UPDATED FOR ANDROID 4

Join the discussion @ p2p.wrox.com

Professional Android™ 4 Application Development

Reto Meier

Table of Contents

Chapter 1

Hello, Android

What's in this Chapter?

- A background of mobile application development
- What Android is (and what it isn't)
- An introduction to the Android SDK features
- Which devices Android runs on
- Why you should develop for mobile and Android
- An introduction to the Android SDK and development framework

Whether you're an experienced mobile engineer, a desktop or web developer, or a complete programming novice, Android represents an exciting new opportunity to write innovative applications for an increasingly wide range of devices.

Despite the name, Android will not help you create an unstoppable army of emotionless robot warriors on a relentless quest to cleanse the earth of the scourge of humanity. Instead, Android is an open-source software

stack that includes the operating system, middleware, and key mobile applications, along with a set of API libraries for writing applications that can shape the look, feel, and function of the devices on which they run.

Small, stylish, and versatile, modern mobile devices have become powerful tools that incorporate touchscreens, cameras, media players, Global Positioning System (GPS) receivers, and Near Field Communications (NFC) hardware. As technology has evolved, mobile phones have become about much more than simply making calls. With the introduction of tablets and Google TV, Android has expanded beyond its roots as a mobile phone operating system, providing a consistent platform for application development across an increasingly wide range of hardware.

In Android, native and third-party applications are written with the same APIs and executed on the same run time. These APIs feature hardware access, video recording, location-based services, support for background services, map-based activities, relational databases, inter-application communication, Bluetooth, NFC, and 2D and 3D graphics.

This book describes how to use these APIs to create your own Android applications. In this chapter you'll learn some guidelines for mobile and embedded hardware development, as well as be introduced to some of the platform features available for Android development.

Android has powerful APIs, excellent documentation, a

thriving developer community, and no development or distribution costs. As mobile devices continue to increase in popularity, and Android devices expand into exciting new form-factors, you have the opportunity to create innovative applications no matter what your development experience.

A Little Background

In the days before Twitter and Facebook, when Google was still a twinkle in its founders' eyes and dinosaurs roamed the earth, mobile phones were just that—portable phones small enough to fit inside a briefcase, featuring batteries that could last up to several hours. They did, however, offer the freedom to make calls without being physically connected to a landline.

Increasingly small, stylish, and powerful, mobile phones are now ubiquitous and indispensable. Hardware advancements have made mobiles smaller and more efficient while featuring bigger, brighter screens and including an increasing number of hardware peripherals.

After first including cameras and media players, mobiles now feature GPS receivers, accelerometers, NFC hardware, and high-definition touchscreens. These hardware innovations offer fertile ground for software development, but until relatively recently the applications available for mobile phones have lagged behind their hardware counterparts.

The Not-So-Distant Past

Historically, developers, generally coding in low-level C or C++, have needed to understand the specific hardware they were coding for, typically a single device or possibly a range of devices from a single manufacturer. As hardware technology and mobile Internet access has advanced, this closed approach has become outmoded.

Platforms such as Symbian were later created to provide developers with a wider target audience. These systems proved more successful in encouraging mobile developers to provide rich applications that better leveraged the hardware available.

Although these platforms did, and continue to, offer some access to the device hardware, they generally required developers to write complex C/C++ code and make heavy use of proprietary APIs that are notoriously difficult to work with. This difficulty is amplified for applications that must work on different hardware implementations and those that make use of a particular hardware feature, such as GPS.

In more recent years, the biggest advance in mobile phone development was the introduction of Java-hosted MIDlets. MIDlets are executed on a Java virtual machine (JVM), a process that abstracts the underlying hardware and lets developers create applications that run on the wide variety of devices that support the Java run time.

Unfortunately, this convenience comes at the price of restricted access to the device hardware.

In mobile development, it was long considered normal for third-party applications to receive different hardware access and execution rights from those given to native applications written by the phone manufacturers, with MIDlets often receiving few of either.

The introduction of Java MIDlets expanded developers' audiences, but the lack of low-level hardware access and sandboxed execution meant that most mobile applications were regular desktop programs or websites designed to render on a smaller screen, and didn't take advantage of the inherent mobility of the handheld platform.

Living in the Future

Android sits alongside a new wave of modern mobile operating systems designed to support application development on increasingly powerful mobile hardware. Platforms like Microsoft's Windows Phone and the Apple iPhone also provide a richer, simplified development environment for mobile applications; however, unlike Android, they're built on proprietary operating systems. In some cases they prioritize native applications over those created by third parties, restrict communication among applications and native phone data, and restrict or control the distribution of third-party applications to their platforms.

Android offers new possibilities for mobile applications by offering an open development environment built on an open-source Linux kernel. Hardware access is available to all applications through a series of API libraries, and application interaction, while carefully controlled, is fully supported.

In Android, all applications have equal standing. Third-party and native Android applications are written with the same APIs and are executed on the same run time. Users can remove and replace any native application with a third-party developer's alternative; indeed, even the dialer and home screens can be replaced.

What Android Isn't

As a disruptive addition to a mature field, it's not hard to see why there has been some confusion about what exactly Android is. Android is *not* the following:

- **A Java ME implementation**—Android applications are written using the Java language, but they are not run within a Java ME (Mobile Edition) VM, and Java-compiled classes and executables will not run natively in Android.
- **Part of the Linux Phone Standards Forum (LiPS) or the Open Mobile Alliance (OMA)**—Android runs on an open-source Linux kernel, but, while their goals are similar, Android's complete software stack approach goes further than the focus of these standards-defining organizations.
- **Simply an application layer (such as UIQ or S60)**—Although Android does include an application layer, “Android” also describes the entire software stack, encompassing the underlying operating system, the API libraries, and the applications themselves.
- **A mobile phone handset**—Android includes a reference design for mobile handset manufacturers,

but there is no single “Android phone.” Instead, Android has been designed to support many alternative hardware devices.

- **Google's answer to the iPhone**—The iPhone is a fully proprietary hardware and software platform released by a single company (Apple), whereas Android is an open-source software stack produced and supported by the Open Handset Alliance (OHA) and designed to operate on any compatible device.

Android: An Open Platform for Mobile Development

Google's Andy Rubin describes Android as follows:

> *The first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications—all of the software to run a mobile phone but without the proprietary obstacles that have hindered mobile innovation.*
>
> —Where's My Gphone? (*http://googleblog.blogspot.com/2007/11/wheres-my-gphone.html*)

More recently, Android has expanded beyond a pure mobile phone platform to provide a development platform for an increasingly wide range of hardware, including tablets and televisions.

Put simply, Android is an ecosystem made up of a combination of three components:

- A free, open-source operating system for embedded devices

- An open-source development platform for creating applications
- Devices, particularly mobile phones, that run the Android operating system and the applications created for it

More specifically, Android is made up of several necessary and dependent parts, including the following:

- A Compatibility Definition Document (CDD) and Compatibility Test Suite (CTS) that describe the capabilities required for a device to support the software stack.
- A Linux operating system kernel that provides a low-level interface with the hardware, memory management, and process control, all optimized for mobile and embedded devices.
- Open-source libraries for application development, including SQLite, WebKit, OpenGL, and a media manager.
- A run time used to execute and host Android applications, including the Dalvik Virtual Machine (VM) and the core libraries that provide Android-specific functionality. The run time is designed to be small and efficient for use on mobile devices.
- An application framework that agnostically exposes system services to the application layer, including the window manager and location manager, databases,

telephony, and sensors.
- A user interface framework used to host and launch applications.
- A set of core pre-installed applications.
- A software development kit (SDK) used to create applications, including the related tools, plug-ins, and documentation.

What really makes Android compelling is its open philosophy, which ensures that you can fix any deficiencies in user interface or native application design by writing an extension or replacement. Android provides you, as a developer, with the opportunity to create mobile phone interfaces and applications designed to look, feel, and function exactly as you imagine them.

Native Android Applications

Android devices typically come with a suite of preinstalled applications that form part of the Android Open Source Project (AOSP), including, but not necessarily limited to, the following:

- An e-mail client
- An SMS management application
- A full PIM (personal information management) suite, including a calendar and contacts list
- A WebKit-based web browser
- A music player and picture gallery
- A camera and video recording application
- A calculator
- A home screen
- An alarm clock

In many cases Android devices also ship with the following proprietary Google mobile applications:

- The Google Play Store for downloading third-party Android applications

- A fully featured mobile Google Maps application, including StreetView, driving directions, and turn-by-turn navigation, satellite views, and traffic conditions
- The Gmail email client
- The Google Talk instant-messaging client
- The YouTube video player

The data stored and used by many of these native applications—such as contact details—are also available to third-party applications. Similarly, your applications can respond to events such as incoming calls.

The exact makeup of the applications available on new Android phones is likely to vary based on the hardware manufacturer and/or the phone carrier or distributor.

The open-source nature of Android means that carriers and OEMs can customize the user interface and the applications bundled with each Android device. Several OEMs have done this, including HTC with Sense, Motorola with MotoBlur, and Samsung with TouchWiz.

It's important to note that for compatible devices, the underlying platform and SDK remain consistent across OEM and carrier variations. The look and feel of the user interface may vary, but your applications will function in the same way across all compatible Android devices.

Android SDK Features

The true appeal of Android as a development environment lies in its APIs.

As an application-neutral platform, Android gives you the opportunity to create applications that are as much a part of the phone as anything provided out-of-the-box. The following list highlights some of the most noteworthy Android features:

- GSM, EDGE, 3G, 4G, and LTE networks for telephony or data transfer, enabling you to make or receive calls or SMS messages, or to send and retrieve data across mobile networks
- Comprehensive APIs for location-based services such as GPS and network-based location detection
- Full support for applications that integrate map controls as part of their user interfaces
- Wi-Fi hardware access and peer-to-peer connections
- Full multimedia hardware control, including playback and recording with the camera and microphone
- Media libraries for playing and recording a variety of audio/video or still-image formats
- APIs for using sensor hardware, including

accelerometers, compasses, and barometers
- Libraries for using Bluetooth and NFC hardware for peer-to-peer data transfer
- IPC message passing
- Shared data stores and APIs for contacts, social networking, calendar, and multi-media
- Background Services, applications, and processes
- Home-screen Widgets and Live Wallpaper
- The ability to integrate application search results into the system searches
- An integrated open-source HTML5 WebKit-based browser
- Mobile-optimized, hardware-accelerated graphics, including a path-based 2D graphics library and support for 3D graphics using OpenGL ES 2.0
- Localization through a dynamic resource framework
- An application framework that encourages the reuse of application components and the replacement of native applications

Access to Hardware, Including Camera, GPS, and Sensors

Android includes API libraries to simplify development involving the underlying device hardware. They ensure that you don't need to create specific implementations of your software for different devices, so you can create Android applications that work as expected on any device that supports the Android software stack.

