIEXPORT jint JNICALL Java_com_somecompany_DoSomething_SomeMethod(
JNIEnv * env,
jobject  obj,
jint i,
jstring s);
```

The function is named according to the JNI standard, including the prefix of `Java`, followed by the fully qualified class name and the method name. Every exported JNI function has at minimum the same first two arguments. The first argument is a pointer to the Java Environment. See the jni.h header file shipped with the NDK for available functionality in the `JNIEnv` object. The next argument is a pointer to the `this` object. Any additional arguments follow these two standard arguments. In this case, there's an integer argument, which is a data type of `jint`, and a `String` argument of type `jstring`. Again, see the jni.h header file for a more complete view of the available data types.

Calling a JNI function from Java first requires that the library be loaded. This is accomplished with a call to `System.loadLibrary`, passing in the module name. The module name is the name of the shared library minus the lib prefix and the .so extension. The next listing shows how to load this module and declare the sample method.

Listing 19.2 Calling a JNI function

```
package com.somecompany;

public class SomeObject {

    public SomeObject {

        native int SomeMethod(jint i,jstring s);
        static {
            System.loadLibrary("SomeModule");
        }
    }
}
```

In this simple example, note the package name and class name. These names combine to form a portion of the JNI function name. The JNI function named `SomeMethod` is defined with a native qualifier. To gain access to this function, it must be loaded via a call to `loadLibrary`, passing in the module name of the library.

Much more is involved in the JNI specification, but you have enough here to get started on the sample application code.

19.3.2 *Implementing the library*

Finally, you get to look at the C code that implements the image-processing functions! The code listings are broken into three logical sections: the header of the C file with macros and data type definitions, and then each of the two image-processing functions. First you see the header of the file ua2efindedges.c.

Listing 19.3 ua2efindedges.c

```
#include <jni.h>                              ◁ ❶ jni header file
#include <android/log.h>                                            ❷ Log, bitmap headers
#include <android/bitmap.h>

#define  LOG_TAG    "libua2efindedges"                                   ❸ Logging
#define  LOGI(...)  __android_log_print(ANDROID_LOG_INFO,                  macros
    LOG_TAG,__VA_ARGS__)
#define  LOGE(...)  __android_log_print(ANDROID_LOG_ERROR,
    LOG_TAG,__VA_ARGS__)

typedef struct                                    ◁  Structure for
{                                                 ❹  image handling
    uint8_t alpha;
    uint8_t red;
    uint8_t green;
    uint8_t blue;
} argb;
```

The first header file included in the C source file is jni.h ❶. This file contains the required data types and macros for the JNI. Without this header file, data types would be unrecognized and the code would never compile. The Android NDK provides support for a handful of Android subsystems, including the logging and bitmap handling, among others. Those headers are included as well ❷ because they're required for this application. A few macros ❸ aid in accessing the LogCat functionality. Because this application is dealing with image data where each pixel is defined as a 32-bit structure representing a Color object, a structure is defined to easily manage the pixel data ❹.

NOTE The bitmap functionality shown in this example requires Android 2.2 or later.

Let's now discuss the image-processing routines. Don't concern yourself with the details of these functions unless they're of interest to you. The basic approach of this application is to pass Bitmap objects from the Java code to the JNI code. The pixel buffers are locked such that the memory is accessible to the C code for raw manipulation. When the image processing is complete, the pixels are unlocked.

The first function is named converttogray and takes two arguments. The first argument is an Android Bitmap of the style RGBA_8888, which means each pixel contains a byte representing the Alpha channel and then one byte each for Red, Green, and Blue values. Each value is represented by an integer ranging in value from 0 to 255. The second Bitmap is created as a grayscale, 8 bits per pixel image. The first parameter is the input image and the second is the output image. The following listing contains the converttogray method. Note the long function name!

Listing 19.4 converttogray function implementation

```
JNIEXPORT void JNICALL
Java_com_msi_manning_ua2efindedges_UA2EFindEdges_
➥converttogray(
```

❶ Specify function name

```
JNIEnv * env, jobject  obj,
jobject bitmapcolor,
jobject bitmapgray)
{
    AndroidBitmapInfo  infocolor;
    void*              pixelscolor;
    AndroidBitmapInfo  infogray;
    void*              pixelsgray;
    int                ret;
    int                y;
    int                x;
```

❷ Define AndroidBitmapInfo structure

❸ Contain pointer to pixels

```
    if ((ret = AndroidBitmap_getInfo(env,
    bitmapcolor, &infocolor)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }
```

❹ Get Bitmap info

```
    if ((ret = AndroidBitmap_getInfo(env,
    bitmapgray, &infogray)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }
```

❹ Get Bitmap info

```
    LOGI("color image :: width is %d; height is %d; stride is %d; format is
%d;flags is %d",infocolor.width,infocolor.height,infocolor.stride,
 infocolor.format,infocolor.flags);
  if (infocolor.format != ANDROID_BITMAP_FORMAT_RGBA_8888) {
    LOGE("Bitmap format is not RGBA_8888 !");
    return;                                       Check Bitmap format ❺
  }

  LOGI("gray image :: width is %d; height is %d; stride is %d; format is
%d;flags is %d",infogray.width,infogray.height,infogray.stride,
 infogray.format,infogray.flags);
  if (infogray.format != ANDROID_BITMAP_FORMAT_A_8) {
    LOGE("Bitmap format is not A_8 !");
    return;
  }

  if ((ret = AndroidBitmap_lockPixels(env,
  bitmapcolor, &pixelscolor)) < 0) {
    LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
  }                                               Lock Bitmap pixels ❻

  if ((ret = AndroidBitmap_lockPixels(env, bitmapgray,
  &pixelsgray)) < 0) {
    LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
  }

  // modify pixels with image processing algorithm
  for (y=0;y<infocolor.height;y++) {
   argb * line = (argb *) pixelscolor;            ❼ Access image data
   uint8_t * grayline = (uint8_t *) pixelsgray;
   for (x=0;x<infocolor.width;x++) {
    grayline[x] = 0.3 * line[x].red + 0.59 *
    line[x].green + 0.11*line[x].blue;
   }

   pixelscolor = (char *)pixelscolor + infocolor.stride;   ❽ Advance through
   pixelsgray = (char *) pixelsgray + infogray.stride;       pixel buffer
  }

  AndroidBitmap_unlockPixels(env, bitmapcolor);   ❾ Unlock pixels
  AndroidBitmap_unlockPixels(env, bitmapgray);
}
```

The simple name of `converttogray` is expanded to a much longer JNI name ❶. Arguments to the function include a color `Bitmap`, which is used as the input image, and a gray `Bitmap`, which is the resulting (or output) `Bitmap` for this function. The `AndroidBitmapInfo` structure ❷ holds information about a `Bitmap`, which is obtained with a call to `AndroidBitmap_getInfo` ❹. Details of the `Bitmap` are logged to LogCat with the previously introduced macros. Local variables are used to gain access to the pixel data ❸ and loop through the rows and columns of pixels with a couple of aptly named variables, `x` and `y`. If the bitmaps are in the expected format ❺, the function proceeds to lock down the pixel buffers ❻.

At this point, the function can confidently navigate through a contiguous memory block to access the pixels of the image ❼. This is important because most image-processing routines rely on this sort of direct memory access through pointers. This means easier inclusion of image-processing code from sources such as existing Linux code bases. The color bitmap is accessed row by row. Each color pixel is converted to a gray pixel. Pointer arithmetic ❽ aids in the navigation through the pixel memory buffer. When the images have been completely processed, the pixels are unlocked ❾.

When the converttogray function is complete, the calling Java code now has a grayscale version of the color image. The Java code to call this C code is shown later in this chapter; first let's look at the routine that detects the edges, shown in the following listing. Only the new features are discussed, as there's a great deal of similarity between the converttogray and detectedges routines.

Listing 19.5 detectedges routine

```
JNIEXPORT void JNICALL
Java_com_msi_manning_ua2efindedges_UA2EFindEdges_detectedges(
JNIEnv * env, jobject  obj,
jobject bitmapgray,
jobject bitmapedges)
{
    AndroidBitmapInfo  infogray;
    void*              pixelsgray;
    AndroidBitmapInfo  infoedges;
    void*              pixelsedge;
    int                ret;
    int                y;
    int                x;
    int                sumX,sumY,sum;
    int                i,j;
    int                Gx[3][3];
    int                Gy[3][3];
    uint8_t            *graydata;
    uint8_t            *edgedata;

    Gx[0][0] = -1;Gx[0][1] = 0;Gx[0][2] = 1;
    Gx[1][0] = -2;Gx[1][1] = 0;Gx[1][2] = 2;
    Gx[2][0] = -1;Gx[2][1] = 0;Gx[2][2] = 1;

    Gy[0][0] = 1;Gy[0][1] = 2;Gy[0][2] = 1;
    Gy[1][0] = 0;Gy[1][1] = 0;Gy[1][2] = 0;
    Gy[2][0] = -1;Gy[2][1] = -2;Gy[2][2] = -1;

    LOGI("detectedges in JNI code");

    if ((ret = AndroidBitmap_getInfo(env, bitmapgray, &infogray)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }

    if ((ret = AndroidBitmap_getInfo(env, bitmapedges, &infoedges)) < 0) {
        LOGE("AndroidBitmap_getInfo() failed ! error=%d", ret);
        return;
    }

    LOGI("gray image :: width is %d; height is %d; stride is %d; format is
%d;flags is %d",infogray.width,infogray.height,infogray.stride,
  infogray.format,infogray.flags);
    if (infogray.format != ANDROID_BITMAP_FORMAT_A_8) {
```

❶ **Input grayscale Bitmap**

❷ **Output edges Bitmap**

❸ **Setup masks**

❹ **Point to pixel data**

❺ **Set up transformations**

```
        LOGE("Bitmap format is not A_8 !");
        return;
    }

    LOGI("color image :: width is %d; height is %d; stride is %d; format is
%d;flags is %d",infoedges.width,infoedges.height,infoedges.stride,
  infoedges.format,infoedges.flags);
    if (infoedges.format != ANDROID_BITMAP_FORMAT_A_8) {
        LOGE("Bitmap format is not A_8 !");
        return;
    }

    if ((ret = AndroidBitmap_lockPixels(env,
    bitmapgray, &pixelsgray)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }

    if ((ret = AndroidBitmap_lockPixels(env,
    bitmapedges, &pixelsedge)) < 0) {
        LOGE("AndroidBitmap_lockPixels() failed ! error=%d", ret);
    }


    // modify pixels with image processing algorithm
    graydata = (uint8_t *) pixelsgray;
    edgedata = (uint8_t *) pixelsedge;

    for (y=0;y<=infogray.height - 1;y++) {
      for (x=0;x<infogray.width -1;x++) {
        sumX = 0;
        sumY = 0;
        // check boundaries
        if (y==0 || y == infogray.height-1) {
          sum = 0;
        } else if (x == 0 || x == infogray.width -1) {
          sum = 0;
        } else {
          // calc X gradient
          for (i=-1;i<=1;i++) {
            for (j=-1;j<=1;j++) {
              sumX += (int) ( (*(graydata + x + i +
              (y + j) * infogray.stride)) * Gx[i+1][j+1]);
            }
          }
          // calc Y gradient
          for (i=-1;i<=1;i++) {
            for (j=-1;j<=1;j++) {
              sumY += (int) ( (*(graydata + x + i +
              (y + j) * infogray.stride)) * Gy[i+1][j+1]);
            }
          }
          sum = abs(sumX) + abs(sumY);
        }
        if (sum>255) sum = 255;
        if (sum<0) sum = 0;

        *(edgedata + x + y*infogray.width) = 255 - (uint8_t) sum;
      }
```

6 Access pixels

7 Ignore border pixels

8 Calculate X dimension

9 Calculate Y dimension

10 Constrain pixel values

11 Calculate edge value