The Android SDK includes APIs for location-based hardware (such as GPS), the camera, audio, network connections, Wi-Fi, Bluetooth, sensors (including accelerometers), NFC, the touchscreen, and power management. You can explore the possibilities of some of Android's hardware APIs in more detail in Chapters 12 and 15–17.

Data Transfers Using Wi-Fi, Bluetooth, and NFC

Android offers rich support for transferring data between devices, including Bluetooth, Wi-Fi Direct, and Android Beam. These technologies offer a rich variety of techniques for sharing data between paired devices, depending on the hardware available on the underlying device, allowing you to create innovative collaborative applications.

In addition, Android offers APIs to manage your network connections, Bluetooth connections, and NFC tag reading.

Details on using Android's communications APIs are available in Chapter 16, “Bluetooth, NFC, Networks, and Wi-Fi.”

Maps, Geocoding, and Location-Based Services

Embedded map support enables you to create a range of map-based applications that leverage the mobility of Android devices. Android lets you design user interfaces that include interactive Google Maps that you can control programmatically and annotate using Android's rich graphics library.

Android's location-based services manage technologies such as GPS and Google's network-based location technology to determine the device's current position. These services enforce an abstraction from specific location-detecting technology and let you specify minimum requirements (e.g., accuracy or cost) rather than selecting a particular technology. This also means your location-based applications will work no matter what technology the host device supports.

To combine maps with locations, Android includes an API for forward and reverse geocoding that lets you find map coordinates for an address, and the address of a map position.

You'll learn the details of using maps, the geocoder, and location-based services in Chapter 13, “Maps, Geocoding, and Location-

Based Services."

Background Services

Android supports applications and services designed to run in the background while your application isn't being actively used.

Modern mobiles and tablets are by nature multifunction devices; however, their screen sizes and interaction models mean that generally only one interactive application is visible at any time. Platforms that don't support background execution limit the viability of applications that don't need your constant attention.

Background services make it possible to create invisible application components that perform automatic processing without direct user action. Background execution allows your applications to become event-driven and to support regular updates, which is perfect for monitoring game scores or market prices, generating location-based alerts, or prioritizing and prescreening incoming calls and SMS messages.

Notifications are the standard means by which a mobile device traditionally alerts users to events that have happened in a background application. Using the Notification Manager, you can trigger audible alerts, cause vibration, and flash the device's LED, as well as control status bar notification icons.

Learn more about how to use Notifications and get the most out of background services in Chapters 9 and 10.

SQLite Database for Data Storage and Retrieval

Rapid and efficient data storage and retrieval are essential for a device whose storage capacity is relatively limited.

Android provides a lightweight relational database for each application via SQLite. Your applications can take advantage of this managed relational database engine to store data securely and efficiently.

By default, each application database is sandboxed—its content is available only to the application that created it—but Content Providers supply a mechanism for the managed sharing of these application databases as well as providing an abstraction between your application and the underlying data source.

Databases and Content Providers are covered in detail in Chapter 8, “Databases and Content Providers.”

Shared Data and Inter-Application Communication

Android includes several techniques for making information from your applications available for use elsewhere, primarily: Intents and Content Providers.

Intents provide a mechanism for message-passing within and between applications. Using Intents, you can broadcast a desired action (such as dialing the phone or editing a contact) systemwide for other applications to handle. Using the same mechanism, you can register your own application to receive these messages or execute the requested actions.

You can use *Content Providers* to provide secure, managed access to your applications' private databases. The data stores for native applications, such as the contact manager, are exposed as Content Providers so you can read or modify this data from within your own applications.

Intents are a fundamental component of Android and are covered in depth in Chapter 5, “Intents and Broadcast Receivers.”
Chapter 8 covers content providers in detail, including the native providers, and demonstrates how to create and use providers of your own.

Using Widgets and Live Wallpaper to Enhance the Home Screen

Widgets and Live Wallpaper let you create dynamic application components that provide a window into your applications, or offer useful and timely information, directly on the home screen.

Offering a way for users to interact with your application directly from the home screen increases user engagement by giving them instant access to interesting information without needing to open the application, as well as adding a dynamic shortcut into your application from their home screen.

You'll learn how to create application components for the home screen in Chapter 14, “Invading the Home Screen.”

Extensive Media Support and 2D/3D Graphics

Bigger screens and brighter, higher-resolution displays have helped make mobiles multimedia devices. To help you make the most of the hardware available, Android provides graphics libraries for 2D canvas drawing and 3D graphics with OpenGL.

Android also offers comprehensive libraries for handling still images, video, and audio files, including the MPEG4, H.264, HTTP Live Streaming, VP8, WEBP, MP3, AAC, AMR, HLS, JPG, PNG, and GIF formats.

2D and 3D graphics are covered in depth in Chapter 11, “Advanced User Experience,” and Android media management libraries are covered in Chapter 15, “Audio, Video, and Using the Camera.”

Cloud to Device Messaging

The Android Cloud to Device Messaging (C2DM) service provides an efficient mechanism for developers to create event-driven applications based on server-side pushes.

Using C2DM you can create a lightweight, always-on connection between your mobile application and your server, allowing you to send small amounts of data directly to your device in real time.

The C2DM service is typically used to prompt applications of new data available on the server, reducing the need for polling, decreasing the battery impact of an application's updates, and improving the timeliness of those updates.

Optimized Memory and Process Management

Like Java and .NET, Android uses its own run time and VM to manage application memory. Unlike with either of these other frameworks, the Android run time also manages the process lifetimes. Android ensures application responsiveness by stopping and killing processes as necessary to free resources for higher-priority applications.

In this context, the highest priority is given to the application with which the user is interacting. Ensuring that your applications are prepared for a swift death but are still able to remain responsive, and to update or restart in the background if necessary, is an important consideration in an environment that does not allow applications to control their own lifetimes.

You will learn more about the Android application lifecycle in Chapter 3, “Creating Applications and Activities.”

Introducing the Open Handset Alliance

The Open Handset Alliance (OHA) is a collection of more than 80 technology companies, including hardware manufacturers, mobile carriers, software developers, semiconductor companies, and commercialization companies. Of particular note are the prominent mobile technology companies, including Samsung, Motorola, HTC, T-Mobile, Vodafone, ARM, and Qualcomm. In their own words, the OHA represents the following:

> *A commitment to openness, a shared vision for the future, and concrete plans to make the vision a reality. To accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience.*
>
> —www.openhandsetalliance.com

The OHA hopes to deliver a better mobile software experience for consumers by providing the platform needed for innovative mobile development at a faster rate and with higher quality than existing platforms, without licensing fees for either software developers or handset manufacturers.

What Does Android Run On?

The first Android mobile handset, the T-Mobile G1, was released in the United States in October 2008. By the beginning of 2012, more than 300 million Android-compatible devices have been sold from more than 39 manufacturers, in more than 123 countries, on 231 different carrier networks.

Rather than being a mobile OS created for a single hardware implementation, Android is designed to support a large variety of hardware platforms, from smartphones to tablets and televisions.

With no licensing fees or proprietary software, the cost to handset manufacturers for providing Android devices is comparatively low. Many people now expect that the advantages of Android as a platform for creating powerful applications will encourage device manufacturers to produce increasingly diverse and tailored hardware.

Why Develop for Mobile?

In market terms, the emergence of modern mobile smartphones—multifunction devices including a phone but featuring a full-featured web browser, cameras, media players, Wi-Fi, and location-based services—has fundamentally changed the way people interact with their mobile devices and access the Internet.

Mobile-phone ownership easily surpasses computer ownership in many countries, with more than 3 billion mobile phone users worldwide. 2009 marked the year that more people accessed the Internet for the first time from a mobile phone rather than a PC. Many people believe that within the next 5 years more people will access the Internet by mobile phone rather than using personal computers.

The increasing popularity of modern smartphones, combined with the increasing availability of high-speed mobile data and Wi-Fi hotspots, has created a huge opportunity for advanced mobile applications.

The ubiquity of mobile phones, and our attachment to them, makes them a fundamentally different platform for development from PCs. With a microphone, camera, touchscreen, location detection, and environmental sensors, a phone can effectively become an extra-sensory perception device.

Smartphone applications have changed the way people use their phones. This gives you, the application developer, a unique opportunity to create dynamic, compelling new applications that become a vital part of people's lives.

Why Develop for Android?

Android represents a clean break, a mobile framework based on the reality of modern mobile devices designed by developers, for developers.

With a simple, powerful, and open SDK, no licensing fees, excellent documentation, and a thriving developer community, Android represents an opportunity to create software that changes how and why people use their mobile phones.

The barrier to entry for new Android developers is minimal:

- No certification is required to become an Android developer.
- Google Play provides free, up-front purchase, and in-app billing options for distribution and monetization of your applications.
- There is no approval process for application distribution.
- Developers have total control over their brands.

From a commercial perspective, more than 850,000 new Android devices are activated daily, with many studies showing the largest proportion of new smartphone sales

belonging to Android devices.

As of March 2012, Google Play (formerly Android Market) has expanded its support for application sales to 131 countries, supporting more than 10 billion installs at a growth rate of 1 billion downloads per month.

Factors Driving Android's Adoption

Developers have always been a critical element within the Android ecosystem, with Google and the OHA betting that the way to deliver better mobile software to consumers is to make it easier for developers to write it.

As a development platform, Android is powerful and intuitive, enabling developers who have never programmed for mobile devices to create innovative applications quickly and easily. It's easy to see how compelling Android applications have created demand for the devices necessary to run them, particularly when developers write applications for Android because they *can't* write them for other platforms.

As Android expands into more form-factors, with increasingly powerful hardware, advanced sensors, and new developer APIs, the opportunities for innovation will continue to grow.

Open access to the nuts and bolts of the underlying system is what's always driven software development and platform adoption. The Internet's inherent openness has seen it become the platform for a multibillion-dollar industry within 10 years of its inception. Before that, it was open systems such as Linux and the powerful APIs provided as part of the Windows operating system that enabled the explosion in personal computers and the movement of

computer programming from the arcane to the mainstream.

This openness and power ensure that anyone with the inclination can bring a vision to life at minimal cost.

What Android Has That Other Platforms Don't Have

Many of the features listed previously, such as 3D graphics and native database support, are also available in other native mobile SDKs, as well as becoming available on mobile browsers.

The pace of innovation in mobile platforms, both Android and its competitors, makes an accurate comparison of the available features difficult. The following noncomprehensive list details some of the features available on Android that may not be available on all modern mobile development platforms:

- **Google Maps applications**—Google Maps for Mobile has been hugely popular, and Android offers a Google Map as an atomic, reusable control for use in your applications. The Map View lets you display, manipulate, and annotate a Google Map within your Activities to build map-based applications using the familiar Google Maps interface.
- **Background services and applications**—Full support for background applications and services lets you create applications based on an event-driven model, working silently while other applications are being used or while your mobile sits ignored until it

rings, flashes, or vibrates to get your attention. Maybe it's a streaming music player, an application that tracks the stock market, alerting you to significant changes in your portfolio, or a service that changes your ringtone or volume depending on your current location, the time of day, and the identity of the caller. Android provides the same opportunities for all applications and developers.

- **Shared data and inter-process communication**—Using Intents and Content Providers, Android lets your applications exchange messages, perform processing, and share data. You can also use these mechanisms to leverage the data and functionality provided by the native Android applications. To mitigate the risks of such an open strategy, each application's process, data storage, and files are private unless explicitly shared with other applications via a full permission-based security mechanism, as detailed in Chapter 18, “Advanced Android Development.”
- **All applications are created equal**—Android doesn't differentiate between native applications and those developed by third parties. This gives consumers unprecedented power to change the look and feel of their devices by letting them completely replace every native application with a third-party alternative that has access to the same underlying data and hardware.
- **Wi-Fi Direct and Android Beam**—Using these

innovative new inter-device communication APIs, you can include features such as instant media sharing and streaming. Android Beam is an NFC-based API that lets you provide support for proximity-based interaction, while Wi-Fi Direct offers a wider range peer-to-peer for reliable, high-speed communication between devices.

- **Home-screen Widgets, Live Wallpaper, and the quick search box**—Using Widgets and Live Wallpaper, you can create windows into your application from the phone's home screen. The quick search box lets you integrate search results from your application directly into the phone's search functionality.

The Changing Mobile Development Landscape

Existing mobile development platforms have created an aura of exclusivity around mobile development. In contrast, Android allows, even encourages, radical change.

As consumer devices, Android handsets ship with a core set of the standard applications that consumers expect on a new phone, but the real power lies in users' ability to completely customize their devices' look, feel, and function —giving application developers an exciting opportunity.

All Android applications are a native part of the phone, not just software that's run in a sandbox on top of it. Rather than writing small-screen versions of software that can be run on low-power devices, you can now build mobile applications that change the way people use their phones.

The field of mobile development is currently enjoying a period of rapid innovation and incredible growth. This provides both challenges and opportunities for developers simply to keep up with the pace of change, let alone identify the opportunities these changes make possible.

Android will continue to advance and improve to compete with existing and future mobile development platforms, but as an open-source developer framework, the strength of the SDK is very much in its favor. Its free and open approach to mobile application development, with

total access to the phone's resources, represents an opportunity for any mobile developer looking to seize the opportunities now available in mobile development.

Introducing the Development Framework

With the “why” covered, let's take a look at the “how.”

Android applications normally are written using Java as the programming language but executed by means of a custom VM called *Dalvik*, rather than a traditional Java VM.

Later in this chapter you'll be introduced to the framework, starting with a technical explanation of the Android software stack, followed by a look at what's included in the SDK, an introduction to the Android libraries, and a look at the Dalvik VM.

Each Android application runs in a separate process within its own Dalvik instance, relinquishing all responsibility for memory and process management to the Android run time, which stops and kills processes as necessary to manage resources.

Dalvik and the Android run time sit on top of a Linux kernel that handles low-level hardware interaction, including drivers and memory management, while a set of APIs provides access to all the underlying services, features, and hardware.

What Comes in the Box

The Android SDK includes everything you need to start developing, testing, and debugging Android applications:

- **The Android APIs**—The core of the SDK is the Android API libraries that provide developer access to the Android stack. These are the same libraries that Google uses to create native Android applications.
- **Development tools**—The SDK includes several development tools that let you compile and debug your applications so that you can turn Android source code into executable applications. You will learn more about the developer tools in Chapter 2, "Getting Started."
- **The Android Virtual Device Manager and emulator**—The Android emulator is a fully interactive mobile device emulator featuring several alternative skins. The emulator runs within an Android Virtual Device (AVD) that simulates a device hardware configuration. Using the emulator you can see how your applications will look and behave on a real Android device. All Android applications run within the Dalvik VM, so the software emulator is an excellent development environment—in fact, because

it's hardware-neutral, it provides a better independent test environment than any single hardware implementation.

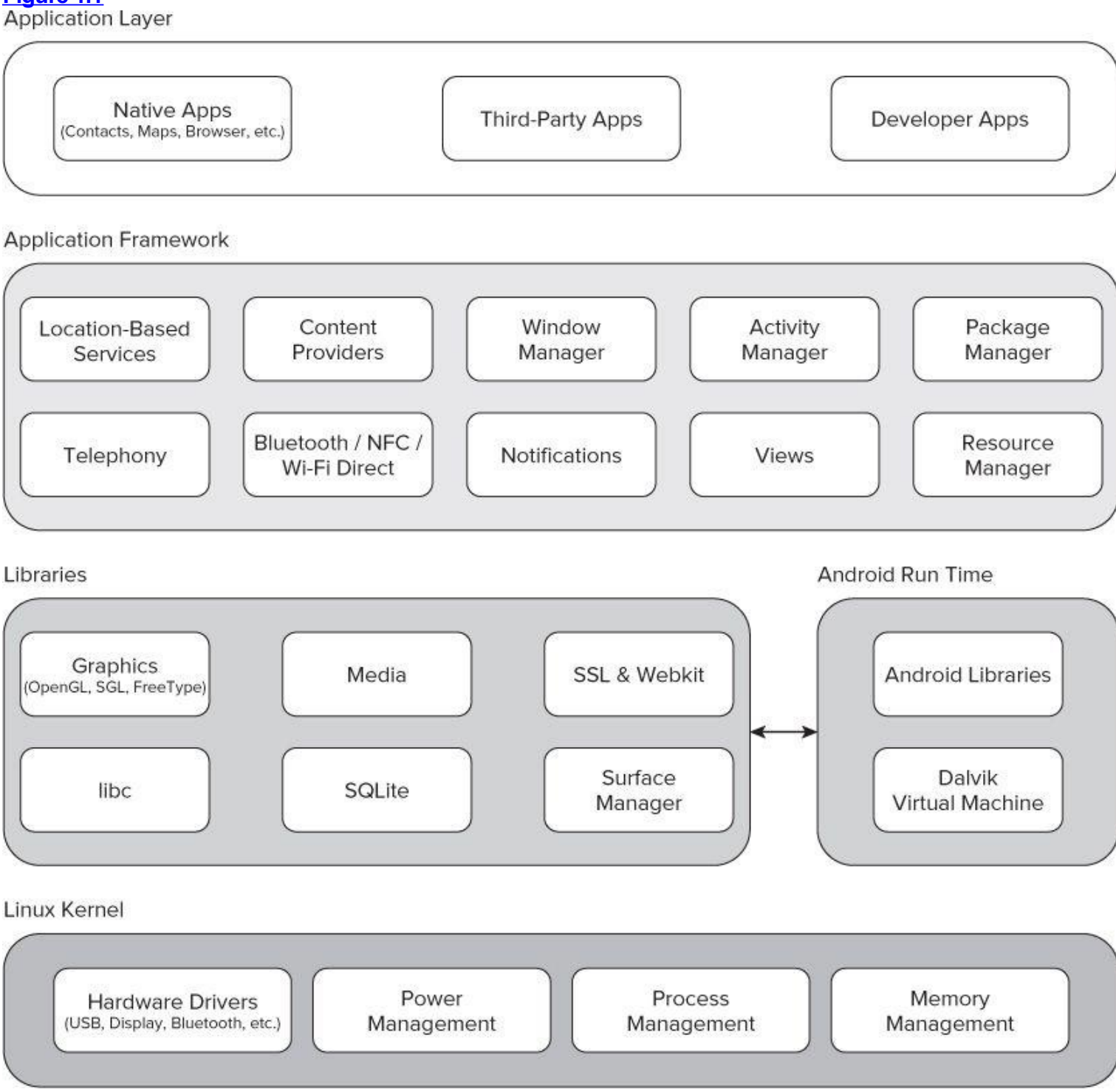
- **Full documentation**—The SDK includes extensive code-level reference information detailing exactly what's included in each package and class and how to use them. In addition to the code documentation, Android's reference documentation and developer guide explains how to get started, gives detailed explanations of the fundamentals behind Android development, highlights best practices, and provides deep-dives into framework topics.
- **Sample code**—The Android SDK includes a selection of sample applications that demonstrate some of the possibilities available with Android, as well as simple programs that highlight how to use individual API features.
- **Online support**—Android has rapidly generated a vibrant developer community. The Google Groups (http://developer.android.com/resources/community-groups .html#ApplicationDeveloperLists) are active forums of Android developers with regular input from the Android engineering and developer relations teams at Google. Stack Overflow (www.stackoverflow.com/questions/tagged/android) is also a hugely popular destination for Android questions and a great place to find answers to beginner questions.

For those of you using Eclipse, Android has released the Android Development Tools (ADT) plug-in that simplifies project creation and tightly integrates Eclipse with the Android emulator and the build and debugging tools. The features of the ADT plug-in are covered in more detail in Chapter 2.

Understanding the Android Software Stack

The Android software stack is, put simply, a Linux kernel and a collection of C/C++ libraries exposed through an application framework that provides services for, and management of, the run time and applications. The Android software stack is composed of the elements shown in Figure 1.1.

Figure 1.1



- **Linux kernel**—Core services (including hardware

drivers, process and memory management, security, network, and power management) are handled by a Linux 2.6 kernel. The kernel also provides an abstraction layer between the hardware and the remainder of the stack.