```
    }
    AndroidBitmap_unlockPixels(env, bitmapgray);
    AndroidBitmap_unlockPixels(env, bitmapedges);
}
```

Like the prior function, this one takes two Bitmap arguments, both of which are grayscale images. The first contains the grayscale image ❶ created in the converttogray method. The second bitmap argument ❷ becomes the "edges only" version of the image. A number of variables ❸ are defined to aid in the edge-detection process. Like the prior function, we have local pointers ❹ to the image data. The Sobel Edge Detection algorithm involves a mathematical operation known as a *convolution*, which requires initializing a pair of convolution masks ❺. The convolution must be performed across the entire image ❻ with the exception of the border pixels, which are skipped ❼. First the calculations are made in the x dimension ❽ and then in the y dimension ❾. Once the calculations have been performed for a particular pixel, a new value is calculated based on the surrounding pixels and stored in the output image ❿. Prior to storing a value, the new pixel value is constrained to be between 0 and 255 ⓫.

You're now ready to compile this code, but before you can do that you need to define the make file for the library.

19.3.3 Compiling the JNI library

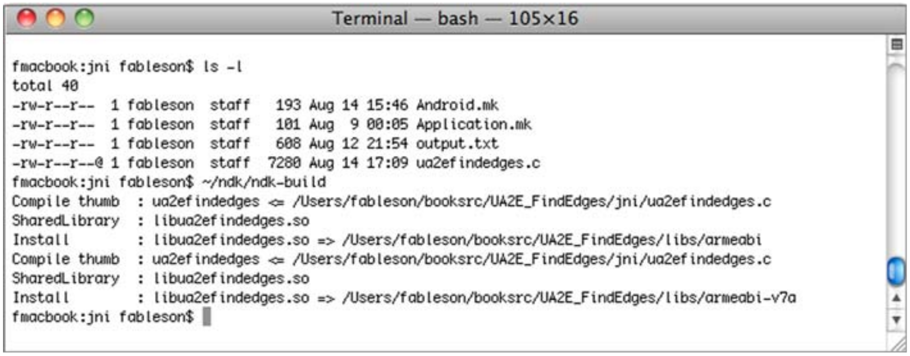
In addition to the C source file, you require a make file to instruct the NDK on how the library is compiled. The following listing contains the make file used with this library.

Listing 19.6 Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := ua2efindedges
LOCAL_SRC_FILES := ua2efindedges.c
LOCAL_LDLIBS    := -llog -ljnigraphics

include $(BUILD_SHARED_LIBRARY)
```

The Android.mk file is a make file containing build information about the JNI library for this project. The name of the library module is ua2efindedges. The actual resulting filename is libua2efindedges.so, which follows the normal file-naming structure for a dynamically loadable library for Linux. The only source file for this library is ua2efindedges.c. The input libraries are also listed. These libraries are searched for code symbols when the library is linked. The standard C and math libraries are automatically searched, so they don't need to be included in the LOCAL_LDLIBS variable.

Building the application is as simple as opening a terminal window to the jni folder and running the file ndk-build, which can be found in the NDK installation directory. Figure 19.7 shows the build process from the command line.



```
fmacbook:jni fableson$ ls -l
total 40
-rw-r--r--  1 fableson  staff   193 Aug 14 15:46 Android.mk
-rw-r--r--  1 fableson  staff   101 Aug  9 00:05 Application.mk
-rw-r--r--  1 fableson  staff   608 Aug 12 21:54 output.txt
-rw-r--r--@ 1 fableson  staff  7280 Aug 14 17:09 ua2efindedges.c
fmacbook:jni fableson$ ~/ndk/ndk-build
Compile thumb  : ua2efindedges <= /Users/fableson/booksrc/UA2E_FindEdges/jni/ua2efindedges.c
SharedLibrary  : libua2efindedges.so
Install        : libua2efindedges.so => /Users/fableson/booksrc/UA2E_FindEdges/libs/armeabi
Compile thumb  : ua2efindedges <= /Users/fableson/booksrc/UA2E_FindEdges/jni/ua2efindedges.c
SharedLibrary  : libua2efindedges.so
Install        : libua2efindedges.so => /Users/fableson/booksrc/UA2E_FindEdges/libs/armeabi-v7a
fmacbook:jni fableson$ 
```

Figure 19.7 Building the JNI library

Now that the JNI library is complete, let's swing back to the Android SDK and build the user interface for this application.

19.4 Building the user interface

The UI for the application is modest. The things the UI must do include responding to two different buttons, one for taking a picture and one for calling the JNI functions. Beyond that, the code performs simple operations to display the various bitmap images. Let's start by looking at the layout for this application.

19.4.1 User interface layout

The layout for this application is contained in the resource file main.xml, shown next.

Listing 19.7 Layout file

```
<?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"
    android:background="#ffffffff">
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:background="#ffffffff"
    android:gravity="center">
<Button android:id="@+id/AcquireImage"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Acquire Image"
    />
<Button android:id="@+id/FindEdges"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Find Edges"
    />
```

1 Outer layout

2 Layout containing Buttons

3 Acquire, Find Edges Buttons

```
</LinearLayout>
<ImageView android:id="@+id/PictureFrame"
    android:layout_width="320px"
    android:layout_height="240px"
    android:scaleType="centerCrop"
    android:layout_gravity="center_vertical|center_horizontal"/>
</LinearLayout>
```

❹ Image View

This layout is straightforward. It contains a vertically oriented `LinearLayout` ❶, which contains all the UI elements of this application. Next is a horizontally oriented `Layout`, which is also centered ❷. This layout contains two `Buttons` ❸, one for the acquisition of a photo and one for calling the image-processing routines. When an image is available, it's shown in an `ImageView` instance ❹. The visibility of the `FindEdges` `Button` is toggled on only after a photo is available.

This application relies on an `Application` object to hold a global variable—in this case, a `Bitmap`. This is necessary because a photo application often results in the user changing the orientation of the device: portrait to landscape or landscape to portrait, and so on. Whenever this occurs, Android's default behavior is to restart the `Activity`. If you store a captured photo in an `Activity`-level variable, you'll lose it each time the device is rotated. To solve this problem, store the `Bitmap` in an `Application` object. The following listing shows the code for this simple class.

Listing 19.8 `UA2EFindEdgesApp.java`

```
package com.msi.manning.ua2efindedges;

import android.app.Application;
import android.graphics.Bitmap;

public class UA2EFindEdgesApp extends Application {
    private Bitmap b;

    public Bitmap getBitmap() {
      return b;
    }
    public void setBitmap(Bitmap b) {
      this.b = b;

    }
}
```

❶ Required imports

❷ Getter and setter routines

The `Application` and `Bitmap` classes must be imported ❶ for this code to compile. The `UA2EFindEdgesApp` class extends the `Application` class. The `Bitmap` is stored as a private member, and of course you have a getter and setter ❷ to manipulate this `Bitmap`.

> **NOTE** Whenever you use an `Application` class, it must be defined in the AndroidManifest.xml file as the `android:name` attribute of the `application` tag.

Let's now look at the primary user interface code to see how you take a photo and store it into the `Application` object.

19.4.2 *Taking a photo*

There are a number of ways to take a photograph on the Android platform. For this application you'll just ask the `Camera` to do the work for you through the use of an `Intent`. The next listing demonstrates this approach. Note that the JNI-related code introduced next is employed in listing 19.10.

Listing 19.9 `UA2EFindEdges.java`

```
package com.msi.manning.ua2efindedges;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.ImageView;
import android.widget.Button;
import android.view.View;
import android.content.Intent;
import android.graphics.Bitmap;
import android.graphics.Bitmap.Config;

public class UA2EFindEdges extends Activity {

    protected ImageView imageView = null;
    private final String tag = "UA2EFindEdges";
    private Button btnAcquire;
    private Button btnFindEdges;
    // declare native methods
    public native int converttogray(Bitmap bitmapcolor,
    Bitmap gray);                                              1 Declare
    public native int detectedges(Bitmap bitmapgray,             native
    Bitmap bitmapedges);                                         methods

    static {
      System.loadLibrary("ua2efindedges");                  Load JNI
    }                                                     2 library

    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      btnAcquire = (Button) this.findViewById(R.id.AcquireImage);
      btnAcquire.setOnClickListener(new View.OnClickListener(){
        public void onClick(View v){
          try {
            Intent action = new
               Intent("android.media.action.IMAGE_CAPTURE");     3 Request
            startActivityForResult(action,1);                      photo
          } catch (Exception e) {
            Log.e(tag,"Error occurred [" + e.getMessage() + "]");
          }
        }
      });

      btnFindEdges = (Button) this.findViewById(R.id.FindEdges);
      btnFindEdges.setOnClickListener(new View.OnClickListener(){
```

```
    // code shown in next listing
    });

   imageView = (ImageView) this.findViewById(R.id.PictureFrame);
   UA2EFindEdgesApp app = (UA2EFindEdgesApp) getApplication();     ❹
   Bitmap b = app.getBitmap();
   if (b != null) {
    imageView.setImageBitmap(b);                    ❺ Display Bitmap
   }
   else {
    btnFindEdges.setVisibility(View.GONE);
   }

  }

  protected void onActivityResult(int requestCode,         ❻ Activity result
  int resultCode,Intent data)
  {
   try {
    if (requestCode == 1) {
     if (resultCode == RESULT_OK) {
      UA2EFindEdgesApp app = (UA2EFindEdgesApp) getApplication();
      Bitmap b = app.getBitmap();
      if (b != null) {                             ❼ Recycle Bitmap
       b.recycle();
      }

      b = (Bitmap) data.getExtras().get("data");      ❽ Extract, store,
      app.setBitmap(b);                                 display Bitmap
      if (b != null) {
       imageView.setImageBitmap(b);                   ❾ Toggle Button
        btnFindEdges.setVisibility(View.VISIBLE);       visibility
      }
     }
    }
   } catch (Exception e) {
    Log.e(tag,"onActivityResult Error [" + e.getMessage() + "]");
   }

  }
}
```

This code is the primary Activity for the application and is also the code that calls the previously written native code. To call the native methods, they must be declared ❶ and the JNI library must be loaded ❷ at runtime. When the Acquire button is selected, you request a photo by creating an Intent and dispatching it with a call to startActivityForResult ❸. Whenever the Activity is created, you need to check whether a Bitmap is available; to do that you first get a reference to the Application object ❹, casting it to the UA2EFindEdgesApp class. If you have a valid Bitmap, you display it ❺ in the ImageView. This approach handles the scenario where the Android device has changed orientations and the Activity has been restarted. When the Camera has captured a photo, it's packaged up into an Intent and sent back to your application via the onActivityResult method ❻. If an existing Bitmap is found in the Application object, it's recycled ❼ and the new one is extracted from the Intent ❽.

The Bitmap is stored in the Application object and displayed to the screen. Once the photo has been acquired, the Find Edges button is shown (9).

Now that you have the photo, you want to run your image-processing routines against it to find the edges. That's up next as we examine the click handler for the Find Edges button.

19.4.3 Finding the edges

The Java side of the image-processing code relies on some knowledge of how Android Bitmaps are constructed. There are three Bitmaps in play by the time the edge detection is complete:

1. The original photo is stored in the Application object as a full-color Bitmap with a format of AARRGGBB, meaning Alpha channel, Red, Green, and Blue pixel data.
2. A grayscale image is created from the color image.
3. A second grayscale image is created, which receives the edges found in the prior grayscale image.

The following listing shows the click handler and steps through the process of converting a color image to an edges-only image.

Listing 19.10 Finding the edges

```
btnFindEdges = (Button) this.findViewById(R.id.ModifyImage);
btnFindEdges.setOnClickListener(new View.OnClickListener(){
    public void onClick(View v){
      try {
        UA2EFindEdgesApp app = (UA2EFindEdgesApp) getApplication();
        Bitmap b = app.getBitmap();
        int width = b.getWidth();
        int height = b.getHeight();
        Bitmap bg = Bitmap.createBitmap(width, height, Config.ALPHA_8);
        Bitmap be = Bitmap.createBitmap(width, height, Config.ALPHA_8);
        converttogray(b,bg);
        detectedges(bg,be);
        app.setBitmap(be);
        imageView.setImageBitmap(be);
        btnFindEdges.setVisibility(View.GONE);
      } catch (Exception e) {
        Log.e(tag,"Error occured [" + e.getMessage() + "]");
      }
    }
});
```

Get Bitmap from Application (1)

Create two grayscale Bitmaps (2)

(3) Find edges

This code first obtains a reference to the Application object to retrieve the color Bitmap (1). To create the two grayscale Bitmaps (2) required for the image processing, you first need to get the dimensions of the original photo. With all of the Bitmaps ready for use, convert the color photo to a grayscale version with a call to converttogray. Next, find the edges in the image with a call to detectedges (3). Store the new image in the application and display it on the screen. You don't want to run the edge-detection routine without first obtaining a new color image, so hide the Find Edges button.

Congratulations—you now know how to use the NDK! All in all, it wasn't painful, but there's one annoying task. Building the JNI library from a command line is simple but not fun, particularly when you're accustomed to saving a Java source file and having the application auto-build. Fortunately there's a way to incorporate the NDK into the Eclipse environment.