- **Libraries**—Running on top of the kernel, Android includes various C/C++ core libraries such as libc and SSL, as well as the following:
    - A media library for playback of audio and video media
    - A surface manager to provide display management
    - Graphics libraries that include SGL and OpenGL for 2D and 3D graphics
    - SQLite for native database support
    - SSL and WebKit for integrated web browser and Internet security
- **Android run time**—The run time is what makes an Android phone an Android phone rather than a mobile Linux implementation. Including the core libraries and the Dalvik VM, the Android run time is the engine that powers your applications and, along with the libraries, forms the basis for the application framework.
    - **Core libraries**—Although most Android application development is written using the Java language, Dalvik is not a Java VM. The core Android libraries provide most of the functionality available in the core Java libraries, as well as the Android-specific libraries.
    - **Dalvik VM**—Dalvik is a register-based Virtual Machine that's been optimized to ensure that a device can run multiple instances efficiently. It relies on the Linux kernel for threading and low-level memory management.
- **Application framework**—The application framework provides the classes used to create Android applications. It also provides a generic abstraction for hardware access and manages the user interface and application resources.
- **Application layer**—All applications, both native and third-party, are built on the application layer by means of the same API libraries. The application layer runs within the Android run time, using the classes and services made available from the application framework.

The Dalvik Virtual Machine

One of the key elements of Android is the Dalvik VM. Rather than using a traditional Java VM such as Java ME, Android uses its own custom VM designed to ensure that multiple instances run efficiently on a single device.

The Dalvik VM uses the device's underlying Linux kernel to handle low-level functionality, including security, threading, and process and memory management. It's also possible to write C/C++ applications that run closer to the underlying Linux OS. Although you *can* do this, in most cases there's no reason you should need to.

If the speed and efficiency of C/C++ is required for your application, Android provides a native development kit (NDK). The NDK is designed to enable you to create C++ libraries using the libc and libm libraries, along with native access to OpenGL.

This book focuses exclusively on writing applications that run within Dalvik using the SDK; NDK development is not within the scope of this book. If your inclinations run toward NDK development, exploring the Linux kernel and *C/C++* underbelly of Android, modifying Dalvik, or otherwise tinkering with things under the hood, check out the Android Internals Google Group at http://groups.google.com/group/android-internals.

All Android hardware and system service access is

managed using Dalvik as a middle tier. By using a VM to host application execution, developers have an abstraction layer that ensures they should never have to worry about a particular hardware implementation.

The Dalvik VM executes Dalvik executable files, a format optimized to ensure minimal memory footprint. You create `.dex` executables by transforming Java language compiled classes using the tools supplied within the SDK.

You'll learn more about how to create Dalvik executables in Chapter 2.

Android Application Architecture

Android's architecture encourages component reuse, enabling you to publish and share Activities, Services, and data with other applications, with access managed by the security restrictions you define.

The same mechanism that enables you to produce a replacement contact manager or phone dialer can let you expose your application's components in order to let other developers build on them by creating new UI front ends or functionality extensions.

The following application services are the architectural cornerstones of all Android applications, providing the framework you'll be using for your own software:

- **Activity Manager and Fragment Manager**—Control the lifecycle of your Activities and Fragments, respectively, including management of the Activity stack (described in Chapters 3 and 4).
- **Views**—Used to construct the user interfaces for your Activities and Fragments, as described in Chapter 4.
- **Notification Manager**—Provides a consistent and nonintrusive mechanism for signaling your users, as described in Chapter 10.
- **Content Providers**—Lets your applications share data, as described in Chapter 8.

- **Resource Manager**—Enables non-code resources, such as strings and graphics, to be externalized, as shown in Chapter 3.
- **Intents**—Provides a mechanism for transferring data between applications and their components, as described in Chapter 5.

Android Libraries

Android offers a number of APIs for developing your applications. Rather than list them all here, check out the documentation at http://developer.android.com/reference/packages.html, which gives a complete list of packages included in the Android SDK.

Android is intended to target a wide range of mobile hardware, so be aware that the suitability and implementation of some of the advanced or optional APIs may vary depending on the host device.

Chapter 2

Getting Started

What's in this Chapter?

- Installing the Android SDK, creating a development environment, and debugging your projects
- Understanding mobile design considerations
- The importance of optimizing for speed and efficiency
- Designing for small screens and mobile data connections
- Using Android Virtual Devices, the Emulator, and other development tools

All you need to start writing your own Android applications is a copy of the Android SDK and the Java Development Kit (JDK). Unless you're a masochist, you'll probably want a Java integrated development environment (IDE)—Eclipse is particularly well supported—to make development a little bit less painful.

The Android SDK, the JDK, and Eclipse are each available for Windows, Mac OS, and Linux, so you can

explore Android from the comfort of whatever operating system you favor. The SDK tools and Emulator work on all three OS environments, and because Android applications are run on a Dalvik virtual machine (VM), there's no advantage to developing on any particular OS.

Android code is written using Java syntax, and the core Android libraries include most of the features from the core Java APIs. Before you can run your projects, you must translate them into Dalvik bytecode. As a result, you get the familiarity of Java syntax while your applications gain the advantage of running on a VM optimized for mobile devices.

The Android SDK starter package contains the SDK platform tools, including the SDK Manager, which is necessary to download and install the rest of the SDK packages.

The Android SDK Manager is used to download Android framework SDK libraries, optional add-ons (including the Google APIs and the support library), complete documentation, and a series of excellent sample applications. It also includes the platform and development tools you will use to write and debug your applications, such as the Android Emulator to run your projects and the Dalvik Debug Monitoring Service (DDMS) to help debug them.

By the end of this chapter, you'll have downloaded the Android SDK starter package and used it to install the SDK and its add-ons, the platform tools, documentation, and sample code. You'll set up your development

environment, build your first Hello World application, and run and debug it using the DDMS and the Emulator running on an Android Virtual Device (AVD).

If you've developed for mobile devices before, you already know that their small-form factor, limited battery life, and restricted processing power and memory create some unique design challenges. Even if you're new to the game, it's obvious that some of the things you can take for granted on the desktop or the Web aren't going to work on mobile or embedded devices.

The user environment brings its own challenges, in addition to those introduced by hardware limitations. Many Android devices are used on the move and are often a distraction rather than the focus of attention, so your application needs to be fast, responsive, and easy to learn. Even if your application is designed for devices more conducive to an immersive experience, such as tablets or televisions, the same design principles can be critical for delivering a high-quality user experience.

This chapter examines some of the best practices for writing Android applications that overcome the inherent hardware and environmental challenges associated with mobile development. Rather than try to tackle the whole topic, we'll focus on using the Android SDK in a way that's consistent with good design principles.

Developing for Android

The Android SDK includes all the tools and APIs you need to write compelling and powerful mobile applications. The biggest challenge with Android, as with any new development toolkit, is learning the features and limitations of its APIs.

If you have experience in Java development, you'll find that the techniques, syntax, and grammar you've been using will translate directly into Android, although some of the specific optimization techniques may seem counterintuitive.

If you don't have experience with Java but have used other object-oriented languages (such as C#), you should find the transition straightforward. The power of Android comes from its APIs, not the language being used, so being unfamiliar with some of the Java-specific classes won't be a big disadvantage.

What You Need to Begin

Because Android applications run within the Dalvik VM, you can write them on any platform that supports the developer tools. This currently includes the following:

- Microsoft Windows (XP or later)
- Mac OS X 10.5.8 or later (Intel chips only)
- Linux (including GNU C Library 2.7 or later)

To get started, you'll need to download and install the following:

- The Android SDK starter package
- Java Development Kit (JDK) 5 or 6

You can download the latest JDK from Sun at http://java.sun.com/javase/downloads/index.jsp.

If you already have a JDK installed, make sure that it meets the preceding requirements, and note that the Java Runtime Environment (JRE) is not sufficient.

In most circumstances, you'll also want to install an IDE. The following sections describe how to install the Android SDK and use Eclipse as your Android IDE.

Downloading and Installing the Android SDK

There's no cost to download or use the API, and Google doesn't require your application to pass a review to distribute your finished programs on the Google Play Store. Although the Google Play Store requires a small one-time fee to publish applications, if you chose not to distribute via the Google Play Store, you can do so at no cost.

You can download the latest version of the SDK starter package for your chosen development platform from the Android development home page at http://developer.android.com/sdk/index.html.

Unless otherwise noted, the version of the Android SDK used for writing this book was version 4.0.3 (API level 15).
As an open-source platform, the Android SDK source is also available for you to download and compile from http://source.android.com.

The starter package is a ZIP file that contains the latest version of the Android tools required to download the rest of the Android SDK packages. Install it by unzipping the SDK into a new folder. Take note of this location, as you'll need it later.

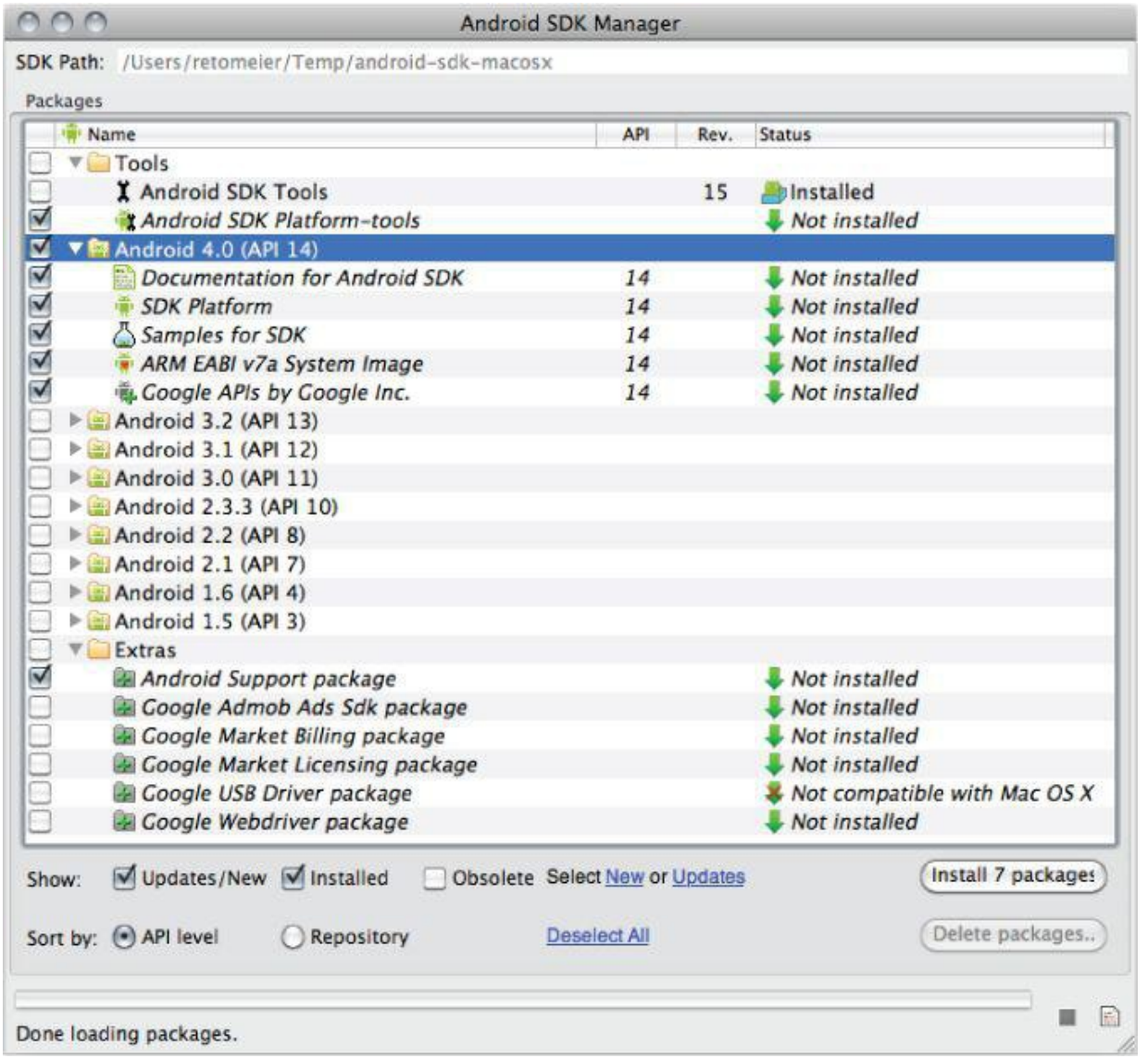
If you are developing from a Windows platform, an

executable Windows installer is available (and recommended) as an alternative to the ZIP file for installing the platform tools.

Before you can begin development, you need to download at least one SDK platform release. You can do this on Windows by running the SDK Manager.exe executable, or on Mac OS or Linux by running the "android" executable in the tools subfolder of the starter package download.

The screen that appears (see Figure 2.1) shows each of the packages available for the download. This includes a node for the platform tools, each of the platform releases, and a collection of extras, such as the Android Support Package and billing/licensing packages.

Figure 2.1



You can expand each platform release node to see a list of the packages included within it, including the tools, documentation, and sample code packages.

To get started, simply check the boxes corresponding to the newest framework SDK and the latest version of the tools, compatibility/support library, documentation, and sample code.

For testing the backward compatibility of your applications, it can often be useful to download the framework SDK for each version you intend to support.

To use the Google APIs (which contain the Maps APIs), you also need to select the Google APIs by Google package from the platform releases you want to support.

When you click the Install Packages button, the packages you've chosen will be downloaded to your SDK installation folder. The result is a collection of framework API libraries, documentation, and several sample applications.

The examples included in the SDK are well documented and are an excellent source for full, working examples of applications written for Android. When you finish setting up your development environment, it's worth going through them.

Downloading and Installing Updates to the SDK

As new versions of the Android framework SDK, developer tools, sample code, documentation, compatibility library, and third-party add-ons become available, you can use the Android SDK Manager to download and install those updates.

All future packages and upgrades will be placed in the same SDK location.

Developing with Eclipse

The examples and step-by-step instructions in this book are targeted at developers using Eclipse with the Android Developer Tools (ADT) plug-in. Neither is required, though; you can use any text editor or Java IDE you're comfortable with and use the developer tools in the SDK to compile, test, and debug the code snippets and sample applications.

As the recommended development platform, using Eclipse with the ADT plug-in for your Android development offers some significant advantages, primarily through the tight integration of many of the Android build and debug tools into your IDE.

Eclipse is a particularly popular open-source IDE for Java development. It's available for download for each of the development platforms supported by Android (Windows, Mac OS, and Linux) from the Eclipse foundation (www.eclipse.org/downloads).

Many variations of Eclipse are available, with Eclipse 3.5 (Galileo) or above required to use the ADT plugin. The following is the configuration for Android used in the preparation of this book:

- Eclipse 3.7 (Indigo) (Eclipse Classic download)
  - Eclipse Java Development Tools (JDT) plug-in
  - Web Standard Tools (WST)

The JDT plug-in and WST are included in most Eclipse IDE packages.

Installing Eclipse consists of uncompressing the download into a new folder, and then running the `eclipse` executable. When it starts for the first time, you should create a new workspace for your Android development projects.

Using the Android Developer Tools Plug-In for Eclipse

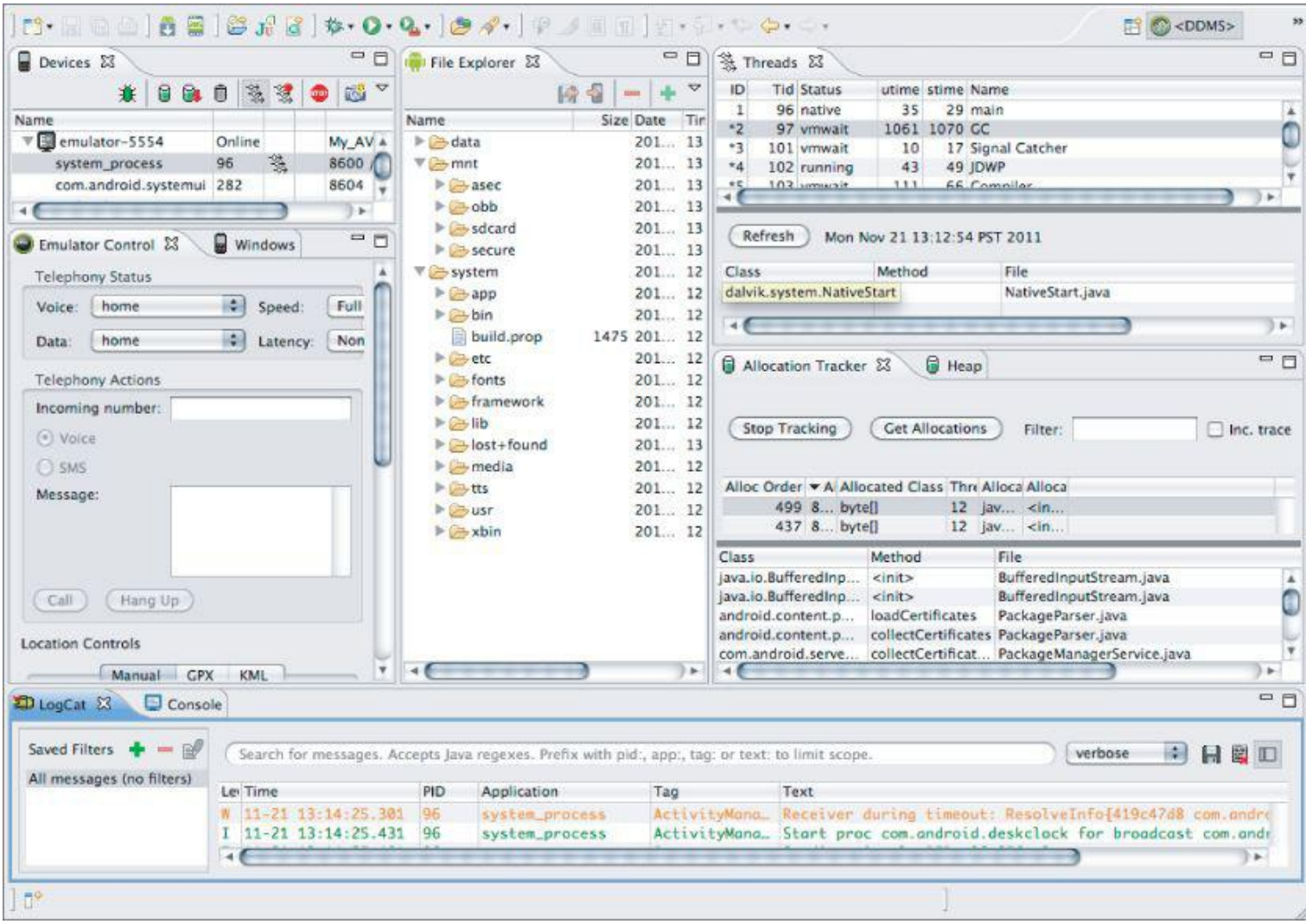
The ADT plug-in for Eclipse simplifies your Android development by integrating the developer tools, including the Emulator and .class-to-.dex converter, directly into the IDE. Although you don't have to use the ADT plug-in, it can make creating, testing, and debugging your applications faster and easier.

The ADT plug-in integrates the following into Eclipse:

- An Android Project Wizard, which simplifies creating new projects and includes a basic application template
- Forms-based manifest, layout, and resource editors to help create, edit, and validate your XML resources
- Automated building of Android projects, conversion to Android executables (`.dex`), packaging to package files (`.apk`), and installation of packages onto Dalvik VMs (running both within the Emulator or on physical devices)
- The Android Virtual Device manager, which lets you create and manage virtual devices to host Emulators that run a specific release of the Android OS and with set hardware and memory constraints
- The Android Emulator, including the ability to control the Emulator's appearance and network connection settings, and the ability to simulate incoming calls and SMS messages
- The Dalvik Debug Monitoring Service (DDMS), which includes port forwarding, stack, heap, and thread viewing, process details, and screen-capture facilities
- Access to the device or Emulator's filesystem, enabling you to navigate the folder tree and transfer files
- Runtime debugging, which enables you to set breakpoints and view call stacks
- All Android/Dalvik log and console outputs

Figure 2.2 shows the DDMS perspective within Eclipse with the ADT plug-in installed.