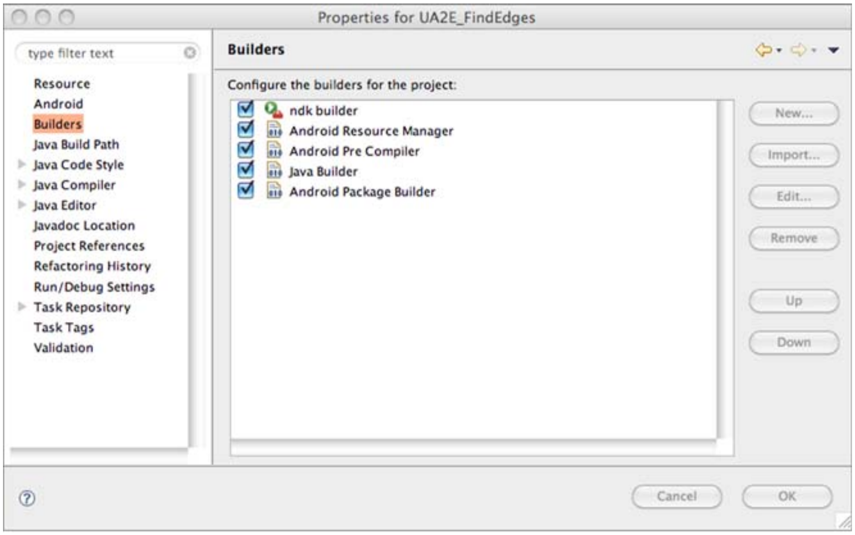
19.5 Integrating the NDK into Eclipse

The goal is simple: build the entire project from within Eclipse and never resort to the command line to perform any build operations. You can accomplish this goal by modifying the project's build properties. Highlight the project in Eclipse and select Properties to open the properties dialog. Select the Builders group, as shown in figure 19.8.

Note the ndk builder entry to the right. To create this in a new project, click the New button on the far right side of the dialog. This opens a dialog with four tabbed sections. You're interested in the first, second, and fourth tabs, as shown in the next few figures. Before looking at those screens, note that you want the NDK to build your code before the other project build steps; that way, you can get the latest version of the JNI library packaged into the final APK file. To do this, highlight the ndk builder tool you just created and select the Up button to position it at the top of the list. Don't neglect this step!

Figure 19.9 shows the options for the external tool to launch, which in this case is the fully qualified path to the ndk-build script in the NDK directory. Note the use of the symbolic link `ndk`, which eases the transition to new versions of the NDK over time.

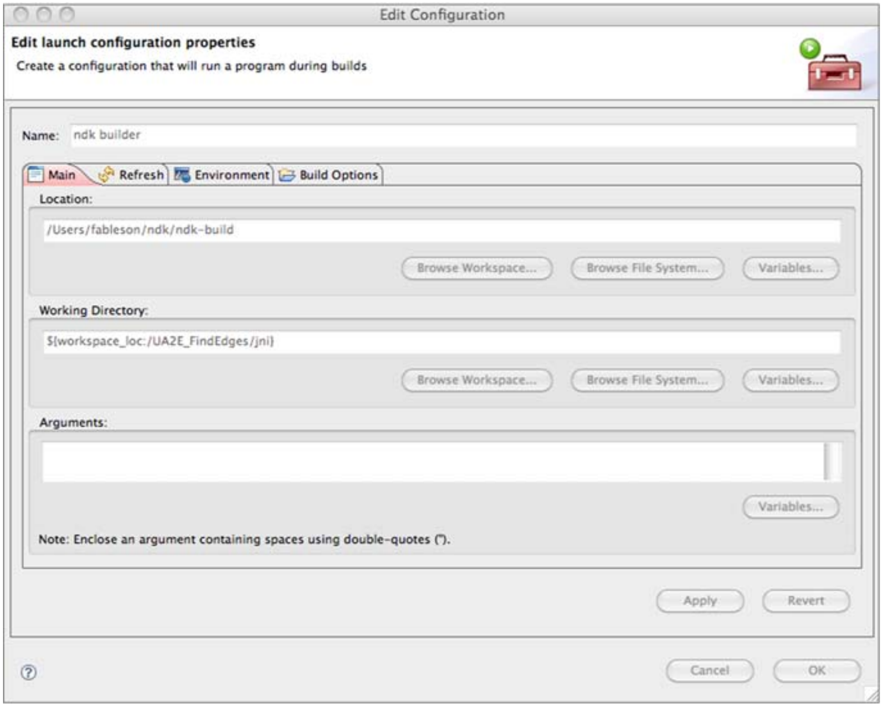
Figure 19.10 shows the next tab of the tool configuration window, which requests that the project resources be refreshed after the build this step.



Configure the builders for the project:

- ☑ ndk builder
- ☑ Android Resource Manager
- ☑ Android Pre Compiler
- ☑ Java Builder
- ☑ Android Package Builder

Figure 19.8 **Project properties**



Edit launch configuration properties

Create a configuration that will run a program during builds

Name: ndk builder

Location:

/Users/fableson/ndk/ndk-build

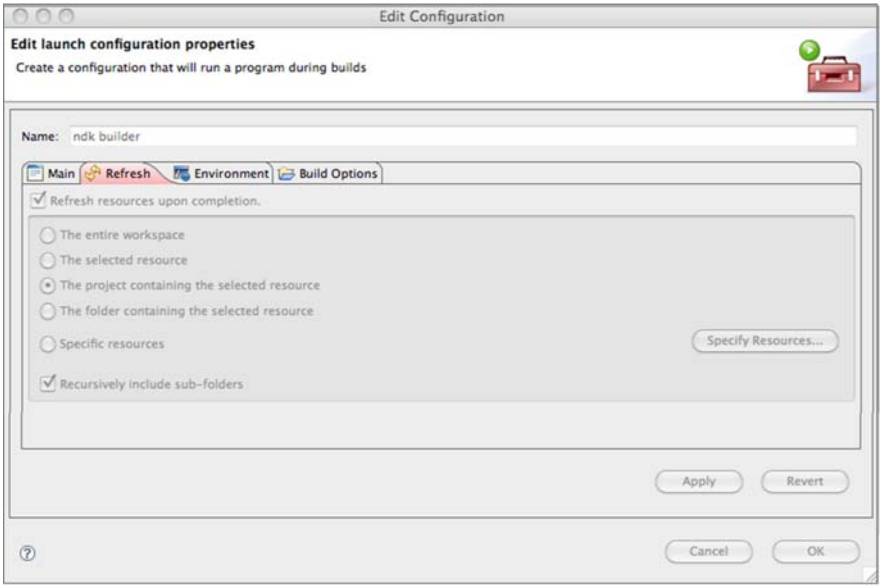
Working Directory:

${workspace_loc:/UA2E_FindEdges/jni}

Arguments:

Note: Enclose an argument containing spaces using double-quotes (").

Figure 19.9 External tool properties



Edit launch configuration properties

Create a configuration that will run a program during builds

Name: ndk builder

☑ Refresh resources upon completion.

○ The entire workspace

○ The selected resource

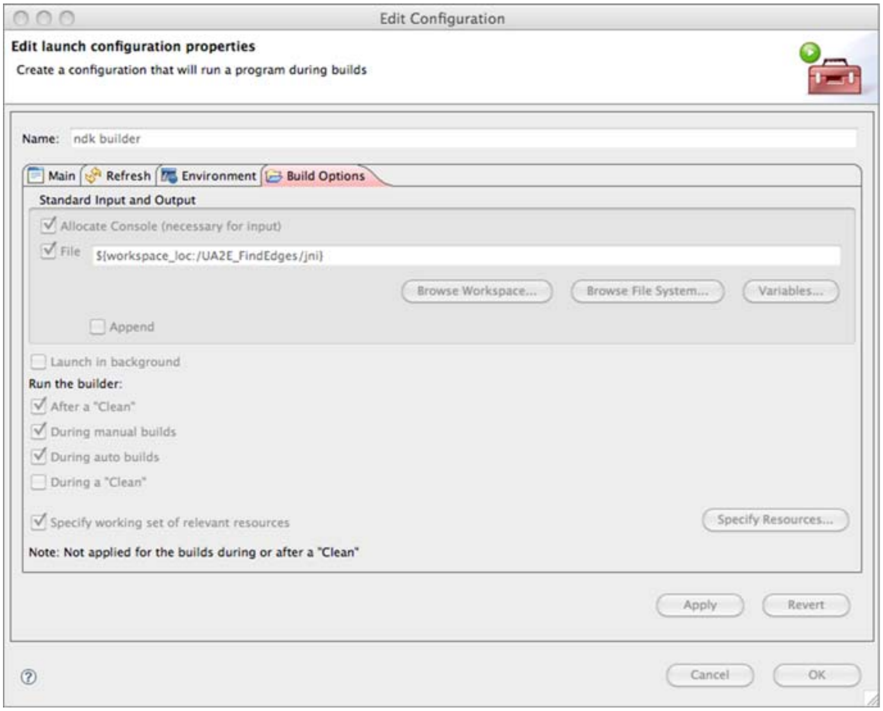
● The project containing the selected resource

○ The folder containing the selected resource

○ Specific resources

☑ Recursively include sub-folders

Figure 19.10 Request refresh after build



Edit Configuration

Edit launch configuration properties

Create a configuration that will run a program during builds

Name: ndk builder

Main | Refresh | Environment | Build Options

Standard Input and Output

☑ Allocate Console (necessary for input)

☑ File ${workspace_loc:/UA2E_FindEdges/jni}

Browse Workspace... | Browse File System... | Variables...

□ Append

□ Launch in background

Run the builder:

☑ After a "Clean"

☑ During manual builds

☑ During auto builds

□ During a "Clean"

☑ Specify working set of relevant resources — Specify Resources...

Note: Not applied for the builds during or after a "Clean"

Apply | Revert

Cancel | OK

Figure 19.11 **Configuring build options**

The fourth tab of the configuration, shown in figure 19.11, requests that the external tool be configured for auto-build. This means that when you save a dependent file of the tool (in this case, the Android.mk file or the ua2efindedges.c file), the NDK tool runs, and if there are no errors, the rest of the build process continues on to package the application for you. Goal accomplished: you stay within Eclipse! If you're a command-line kind of programmer, just disregard this step.

19.6 Summary

In this chapter we explored incorporating C language code via the NDK in the context of an image-processing application. The NDK is the means by which the JNI is accessible for Android developers. The JNI permits the Android developer to leverage the C environment for two primary purposes: to take advantage of C's speed capabilities for conducting time-critical operations to leverage existing code libraries written in C.

The chapter's sample application demonstrated a classic image-processing algorithm called edge detection with a fully functional application that demonstrated not only the C language elements required by the NDK toolset, but also the necessary Java code changes required to employ the custom JNI library.

The chapter wrapped up by adding the NDK build process into the overall Android project build process. This makes building an NDK application as streamlined as possible by setting up the NDK as an external tool to take advantage of Eclipse's auto-build feature.

appendix A *Installing the Android SDK*

This chapter covers

- Meeting development environment requirements
- Obtaining and installing Eclipse
- Obtaining the Android SDK
- Using the SDK and AVD Manager
- Configuring the Android Development Tools for Eclipse

This appendix walks through the installation of Eclipse, the Android SDK, and the ADT plug-in for Eclipse. This appendix is meant to be a reference resource to assist in setting up the environment for Android application development. The topic of using the development tools is covered in chapter 2.

A.1 Development environment requirements

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's a good idea to stay in tune with the latest developments from the Android development team at Google. You'll find the latest information regarding supported platforms and requirements for Android Development Tools (ADT) at http://developer.android.com/sdk/requirements.html.

Compatible development environments for the sample applications in this book include:

- Windows XP/Vista/Windows 7, Mac OS X 10.4.8 or later (Intel x86 only), Linux
- Eclipse 3.4 (or later), including the JDT and Web Tools Platform, which are included in the Eclipse installation package
- JDK and Java Runtime Environment (JRE) version 5 or 6
- ADT plug-in for Eclipse

Once you've identified a compatible computing environment, it's time to obtain and install the development tools. We'll start with the Eclipse IDE.