Figure 2.2



Installing the ADT Plug-In

Install the ADT plug-in by following these steps:

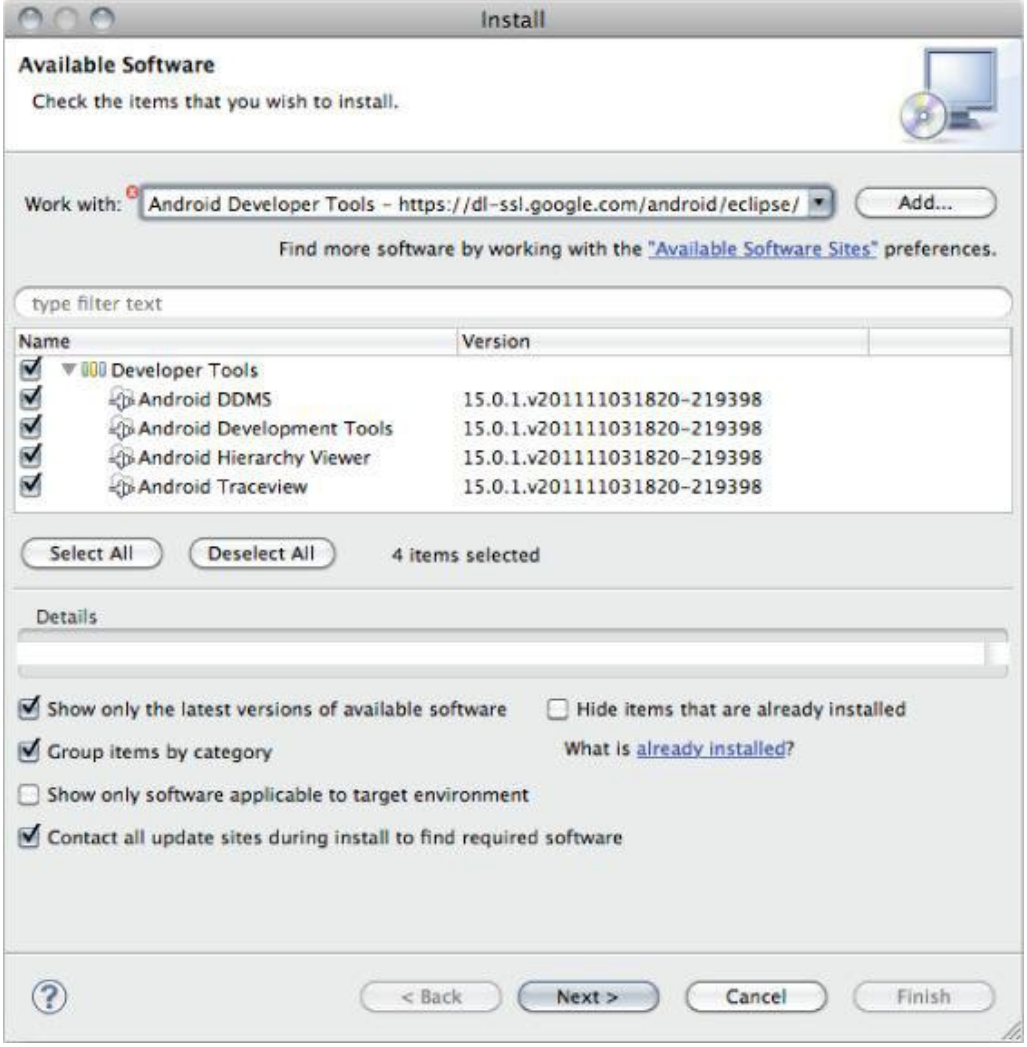
1. Select → Install New Software from within Eclipse.

2. In the Available Software dialog box that appears, click the Add button.

3. In the next dialog, enter a name you will remember (e.g., **Android Developer Tools**) into the Name field, and paste the following address into the Location text entry box: https://dl-ssl.google.com/android/eclipse/.

4. Press OK and Eclipse searches for the ADT plug-in. When finished, it displays the available plug-ins, as shown in Figure 2.3. Select it by clicking the check box next to the Developer Tools root node, and then click Next.

Figure 2.3



5. Eclipse now downloads the plug-in. When it finishes, a list of the Developer Tools displays for your review. Click Next.

6. Read and accept the terms of the license agreement, and click Next and then Finish. As the ADT plug-in is not signed, you'll be prompted before the installation continues.

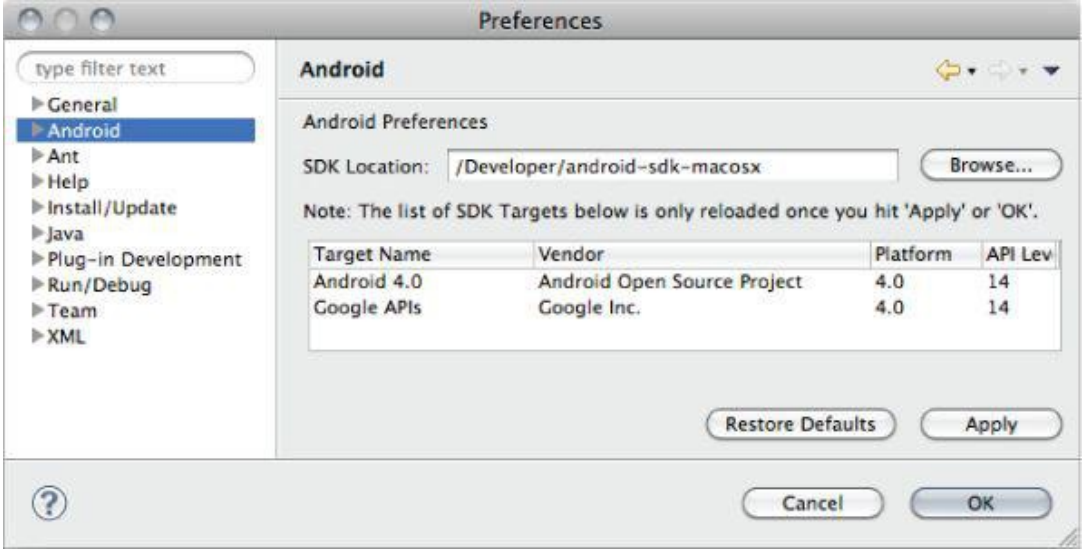
7. When installation is complete, you need to restart Eclipse and update the ADT preferences. Restart and select Window → Preferences (or → Preferences for Mac OS).

8. Select Android from the left panel.

9. Click Browse, navigate to the folder into which you installed the Android SDK, and then click Apply. The list updates to display each available SDK target, as shown in Figure 2.4. Click OK to complete the SDK installation.

If you move your SDK installation to a different location, you will need to update the ADT preference, as described in steps 7 to 9 above, to reflect the new path to the SDK against which the ADT should be building.

Figure 2.4



Updating the ADT Plug-In

In most cases, you can update your ADT plug-in simply as follows:

1. Navigate to Help → Check for Updates.

2. If there are any ADT updates available, they will be presented. Simply select them and choose Install.

Sometimes a plug-in upgrade may be so significant that the dynamic update mechanism can't be used. In those cases you may have to remove the previous plug-in completely before installing the newer version, as described in the previous section.

Using the Support Package

The support library package (previously known as the compatibility library) is a set of static libraries that you can include as part of your projects to gain either convenience APIs that aren't packaged as part of the framework (such as the View Pager), or useful APIs that are not available on all platform releases (such as Fragments).

The support package enables you to use framework API features that were introduced in recent Android platform releases on any device running Android 1.6 (API level 4) and above. This helps you provide a consistent user experience and greatly simplifies your development process by reducing the burden of supporting multiple platform versions.

It's good practice to use the support library rather than the framework API libraries when you want to support devices running earlier platform releases and where the support library offers all the functionality you require.
In the interest of simplicity, the examples in this book target Android API level 15 and use the framework APIs in preference to the support library, highlighting specifi c areas where the support library would not be a suitable alternative.

To incorporate a support library into your project, perform the following steps:

1. Add a new `/libs` folder in the root of your project hierarchy.

2. Copy the support library JAR file from the `/extras/android/support/` folder in your Android SDK installation location.

You'll note that the `support` folder includes multiple subfolders, each of which represents the minimum platform version supported by that library. Simply use the corresponding JAR file stored in the subfolder labeled as less than or equal to the minimum platform version you plan to support.

3. For example, if you want to support all platform versions from Android 1.6 (API level 4) and above, you would copy `v4/android-support-v4.jar`.

4. After copying the file into your project's `/libs` folder, add it to your project build path by right-clicking in the Package Explorer and selecting Build Path → Add to Build Path.

By design, the support library classes mirror the names of their framework counterparts. Some of these classes (such as `SimpleCursorAdapter`) have existed since early platform releases. As a result, there's a significant risk that the code completion and automatic import-management tools in Eclipse (and other IDEs) will select the wrong library—particularly when you're building against newer versions of the SDK.

It's good practice to set your project build target to the minimum platform version you plan to support, and to ensure the import statements are using the compatibility library for classes that also exist in the target framework.

Creating Your First Android Application

You've downloaded the SDK, installed Eclipse, and plugged in the plug-in. You are now ready to start programming for Android. Start by creating a new Android project and setting up your Eclipse *run* and *debug* configurations, as described in the following sections.

Creating a New Android Project

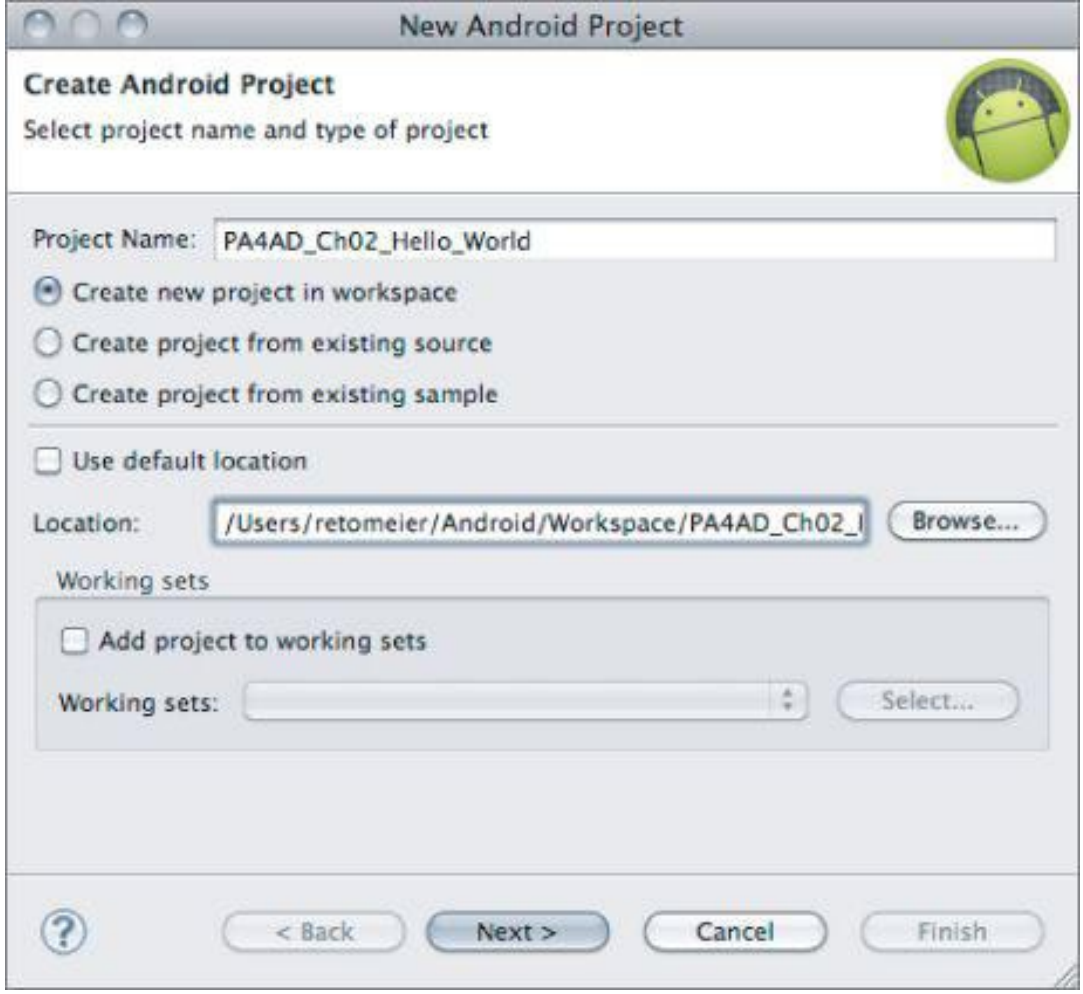
To create a new Android project using the Android New Project Wizard, do the following:

1. Select File → New → Project.

2. Select the Android Project application type from the Android folder, and click Next.

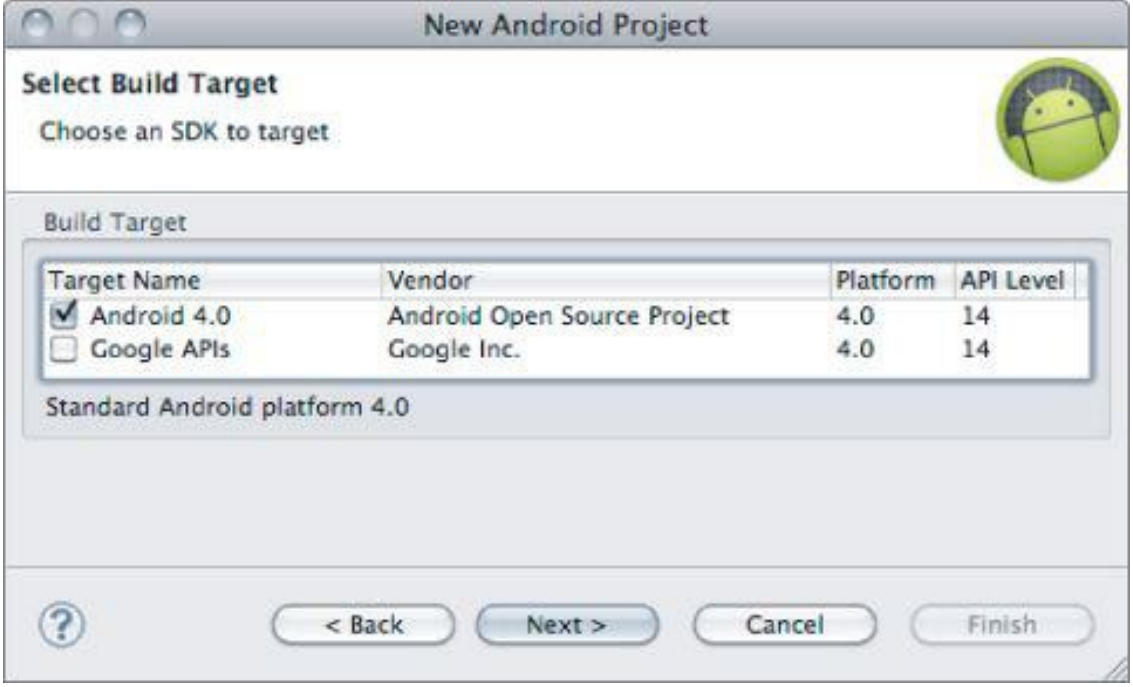
3. In the wizard that appears, enter the details for your new project. On the first page (Figure 2.5), the Project Name is the name of your project file. You can also select the location your project should be saved.

Figure 2.5



4. The next page (Figure 2.6) lets you select the build target for your application. The *build target* is the version of the Android framework SDK that you plan to develop with. In addition to the open sourced Android SDK libraries available as part of each platform release, Google offers a set of proprietary APIs that offer additional libraries (such as Maps). If you want to use these Google-specific APIs, you must select the Google APIs package corresponding to the platform release you want to target.

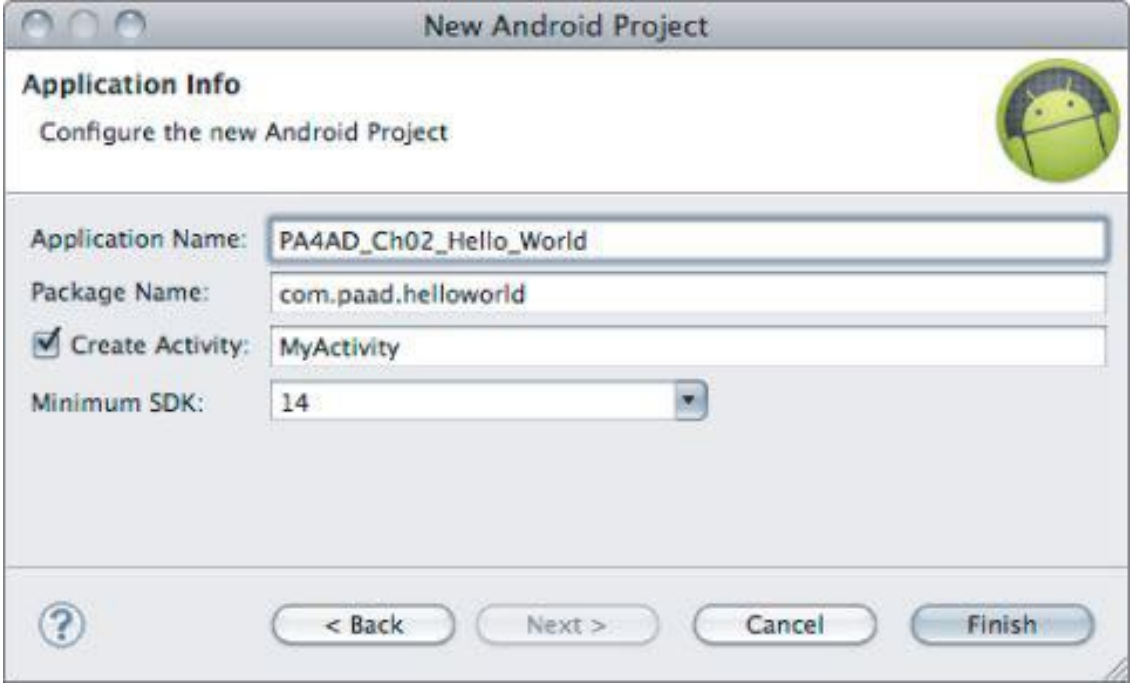
Figure 2.6



Your project's build target does not need to correspond to its minimum SDK or target SDK. For new projects it's good practice to build against the newest version of the SDK to take advantage of efficiency and UI improvements in newer platform releases.

5. The final page (Figure 2.7) allows you to specify the application properties. The Application Name is the friendly name for your application; the Package Name specifies its Java package; the Create Activity option lets you specify the name of a class that will be your initial Activity; and setting the Minimum SDK lets you specify the minimum version of the SDK that your application will run on.

Figure 2.7



Selecting the minimum SDK version requires you to choose the level of backward compatibility you want to support to target a wider group of Android devices. Your application will be available from the Google Play Store on any device running the specified build or higher.
At the time of this writing, more than 98% of Android devices were running at least Android 2.1 (API level 7). The latest Ice Cream Sandwich SDK is 4.0.3 (API level 15).

6. When you've entered the details, click Finish.

If you selected Create Activity, the ADT plug-in will create a new project that includes a class that extends `Activity`. Rather than being completely empty, the default template implements Hello World. Before modifying the project, take this opportunity to configure launch configurations for running and debugging.

Creating an Android Virtual Device

AVDs are used to simulate the hardware and software configurations of different Android devices, allowing you test your applications on a variety of hardware platforms.

There are no prebuilt AVDs in the Android SDK, so without a physical device, you need to create at least one before you can run and debug your applications.

1. Select Window → AVD Manager (or select the AVD Manager icon on the Eclipse toolbar).

2. Select the New… button.

3. The resulting Create new Android Virtual Device (AVD) dialog allows you to configure a name, a target build of Android, an SD card capacity, and device skin.

4. Create a new AVD called “My_AVD” that targets Android 4.0.3, includes a 16MB SD Card, and uses the Galaxy Nexus skin, as shown in Figure 2.8.

5. Click Create AVD and your new AVD will be created and ready to use.

Figure 2.8

Creating Launch Configurations

Launch configurations let you specify runtime options for running and debugging applications. Using a launch configuration you can specify the following:

- The Project and Activity to launch
- The deployment target (virtual or physical device)
- The Emulator's launch parameters
- Input/output settings (including console defaults)

You can specify different launch configurations for running and debugging applications. The following steps show how to create a launch configuration for an Android application:

1. Select Run Configurations… or Debug Configurations… from the Run menu.

2. Select your application from beneath the Android Application node on the project type list, or right-click the Android Application node and select New.

3. Enter a name for the configuration. You can create multiple configurations for each project, so create a descriptive title that will help you identify this particular setup.