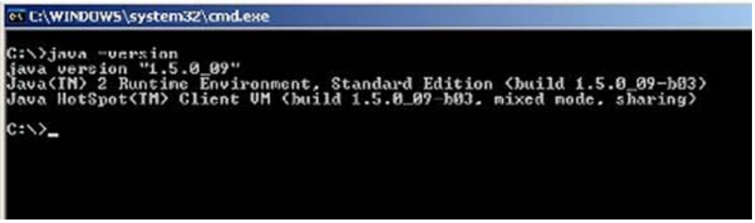
A.2 Obtaining and installing Eclipse

A requirement for running the Eclipse IDE is the JRE version 5 or later. For assistance in determining the best JRE for your development computer, go to http://www.eclipse.org/downloads/moreinfo/jre.php. It's likely that you already have an acceptable JRE installed on your computer. An easy way to determine what version (if any) you have is to run the following command from a command window or terminal session on your development computer:

```
java -version
```

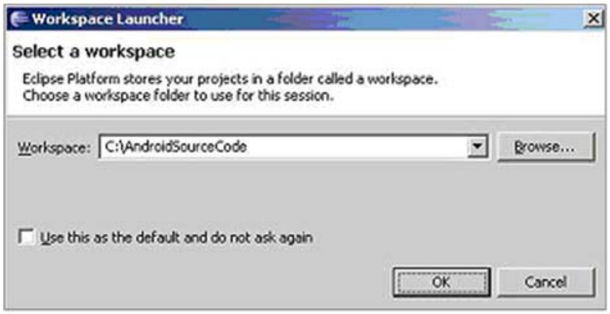
This command checks to see if the JRE is installed and present in your computer's search path. If the command comes back with an error stating an invalid or unrecognized command, that probably means the JRE isn't installed and/or it's not properly configured. Figure A.1 demonstrates using this command to check the version of the installed JRE on a computer running the Windows OS.

Once your JRE is installed, the next step is to install the Eclipse IDE. Download the latest stable release from http://www.eclipse.org/downloads. You'll 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. Once you've downloaded it, extract the contents of the file to a convenient place on your computer. Because this download is simply a compressed file and not an installer, it doesn't create any icons or shortcuts on your computer.



```
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



Workspace Launcher

Select a workspace

Eclipse Platform stores your projects in a folder called a workspace.
Choose a workspace folder to use for this session.

Workspace: C:\AndroidSourceCode

□ Use this as the default and do not ask again

Figure A.2 **Eclipse projects are stored in a workspace, which is a directory on your computer's hard drive.**

To start Eclipse, execute the file named eclipse (run eclipse.exe for Windows users) found in the directory to which you downloaded Eclipse. You may want to make your own menu or desktop shortcut to eclipse(.exe) for convenience. Executing this file loads the Eclipse IDE. Eclipse prompts for a workspace and suggests a default location, such as C:\documents and settings\username\workspace. You can change the workspace location to something Android specific to separate your Android work from other projects, as shown in figure A.2.

Accept the suggested workspace location or specify an alternative workspace location, as desired. Once Eclipse is loaded, click the Workbench: Go to the Workbench icon on the main screen, as shown in figure A.3.

Eclipse consists of many "perspectives," and the default is the Java Perspective, from which Android application development takes place. The Java Perspective is

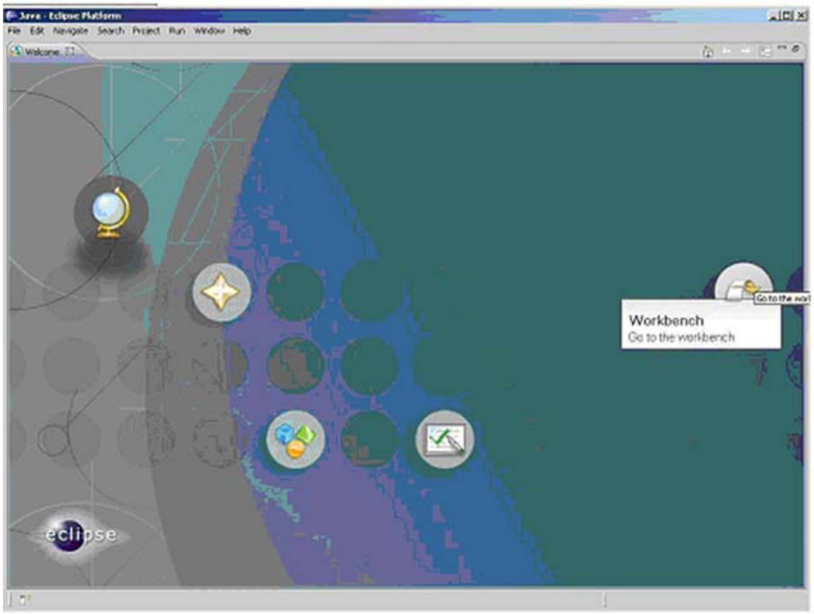
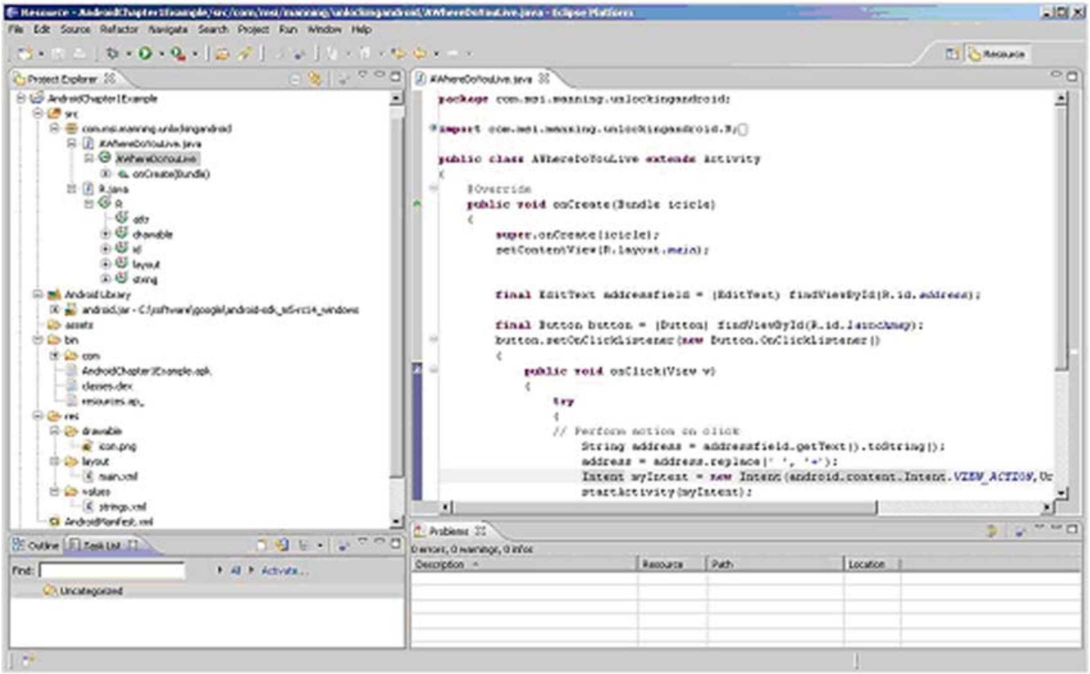


Figure A.3 **Eclipse defaults to the home screen. Go to the workbench.**



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

import com.msi.manning.unlockingandroid.R;

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


        final EditText addressfield = (EditText) findViewById(R.id.address);

        final Button button = (Button) findViewById(R.id.launchmap);
        button.setOnClickListener(new Button.OnClickListener()
        {
            public void onClick(View v)
            {
                try
                {
                // Perform action on click
                    String address = addressfield.getText().toString();
                    address = address.replace(' ', '+');
                    Intent myIntent = new Intent(android.content.Intent.VIEW_ACTION,Ur
                    startActivity(myIntent);
```

0 errors, 0 warnings, 0 infos

Description	Resource	Path	Location

Figure A.4 Android development takes place in the Java Perspective.

shown in figure A.4. Chapter 2 discusses in greater detail the use of the Eclipse IDE for Android application development.

For more information on becoming familiar with the Eclipse environment, visit http://www.eclipse.org, where you can find online tutorials for building Java applications with Eclipse.

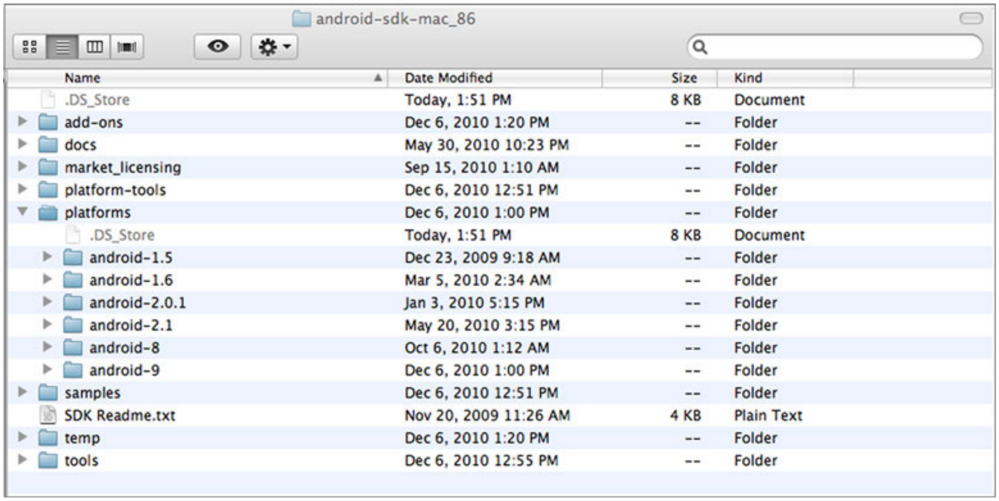
Now that Eclipse is installed, it's time to focus on the Android SDK.

A.3 Obtaining and installing the Android SDK

The Android SDK is available as a free download from a link on the Android home page, at http://developer.android.com/sdk/index.html. SDK installation versions are 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.

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 on a Windows machine, or /somefolder/android-sdk-mac_86, as shown in Figure A.5.

As you can see in figure A.5, the installation footprint is rather simple. Earlier versions of the Android SDK were a complete archive of tools, documentation, and classes. Starting with version 1.6 of the SDK, the archive only contains the tools—essentially a set of SDK Management tools. As specific packages are installed over time



Name	Date Modified	Size	Kind
.DS_Store	Today, 1:51 PM	8 KB	Document
add-ons	Dec 6, 2010 1:20 PM	--	Folder
docs	May 30, 2010 10:23 PM	--	Folder
market_licensing	Sep 15, 2010 1:10 AM	--	Folder
platform-tools	Dec 6, 2010 12:51 PM	--	Folder
platforms	Dec 6, 2010 1:00 PM	--	Folder
.DS_Store	Today, 1:51 PM	8 KB	Document
android-1.5	Dec 23, 2009 9:18 AM	--	Folder
android-1.6	Mar 5, 2010 2:34 AM	--	Folder
android-2.0.1	Jan 3, 2010 5:15 PM	--	Folder
android-2.1	May 20, 2010 3:15 PM	--	Folder
android-8	Oct 6, 2010 1:12 AM	--	Folder
android-9	Dec 6, 2010 1:00 PM	--	Folder
samples	Dec 6, 2010 12:51 PM	--	Folder
SDK Readme.txt	Nov 20, 2009 11:26 AM	4 KB	Plain Text
temp	Dec 6, 2010 1:20 PM	--	Folder
tools	Dec 6, 2010 12:55 PM	--	Folder

Figure A.5 **Unzip the Android SDK archive to your hard drive.**

they are added under the platforms folder. For Windows users, run the file named SDK Setup.exe. For other development environments, run the shell script named android. This will load the Android SDK and AVD Manager, which permits you to manage the SDKs on your development machine as well as define instances of the Android emulator.

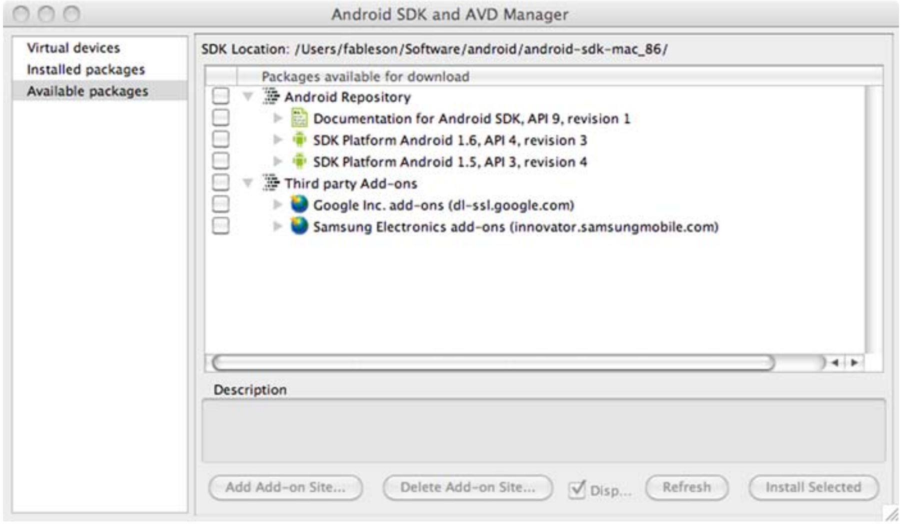
A.4 Using the SDK and AVD Manager

To begin developing Android applications, you must first download at least one of the available Android "platforms." The benefit of this approach is that it allows you to manage and use multiple SDK versions in parallel on your development machine. Given the pace at which the Android team is releasing code, this is a welcome improvement over earlier versions of the SDK.

In figure A.6 you can see the SDK and AVD Manager displaying the available Android packages for download. Only the packages which are not presently on your computer are listed here. As new packages become available are released they appear in this list.

Although the Android platform is generally described as versions, such as 1.1, 1.5, 1.6, and 2.0 and so on, the underlying technologies have been described as *levels*. For example, Android API Level 5 was introduced at the 2.0 release and Android 2.3 is API level 9. When looking at certain API documentation or working with Android platforms, you'll see these level indicators.

Unless you have a specific older Android device in mind for an application, you'll want to focus on the most recent SDK release level. Much of this book uses SDK version 2.2, but you can install multiple versions of the SDK. Figure A.7 shows an installation with support for numerous Android releases.



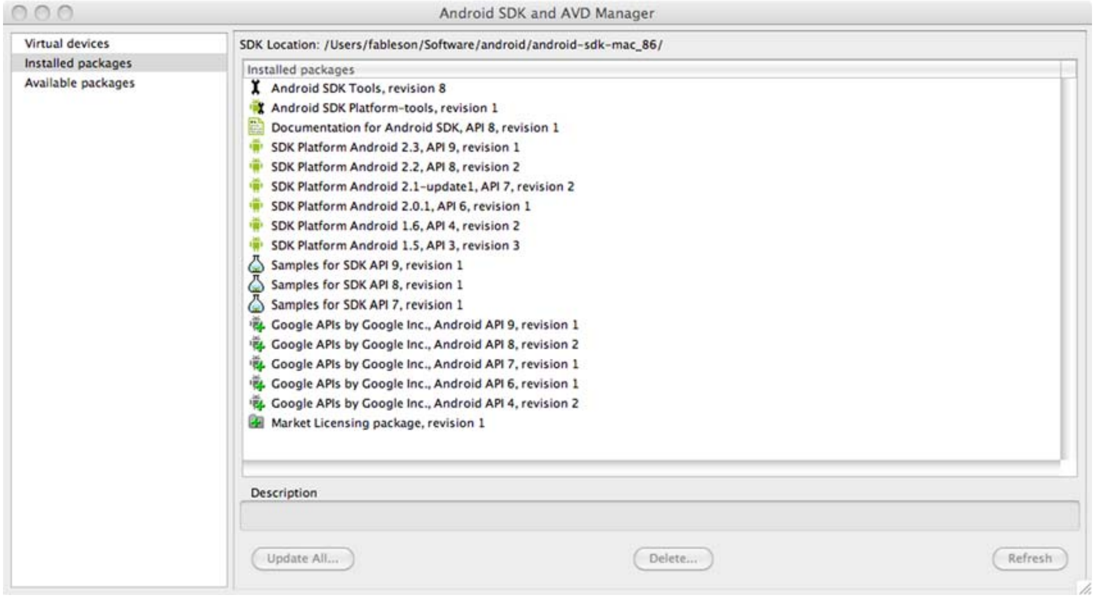
SDK Location: /Users/fableson/Software/android/android-sdk-mac_86/

Packages available for download

- □ Android Repository
  - □ Documentation for Android SDK, API 9, revision 1
  - □ SDK Platform Android 1.6, API 4, revision 3
  - □ SDK Platform Android 1.5, API 3, revision 4
- □ Third party Add-ons
  - □ Google Inc. add-ons (dl-ssl.google.com)
  - □ Samsung Electronics add-ons (innovator.samsungmobile.com)

Figure A.6 **The packages available for download**

Testing is often done via the Android emulator; each instance of the emulator is known as an Android virtual device (AVD). Here's where the AVD Manager comes into play. Under the virtual devices section of the SDK and AVD Manager, you can define instances of the emulator, each with specific characteristics, such as SDK version/API

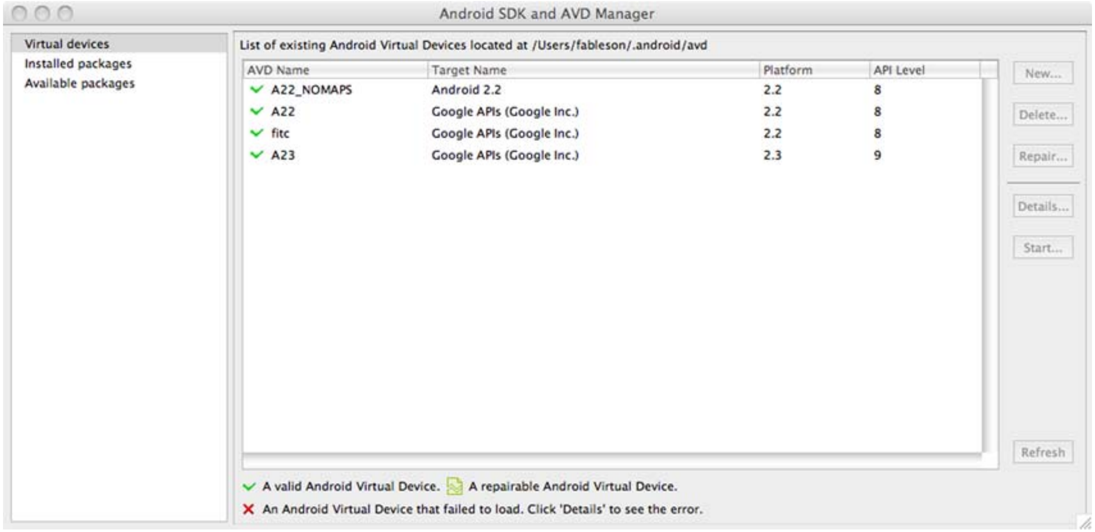


SDK Location: /Users/fableson/Software/android/android-sdk-mac_86/

Installed packages

- Android SDK Tools, revision 8
- Android SDK Platform-tools, revision 1
- Documentation for Android SDK, API 8, revision 1
- SDK Platform Android 2.3, API 9, revision 1
- SDK Platform Android 2.2, API 8, revision 2
- SDK Platform Android 2.1-update1, API 7, revision 2
- SDK Platform Android 2.0.1, API 6, revision 1
- SDK Platform Android 1.6, API 4, revision 2
- SDK Platform Android 1.5, API 3, revision 3
- Samples for SDK API 9, revision 1
- Samples for SDK API 8, revision 1
- Samples for SDK API 7, revision 1
- Google APIs by Google Inc., Android API 9, revision 1
- Google APIs by Google Inc., Android API 8, revision 2
- Google APIs by Google Inc., Android API 7, revision 1
- Google APIs by Google Inc., Android API 6, revision 1
- Google APIs by Google Inc., Android API 4, revision 2
- Market Licensing package, revision 1

Figure A.7 **The currently installed packages**



List of existing Android Virtual Devices located at /Users/fableson/.android/avd

AVD Name	Target Name	Platform	API Level
✓ A22_NOMAPS	Android 2.2	2.2	8
✓ A22	Google APIs (Google Inc.)	2.2	8
✓ fitc	Google APIs (Google Inc.)	2.2	8
✓ A23	Google APIs (Google Inc.)	2.3	9

✓ A valid Android Virtual Device. A repairable Android Virtual Device.

✗ An Android Virtual Device that failed to load. Click 'Details' to see the error.

Figure A.8 Defining an Android virtual device, also known as an emulator

Level, SDCard storage size, and screen size. Figure A.8 shows the definition of a single Android virtual device/emulator named 201 that uses the SDK version 2.0.1. You can name your virtual device anything you like.

You can view API documentation based on your local SDK installation if installed, or online at http://developer.android.com/reference/classes.html. To view documentation locally, select the index.html file under the docs folder in the folder where the Android SDK was unzipped. The SDK's documentation is largely a collection of Javadocs enumerating the packages and classes of the SDK. 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 folder contains Android-specific resource compilers and the very helpful adb tool. These tools are explained and demonstrated in chapter 2 of this book. Each version of the Android platform contains its own set of samples, tools, and runtime libraries.

> **TIP** The SDK changes from time to time as the Android team releases new versions. If you need to upgrade from one version to another, there will be an upgrade document on the Android website—be sure to examine the relevant upgrade documentation file to learn of important changes to the API. Check for items that may impact your previously written applications.

Both Eclipse and the Android SDK are now installed. It's time to install the ADT plug-in for Eclipse to take advantage of the ADT's powerful features, which help you bring your Android applications to life.

A.5 Obtaining and installing the Eclipse plug-in

The following steps show you how to install the Android plug-in for Eclipse, known as the ADT. The most up-to-date installation directions are available from the Android website at http://developer.android.com/sdk/eclipse-adt.html. The first steps are generic for any Eclipse plug-in installation, not just the ADT.

Here are the basic steps to install the ADT:

1 Run the Find and Install feature in Eclipse, found under the Help > Software Updates menu, as shown in figure A.9.

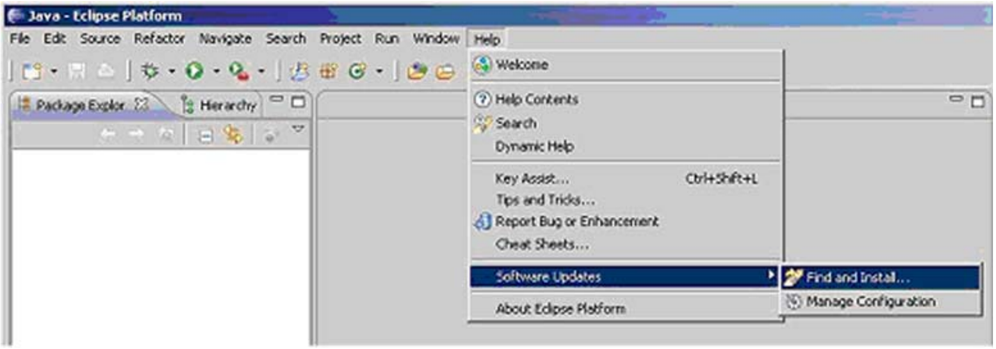
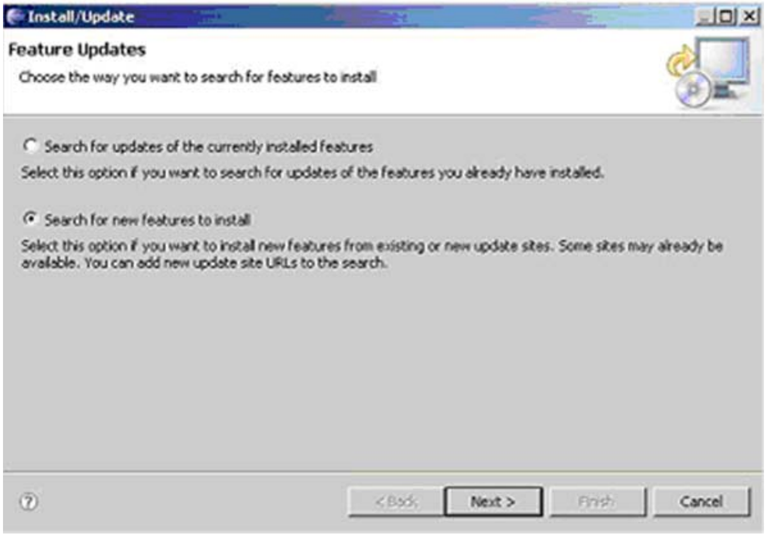


Figure A.9 The Eclipse environment supports an extensible plug-in architecture.

2 Select the "Search for new features to install" option, as shown in figure A.10. Click Next.


Install/Update

Feature Updates

Choose the way you want to search for features to install

○ Search for updates of the currently installed features

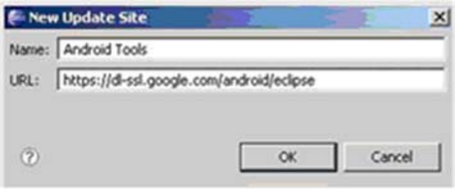
Select this option if you want to search for updates of the features you already have installed.

● Search for new features to install

Select this option if you want to install new features from existing or new update sites. Some sites may already be available. You can add new update site URLs to the search.

Figure A.10 Choose the new features option.

3. Select New Update Site. Give this site a name, such as `Android Tools`, as shown in figure A.11. Use the following URL in the dialog: https://dl-ssl.google.com/android/eclipse. Please note the *https* in the URL. Click OK.



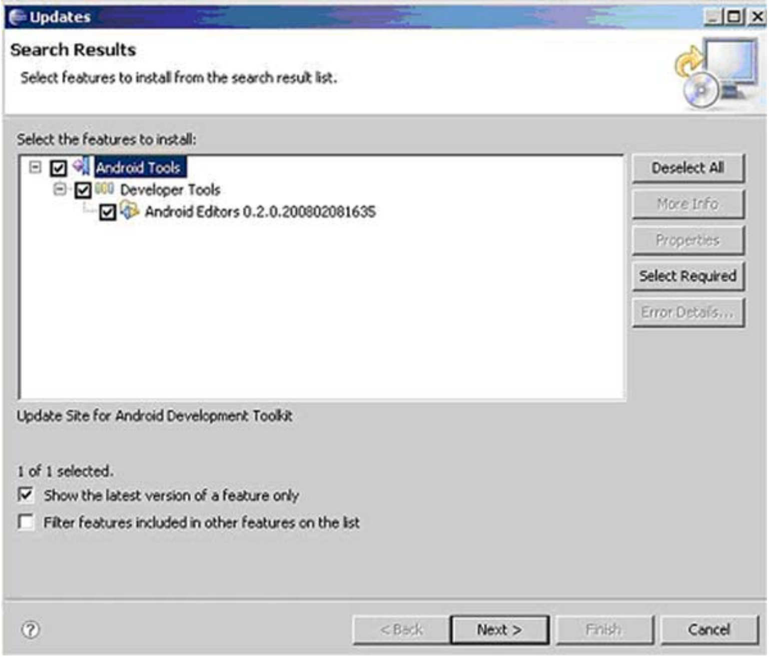
New Update Site

Name: Android Tools

URL: https://dl-ssl.google.com/android/eclipse

Figure A.11 Create a new update site to search for Android-related tools.

4. A new entry is added to the list and is checked by default. Click Finish. The search results display the ADTs.
5. Select Android Tools and click Next, as shown in figure A.12.



Updates

Search Results

Select features to install from the search result list.

Select the features to install:

☑ Android Tools
  ☑ Developer Tools
    ☑ Android Editors 0.2.0.200802081635

Update Site for Android Development Toolkit

1 of 1 selected.

☑ Show the latest version of a feature only

☐ Filter features included in other features on the list

Figure A.12 You must select Android Tools for Eclipse to download and install.

6. After reviewing and accepting the license agreement, click Next.
7. Review and accept the installation location. Click Finish.
8. The plug-in is now downloaded and installed. Restart Eclipse to complete the installation.

Congratulations! The ADT Eclipse plug-in is installed.

Note that from time to time you may need to upgrade your ADT plug-in to support a new Android API level/SDK. As of the writing of this book, the latest version of the ADT plug-in is version 8.01v201012062107-82219. To check for available ADT upgrades, select Software Updates under the Help menu in Eclipse. Highlight the Android Developer Tools in the Installed Software list and click the Upgrade button on the right side of the dialog. This will upgrade your ADT plug-in if an upgrade is available. As with the original installation of the ADT, we recommend that you restart the Eclipse IDE for the software to properly install and be available to you.

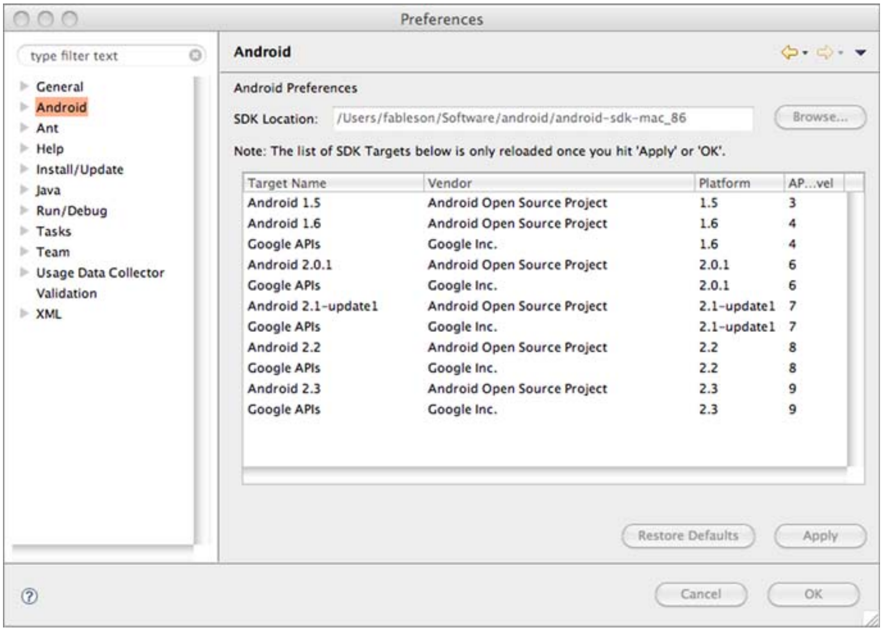
Next step: configuration.

A.6 Configuring the Eclipse plug-in

Once Eclipse is restarted, you connect the plug-in to the Android SDK installation. Select Preferences from the Window menu in Eclipse for Windows or from the Eclipse menu for Mac OS X. Click the Android item in the tree view to the left to expand the Android settings. In the right pane, specify the SDK installation location. For example, the value used for this appendix is /Users/fableson/Software/android/android-sdk-mac_86, as shown in figure A.13.

Once the SDK location is specified, there are five other sections you may configure:

- *Build*—This section has options for automatically rebuilding resources. Leave this checked. The Build option can change the level of verbosity. Normal is the default setting.



Android Preferences

SDK Location: /Users/fableson/Software/android/android-sdk-mac_86

Note: The list of SDK Targets below is only reloaded once you hit 'Apply' or 'OK'.

Target Name	Vendor	Platform	AP...vel
Android 1.5	Android Open Source Project	1.5	3
Android 1.6	Android Open Source Project	1.6	4
Google APIs	Google Inc.	1.6	4
Android 2.0.1	Android Open Source Project	2.0.1	6
Google APIs	Google Inc.	2.0.1	6
Android 2.1-update1	Android Open Source Project	2.1-update1	7
Google APIs	Google Inc.	2.1-update1	7
Android 2.2	Android Open Source Project	2.2	8
Google APIs	Google Inc.	2.2	8
Android 2.3	Android Open Source Project	2.3	9
Google APIs	Google Inc.	2.3	9

Figure A.13 ADT plug-in for Eclipse preferences

- *DDMS*—This service is used for peering into a running virtual machine. 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. Chapter 2 describes how to use the DDMS.
- *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 of the emulator.
- *LogCat?*—The LogCat feature is used to view logging messages on the device. This feature permits you to view both application-level log messages as well as kernel-level messages. Only the font is selectable in this dialog, so adjust this as desired. Don't be fooled by this simple configuration setting—the LogCat is your friend and is demonstrated throughout the book.
- *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 used in an effort to enhance the toolset.

Your Android development environment is complete!

appendix B
Publishing applications

This chapter covers

- Preparing an application for distribution
- Digitally signing an application
- Publishing applications to the Android Market and beyond

Writing and debugging applications can be both exhausting and satisfying, but the day will come when it's time to move past development and graduate to publishing your Android application for others to use. This appendix presents best practices for preparing an application for publication and then walks step by step through the process of digitally signing an application and uploading it to the Android Market.

In the most basic sense, publishing an application involves digitally signing it and uploading it to the Android Market or other venues for distribution. But for you to properly prepare an Android application, a few steps precede the distribution stage. If you observe these guidelines carefully, the odds of your customers having a positive experience with your application increase significantly. Ignore these best practices and you run the risk of tarnishing your reputation as a mobile developer.

B.1 Preparing an application for distribution

Preparing an Android application for distribution is a somewhat straightforward and methodical task, though one that requires considerable attention to detail. This section presents a list of things to consider prior to releasing your application. It's not meant to be an exhaustive list but rather a framework for cleaning up your application.

B.1.1 Logging

During development it's common to accumulate superfluous LogCat statements throughout your code. For example, you may make verbose entries to the LogCat by dumping the contents of objects to the log or recording every response from a remote server. You may even write sensitive information such as "before and after" strings related to an encryption routine. Although these entries are helpful during the debugging of your application, they can be fatal flaws if shipped to users. Imagine leaving breadcrumbs to your company's intellectual property behind in the LogCat—bad idea!

B.1.2 Debugging notifications

Your code may include the use of a `Toast` notification to inform you of some condition or scenario, such as an unhandled branch in a `switch` statement or perhaps a notification of a caught exception. This notification is helpful during the debug cycle but certainly not desirable for a released application.

B.1.3 Sample data

Your application may ship with sample data; if so, be sure that it's properly set up for your users in an intuitive manner. Also, be sure to avoid leaving behind your own data. For example, if you have an FTP application, don't leave behind your own credentials in the database shipped with your application. And don't prepopulate a form with your credit card number!

B.1.4 AndroidManifest.xml

The AndroidManifest.xml file requires careful attention before publication of your application. Let's look at a few items you need to keep in mind.

- Remove the `android:debuggable` tag, or at minimum, set its value to `false`.
- Specify appropriate values for the `label` and `icon` attributes of the application tag. Keep the text as short as possible in the label. Unless you're an artist, get someone to assist you in the creation of attractive logo artwork.
- Specify the `android:versionCode` and `android:versionName` attributes in the `<application>` element of the manifest as well. The `versionCode` is an integer value that can be checked programmatically and is typically incremented at each release. The `versionName` is displayed to users. The online documentation at http://developer.android.com covers these attributes in detail.
- Specify the minimum SDK level required for your application. For example, in the FindEdges application from chapter 19 X requires bitmap features introduced in the 2.2 version of the SDK. The SDK levels are integers that don't correspond exactly to the commonly referenced SDK versions. For example, the FindEdges application has the following line to specify the target and minimum SDK level:

```
<uses-sdk android:targetSdkVersion="8" android:minSdkVersion="8">
</uses-sdk>
```

The FindEdges application won't run on a device with an older OS version due to the bitmap requirements, but your application may not be so constrained. If your application isn't constrained, consider setting the `minSdkVersion` as low as possible to make the application accessible to a wide variety of devices. If your application contains a hard constraint, don't neglect this step because properly specifying a minimum SDK level prevents users with older devices from installing an application that can't run on their devices. This is better than allowing users to install your application and experience problems running the app.

B.1.5 End-user license agreement

We recommend providing your own end-user license agreement (EULA) even though most users will ignore it. You've written this software and likely invested heavily in its creation. You owe it to yourself and your investors to do what you can to protect your interest in it, particularly with respect to hedging against potential liabilities your application may introduce. Obtaining experienced legal counsel is a good idea unless you plan to employ one of the commonly used open source agreements—even then, consider obtaining guidance from an experienced legal expert familiar with software licensing agreements.

It's common to display the EULA when the application is first launched, requiring the user to positively acknowledge and agree to the terms. After the user has consented to the terms of the EULA, don't show it again unless the user explicitly requests to view it via a menu selection. Storing a Boolean value in the `SharedPreferences` is an easy approach to keeping track of the user's consent to EULA agreement. You may also consider showing the EULA on every upgrade in the event that your EULA is modified.

B.1.6 Testing

After you go through these steps, be sure to perform regression testing on your application on a real device prior to distribution. It's easy to break a functioning application during this cleanup phase. The purpose of this testing is to check that all of your potentially damaging debug information has been removed. You can then move on to acceptance-style testing. Having a documented test plan is a good idea, and if possible, let someone other than the primary developer be responsible for signing off on the test plan.

Be sure to run your application under as many conditions as you can with features such as Wi-Fi, 3G, and GPS both enabled and disabled. Verify that the application degrades gracefully in the event that a required service is unavailable, such as when data service is unavailable or when roaming. It's fine for your application to not perform if a missing communications link is unavailable, but the application should present an easy-to-understand message to users, advising them of the situation and perhaps suggesting steps to restore connectivity.

Pay particular attention to how your application responds to being stopped and restarted. Change the screen orientation when running each `Activity`. Does the application behave as expected? Remember that the default behavior is for an

Activity to be stopped and restarted when the screen orientation changes. You may need to return to your code and implement the Activity lifecycle methods.

B.1.7 Finishing touches

As an extra step, if your application persists data locally via a file, SharedPreferences, or a SQL database, consider providing an import/export feature. This feature can be implemented as an Activity, allowing the user to save the data out to the SD card in a readily parsable format such as CSV or XML. The import/export feature should also allow the user to restore the data to your application's local storage. This extra feature may make application upgrades easier and more resistant to errors. Having an easy export/image mechanism via SD card also makes moving to a new device a nonevent because your users can easily bring their data with them to the new device. Your users will love you for this!

Once you're convinced that your application is ready for release, it's time to digitally sign the application in preparation for taking it to the Android Market.

B.2 Digitally signing an application

The Android platform requires every application file—that is, yourappname.apk—to be digitally signed in order to run on a device or emulator; without a signature, an application simply won't run. When you use Eclipse to develop your application, Eclipse silently signs the application with an automatically provided debug key. The signing requirement is entirely transparent to most developers until it's time to publish an application for others to use.

When you're publishing an application for distribution, the application needs to be signed with a nondebug signature. Fortunately, the applications can be self-signed, meaning a certificate authority isn't required. This keeps the complexity and cost down considerably compared to the signing process required for other mobile platforms.

B.2.1 Keystores

By default, the keystores are located under the user's home directory in a folder named .android. The following listing shows the contents of this folder on the author's development machine.

Listing B.1 .android folder showing keystores

```
hostname:.android fableson$ ls -l
total 64
-rw-r--r--  1 fableson  staff   123 Jul  9 20:32 adb_usb.ini
-rw-r--r--  1 fableson  staff   198 May 22 10:28 androidtool.cfg
drwxr-xr-x  5 fableson  staff   170 Jul  9 20:45 avd
-rw-r--r--  1 fableson  staff    58 Apr 19 22:58 ddms.cfg
-rw-r--r--  1 fableson  staff  1269 Jun  2 21:23 debug.keystore
-rw-r--r--  1 fableson  staff   759 Jun 10 03:21 default.keyset
-rw-r--r--  1 fableson  staff    51 Oct 24  2009 emulator-user.ini
-rw-r--r--  1 fableson  staff  2265 Aug 15 22:02 releasekey.keystore
-rw-r--r--  1 fableson  staff    72 Jul 20 00:53 repositories.cfg
```

1 Debug keystore

2 Release keystore

The .android folder contains files and directories required by the Android Development Tools. Of particular interest here is the debug.keystore ❶, which contains the debug key used by Eclipse during normal edit, compile, install, and testing iterations. Eclipse silently signs every application with the key stored within debug.keystore. When the time comes to distribute applications to the Android Market, or other venues, a nondebug key must be created and stored in a separate keystore. In this case we've created a nondebug key and stored it in releasekey.keystore ❷. This keystore may be named arbitrarily by the developer.

The next section walks through the process of creating a nondebug key and keystore.

B.2.2 keytool

This section demonstrates the creation of a key and its containing keystore via the program named keytool. keytool is provided with the Java SDK and should be in the executable path of your terminal or command window. When this step is complete, you'll have a valid key with which an Android application may be signed for distribution.

The following command is an example of using keytool to create a self-signed private key in the .android directory:

```
keytool -genkey -v -keystore ~/.android/releasekey.keystore -alias
releasekey -keyalg
RSA -validity 10000
```

This command generates a key (-genkey) in verbose mode (-v) stored in a keystore file named releasekey.keystore with an alias of releasekey. The cryptographic algorithm is RSA and the key has a validity of 10,000 days prior to expiration. Every key in a keystore requires an alias. The alias is used when referring to the key within the keystore during the signing of the APK file. The Android documentation recommends at least a 25-year key life.

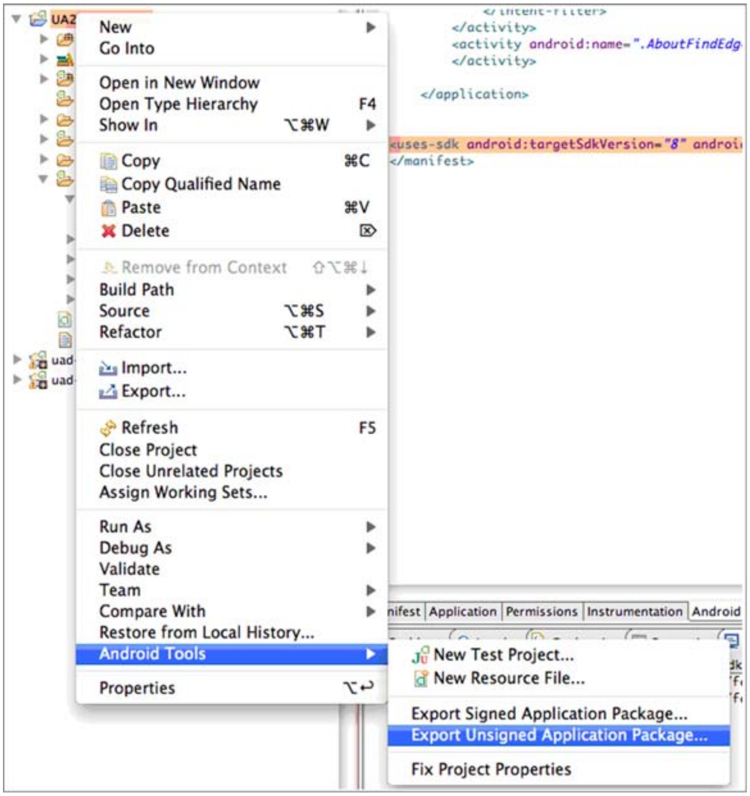
The keytool command prompts for a key password and organizational information when creating a key. You should use accurate information as it's possible for your users to view this later, and you should use a strong password. Once you create your key, you also need to be careful to store it securely and keep the password private. If your key is lost or compromised, your identity can be misused, and the trust relationships to your key via your applications can be abused.

Now that you have a valid key, it's time to sign the application. For this task, you'll utilize the jarsigner application.

B.2.3 jarsigner

Signing applications is accomplished with the jarsigner tool. Like keytool, jarsigner is part of the Java SDK so be sure that it is in your executable path.

To sign an application, you must export it as an unsigned APK file. In Eclipse, right-click and select the Android Tools > Export Unsigned Application Package option, as shown in figure B.1.



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

</application>

<uses-sdk android:targetSdkVersion="8" androi
</manifest>
```

Figure B.1 Using Android Tools from the Eclipse/ADT environment to export an unsigned application archive package

Save the file as projectname-unaligned.apk. In this example you'll export the unsigned application file for the UA2E_FindEdges application to a file named UA2E_FindEdges_unaligned.apk.

Let's now use jarsigner to sign the archive with our key, as shown here:

```
jarsigner -verbose -keystore ~/.android/releasekey.keystore
 UA2E_FindEdges-unaligned.apk releasekey
```

This command tells jarsigner to sign the APK file with a key named releasekey stored in the previously created keystore file—that is, releasekey.keystore in the ~/.android folder.

The jarsigner tool prompts for the password used when the key was created. Assuming the correct password is entered, jarsigner proceeds to sign the contents of the archive file as well as creating or updating manifest files, as shown here:

```
jarsigner -verbose -keystore ~/.android/releasekey.keystore
UA2E_FindEdges-unaligned.apk releasekey
Enter Passphrase for keystore: ****************
 updating: META-INF/RELEASEK.SF
 updating: META-INF/RELEASEK.RSA
```

```
  signing: res/layout/about.xml
  signing: res/layout/main.xml
  signing: AndroidManifest.xml
  signing: resources.arsc
  signing: res/drawable-hdpi/icon.png
  signing: res/drawable-ldpi/icon.png
  signing: res/drawable-mdpi/icon.png
  signing: classes.dex
  signing: lib/armeabi/libua2efindedges.so
  signing: lib/armeabi-v7a/libua2efindedges.so
```

Note that every file in the archive is signed, including the native JNI library files in addition to the Android SDK classes and resources.

At this point the application is ready to be installed, but there's a recommended optimization step. If all the resources within the archive are properly aligned, the Android OS can access a memory map of the file, thereby preserving the runtime RAM required because the application need not be "copied" into memory. To accomplish this alignment step, you can use the zipalign tool, which you'll find in the Android SDK/tools folder:

```
zipalign -v 4 UA2E_FindEdges_unaligned.apk UA2E_FindEdges.apk
```

The APK file is now ready for deployment to either a local device via the adb tool or for publishing to the Android Market.

To install the file to a locally attached device, use the `adb` command as follows:

```
adb install UA2E_FindEdges.apk
```

Replace the APK file with your filename, of course. To remove a currently installed application, either uninstall it from the settings application on the device or again use adb:

```
adb uninstall com.msi.manning.ua2efindedges
```

Be sure to substitute your application's package name.

Let's take a look at publishing an application to the Android Market.

B.3 Publishing to the Android Market

Every Android phone has a built-in application known as the Android Market; the label of the application says simply "Market." This application permits users to browse the extensive catalog of applications by category and price. The best way to get your application onto thousands of Android devices is to publish your application to the Market. This is done through web-based tools found at android.com.

Checking the validity of an application's license is accomplished by interacting with the License Verification Library (LVL). Interacting with this library requires the inclusion of the `com.android.vending.licensing` package and is beyond the scope of this chapter. Please examine the online documentation found at http://developer.android.com/guide/publishing/licensing.html for more details on the LVL.

B.3.1 *The Market rules*

Before you put your application on the Market, you should carefully read the developer terms (http://www.android.com/us/developer-distribution-agreement.html) and the content guidelines (http://www.android.com/market/terms/developer-content-policy.html).

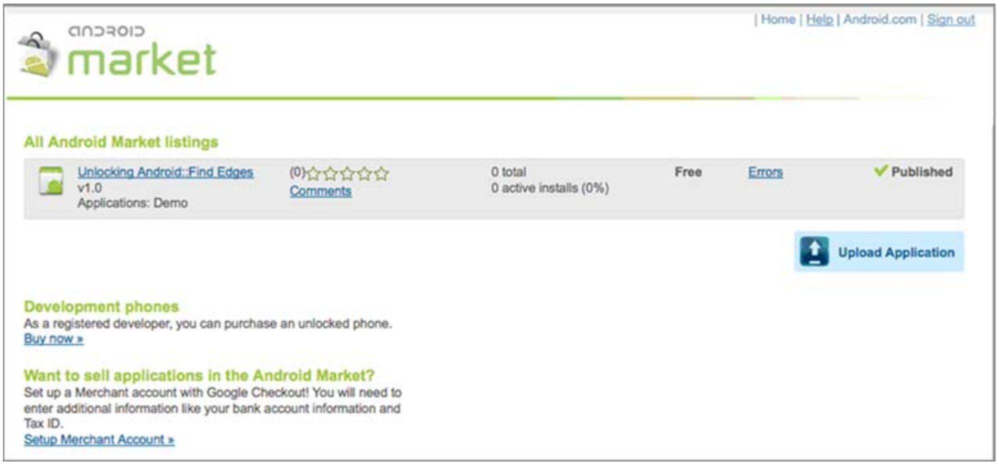
The Market terms cover pricing, payments, returns, license grants, revocations, and other relevant topics to anyone looking to publish applications to the Android Market. The content guidelines further define what's acceptable in terms of subject matter and media, though in practice an application must be very egregious to be pulled out of the Market. The bar for entry is very low.

If the Market terms are amenable to you and you plan to post applications, you need to register as an Android developer as well as have a Google account. There's a nominal fee to register as an Android developer. Once you're set up, you can begin placing your applications in the Market for users to download and install directly. Optionally, you can publish applications for a price other than "free"—that is, you can sell your software. To do so, you must also provide banking and tax identifier information.

B.3.2 *Getting your application in the Market*

Registered Market developers simply use an online form to upload applications. When uploading applications, you can define the different Market locations that are supported, pricing and terms, as well as a category and description and other options. To demonstrate the application publication process, we'll review this author's account with a single published application. The application used for this exercise is the Find Edges application created chapter 19.

Figure B.2 shows the single application listed in the Android Market, ready for maintenance. Note that this screen is also the place where new applications can added to the market by clicking the Upload Application button in the lower-right corner.



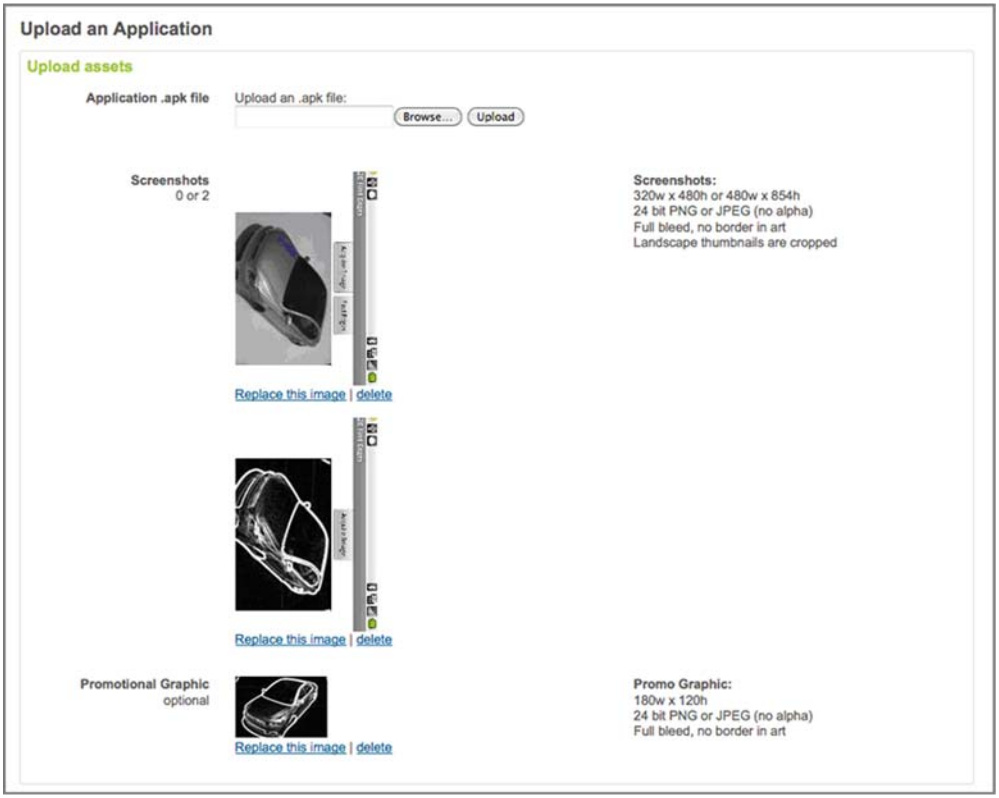
All Android Market listings

Unlocking Android::Find Edges v1.0 Applications: Demo	(0)☆☆☆☆☆ Comments	0 total 0 active installs (0%)	Free	Errors	✓ Published

Development phones
As a registered developer, you can purchase an unlocked phone.
Buy now »

Want to sell applications in the Android Market?
Set up a Merchant account with Google Checkout! You will need to enter additional information like your bank account information and Tax ID.
Setup Merchant Account »

Figure B.2 **Managing Android Market applications**



Upload an Application

Upload assets

Application .apk file

Upload an .apk file:

Screenshots
0 or 2

Screenshots:
320w x 480h or 480w x 854h
24 bit PNG or JPEG (no alpha)
Full bleed, no border in art
Landscape thumbnails are cropped

Promotional Graphic
optional

Promo Graphic:
180w x 120h
24 bit PNG or JPEG (no alpha)
Full bleed, no border in art

Browse...
Upload
Replace this image | delete
Replace this image | delete
Replace this image | delete

Figure B.3 **Managing the APK file and screenshots**

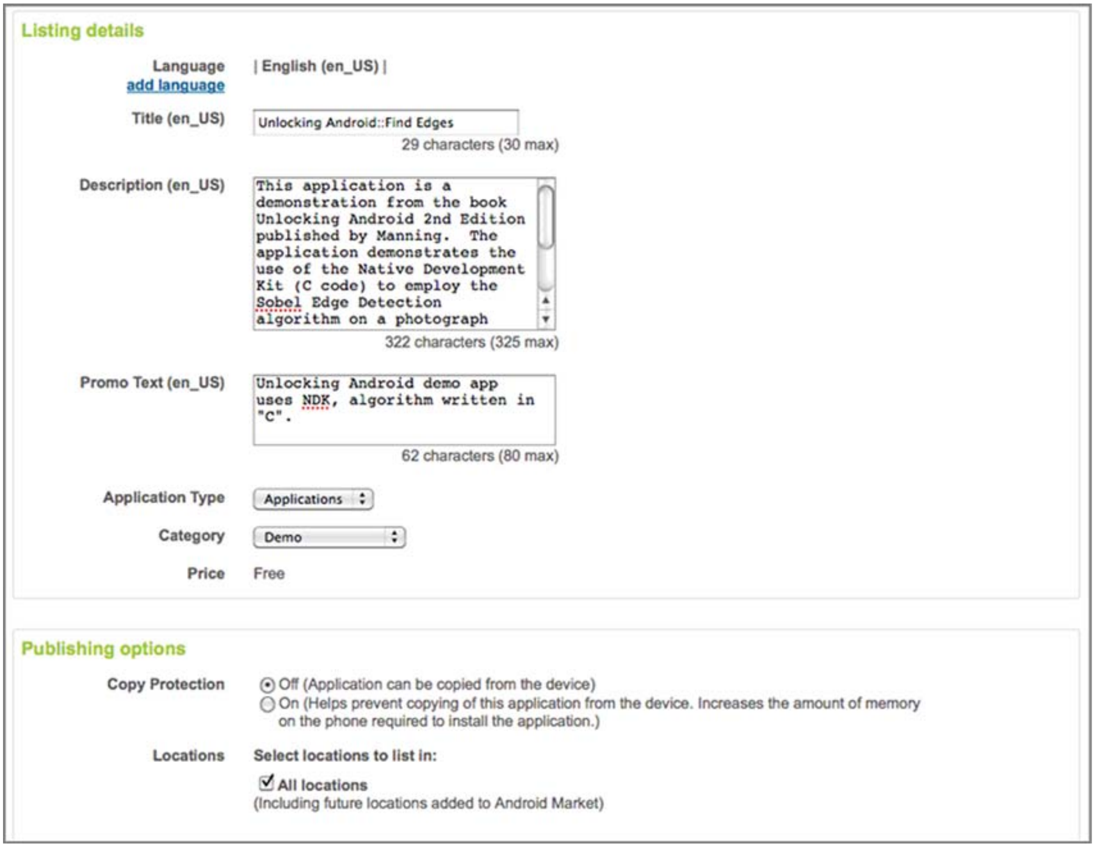
Clicking through the application allows you to edit this application's properties on the Market. The editing screen is too large to fit into one screenshot, so it's split between two figures. Figure B.3 shows the top portion of the management interface where updates to the APK file can be loaded along with screenshots to display the application to prospective users browsing in the Market.

Figure B.4 shows the textual aspects of the application description.

Once the application is published to the market, it's visible to the Market application on the Android devices worldwide within moments. Figure B.5 shows a screenshot of the Market application running on a physical Android device. The Market application is displaying the catalog entry for the Find Edges application.

After that simple process your application is available for download to users across the globe.

The Android Market is easy to use—but is it effective?



Listing details

Language | English (en_US) |
add language

Title (en_US): Unlocking Android::Find Edges
29 characters (30 max)

Description (en_US): This application is a demonstration from the book Unlocking Android 2nd Edition published by Manning. The application demonstrates the use of the Native Development Kit (C code) to employ the Sobel Edge Detection algorithm on a photograph
322 characters (325 max)

Promo Text (en_US): Unlocking Android demo app uses NDK, algorithm written in "C".
62 characters (80 max)

Application Type: Applications

Category: Demo

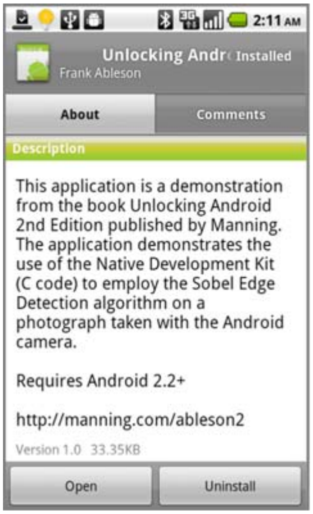
Price: Free

Publishing options

Copy Protection
● Off (Application can be copied from the device)
○ On (Helps prevent copying of this application from the device. Increases the amount of memory on the phone required to install the application.)

Locations: Select locations to list in:
☑ All locations
(Including future locations added to Android Market)

Figure B.4 Textual descriptions of application along with publication options



2:11 AM

Unlocking Andr‹ Installed
Frank Ableson

About | Comments

Description

This application is a demonstration from the book Unlocking Android 2nd Edition published by Manning. The application demonstrates the use of the Native Development Kit (C code) to employ the Sobel Edge Detection algorithm on a photograph taken with the Android camera.

Requires Android 2.2+

http://manning.com/ableson2

Version 1.0 33.35KB

Open | Uninstall

Figure B.5 The newly published application becomes available on the device in mere moments.

B.3.3 Android Market—the right solution

The Android Market is an effective distribution mechanism because it's built in and accessible to users and developers alike. Generally speaking, it's the first place users go to find applications.

As mentioned in chapter 1, the open nature of the Android platform—and of the Android Market—offers distinct advantages to both developers and users. There's no arbitrary inclusion or exclusion process that an individual or company holds over the Market. Anyone who joins and agrees to the terms can publish applications on the Market without fear of the thought police barring an application.

Virtually all applications are welcome, but some will do better than others. Users rate the applications on a scale of 1 to 5, and they may leave comments as well. These comments often influence the prospective purchasers with their positive or negative remarks. The Android Market is a merit-based system; impress your users and they'll rate your application well and compliment you; shortchange your users and your download count and sales will suffer.

In keeping with the theme of being an open platform, Android applications may be distributed beyond just the Android Market.

B.4 Other distribution means

The last thing to consider with regard to distributing applications is that there are other venues beyond the Android Market.

Various third-party sites also offer distribution channels. These sites have different agreement types and different payment models, so you should research them carefully before using them.

You may want to distribute your application in the Android Market or on third-party services, or you may decide to use multiple distribution channels. If you do use third-party services, keep in mind that although these alternatives are growing in popularity they aren't nearly as accessible to users compared to the built-in Market. Users have to find the third-party service and generally then have to install applications on their own or at least bootstrap the service with an application specifically designed for the market.

Last and certainly not least, Android applications can be distributed directly from a company's website or any other means of shipping an APK file. For commercial developers targeting enterprise applications, this is a distinct advantage over the draconian measures taken by Apple for iPhone and iPad application deployment.

The more means you have at your disposal to get your applications into the hands of users, the greater your influence. Good luck!

Some applications can be developed working exclusively with the emulator whereas others require a real device from day one of development because the emulator can't adequately deliver the complete functionality required for things such as Bluetooth connectivity or taking a photo with a built-in camera. Whether or not your application needs the real device during development, it's a good practice to test all applications on physical hardware before publishing an Android application.

B.5 Recapping the Android Debug Bridge

Although we covered the Android Debug Bridge (adb) in chapter 2, a recap is in order as background for signing and installing applications and working with Android devices.

The adb is a client/server program that lets you interact with the Android SDK in various ways, including pushing and pulling files, installing and removing applications, issuing shell commands, and more. The adb tool consist of three components: a development machine–based server, a development machine client for issuing commands, and a client for each emulator or device in use. Other Android tools, such as the DDMS tool, also create clients to interact with the adb.

You can make sure your local adb server is running by issuing the `adb start-server` command. Similarly, you can stop your server with `adb kill-server` and then restart it, if necessary (or just to get familiar with the process). When you start the Eclipse/ADT environment, it automatically starts an adb server instance.

Once you're sure your adb server is running, you can tell if any devices or emulators are connected by issuing the `adb devices` command. The output of this command with an emulator running and a physical device attached via a USB cable is shown here:

```
#$ adb devices
List of devices attached
emulator-5554    device
HT845GZ49611     device
```

The adb tool acts as the backplane for the Android development process. Communications between the development environment and a device/emulator rely on adb. The first step in getting your applications onto an actual device is to connect your device to the development machine and confirm that it's recognized by adb. If you have any running emulator instances, it'd be a good idea to shut them down prior to beginning this process. Confirm that your physical device is the only attached Android client. To do this, run `adb devices` from the command line and confirm that there's a single entry in the list of attached devices, as shown in figure B.1.

index