4. Choose your start-up options. The first (Android) tab lets you select the project to run and the Activity that you want to start when you run (or debug) the application. Figure 2.9 shows the settings for the project you created earlier.

5. Use the Target tab, as shown in Figure 2.10, to select the default virtual device to launch on, or select Manual to select a physical or virtual device each time you run the application. You can also configure the Emulator's network connection settings and optionally wipe the user data and disable the boot animation when launching a virtual device.

The Android SDK does not include a default AVD, so you need to create one before you can run or debug your applications using the Emulator. If the Virtual Device selection list in Figure 2.10 is empty, click Manager... to open the Android Virtual Device Manager and create one as described in the previous section.
Further details on the Android Virtual Device Manager are available later in this chapter.

6. Set any additional properties in the Common tab.

7. Click Apply, and your launch configuration will be saved.

Figure 2.9

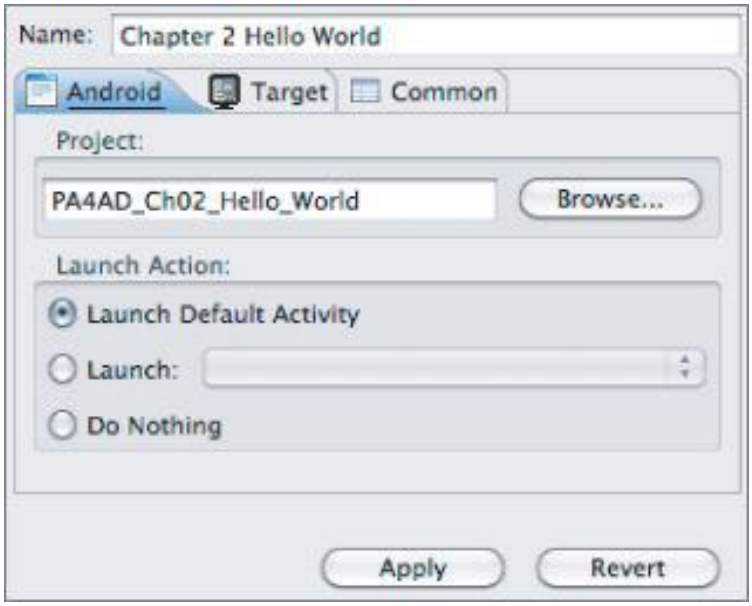
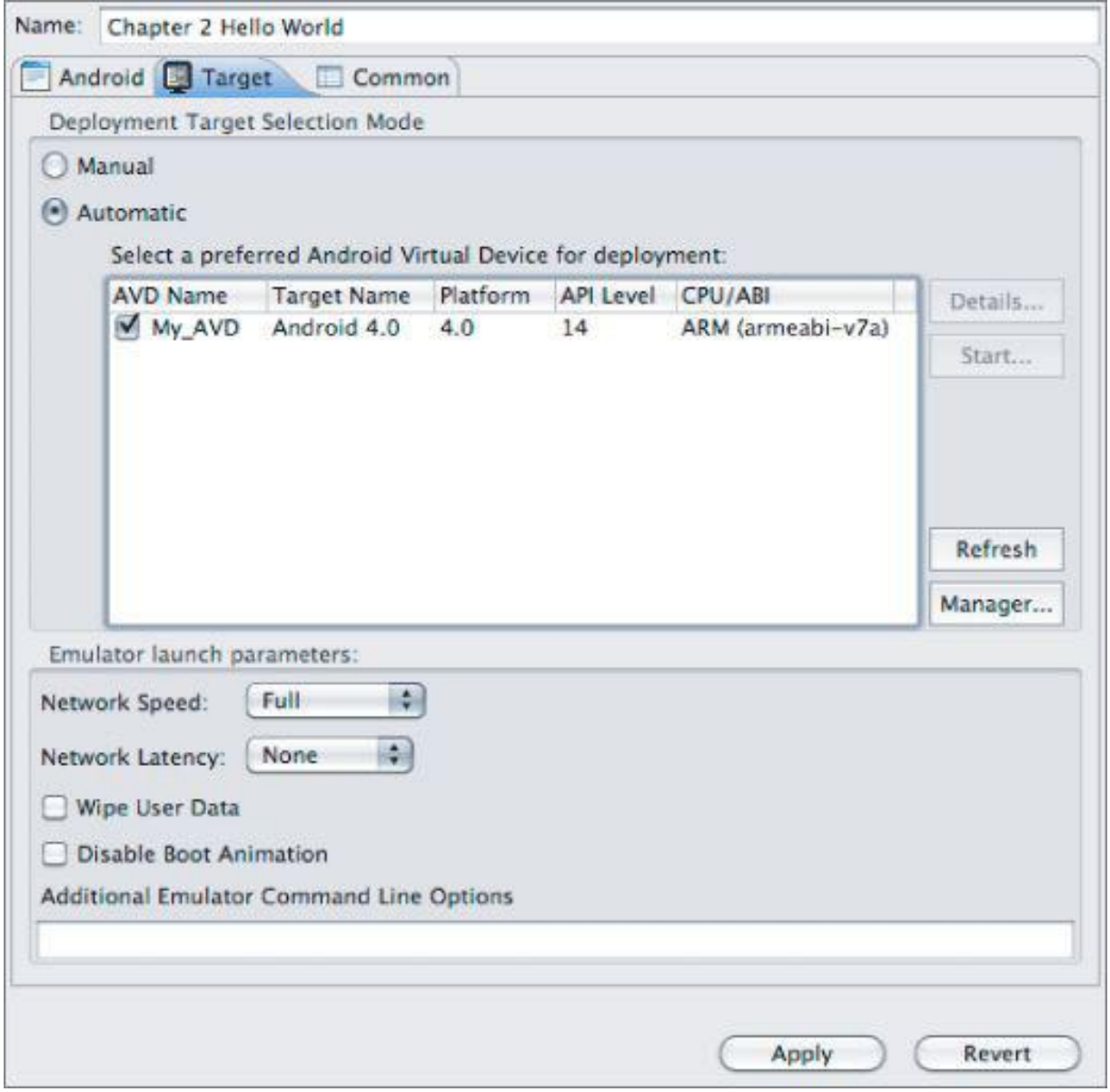


Figure 2.10



Running and Debugging Your Android Application

You've created your first project and created the run and debug configurations for it. Before making any changes, test your installation and configurations by running and debugging the Hello World project.

From the Run menu, select Run or Debug to launch the most recently selected configuration, or select Run Configurations… or Debug Configurations… to select a specific configuration.

If you're using the ADT plug-in, running or debugging your application does the following:

- Compiles the current project and converts it to an Android executable (`.dex`)
- Packages the executable and your project's resources into an Android package (`.apk`)
- Starts the virtual device (if you've targeted one and it's not already running)
- Installs your application onto the target device
- Starts your application

If you're debugging, the Eclipse debugger will then be attached, allowing you to set breakpoints and debug your code.

If everything is working correctly, you'll see a new Activity running on the device or in the Emulator, as shown in Figure 2.11.

Figure 2.11

Understanding Hello World

Take a step back and have a good look at your first Android application.

`Activity` is the base class for the visual, interactive components of your application; it is roughly equivalent to a Form in traditional desktop development (and is described in detail in Chapter 3, "Creating Applications and Activities"). Listing 2.1 shows the skeleton code for an Activity-based class; note that it extends `Activity` and overrides the `onCreate` method.

Listing 2.1: Hello World

```
package com.paad.helloworld;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {

  /** Called when the Activity is first created. **/
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);
  }
}
```

code snippet PA4AD_Ch02_HelloWorld/src/MyActivity.java

In Android, visual components are called *Views*, which are similar to controls in traditional desktop development. The Hello World template created by the wizard overrides the `onCreate` method to call `setContentView`, which lays out the UI by inflating a layout resource, as highlighted in bold in the following snippet:

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

The resources for an Android project are stored in the `res` folder of your project hierarchy, which includes `layout`, `values`, and a series of `drawable` subfolders. The ADT plug-in interprets these resources to provide design-time access to them through the `R` variable, as described in

Chapter 3.

Listing 2.2 shows the UI layout defined in the `main.xml` file created by the Android project template and stored in the project's `res/layout` folder.

Listing 2.2: Hello World layout resource

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
  />
</LinearLayout>
```

code snippet
PA4AD_Ch02_HelloWorld/res/layout/main.xml

Defining your UI in XML and inflating it is the preferred way of implementing your user interfaces (UIs), as it neatly decouples your application logic from your UI design.

To get access to your UI elements in code, you add identifier attributes to them in the XML definition. You can then use the `findViewById` method to return a reference to each named item. The following XML snippet shows an ID attribute added to the Text View widget in the Hello World template:

```
<TextView
  android:id="@+id/myTextView"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:text="@string/hello"
/>
```

And the following snippet shows how to get access to it in code:

```
TextView myTextView =
(TextView)findViewById(R.id.myTextView);
```

Alternatively (although it's not generally considered good practice), you can create your layout directly in code, as shown in Listing 2.3.

Listing 2.3: Creating layouts in code

```
public void onCreate(Bundle
savedInstanceState) {
  super.onCreate(savedInstanceState);

  LinearLayout.LayoutParams lp;
  lp = new
LinearLayout.LayoutParams(LinearLayout.LayoutParams.FILL_PARENT,

LinearLayout.LayoutParams.FILL_PARENT);

  LinearLayout.LayoutParams
textViewLP;
  textViewLP = new
LinearLayout.LayoutParams(

 LinearLayout.LayoutParams.FILL_PARENT,

 LinearLayout.LayoutParams.WRAP_CONTENT);

  LinearLayout ll = new
LinearLayout(this);
  ll.setOrientation(LinearLayout.VERTICAL);

  TextView myTextView = new
TextView(this);
  myTextView.setText(getString(R.string.hello));

  ll.addView(myTextView,
textViewLP);
  this.addContentView(ll, lp);
}
```

code snippet PA4AD_Ch02_Manual_Layout/src/MyActivity.java

All the properties available in code can be set with attributes in the XML layout.

More generally, keeping the visual design decoupled from the application code helps keep the code concise. With Android available on hundreds of different devices of varying screen sizes, defining your layouts as XML resources makes it easier for you to include multiple layouts optimized for different screens.

You'll learn how to build your user interface by creating layouts and building your own custom Views in Chapter 4, “Building User Interfaces.”

Types of Android Applications

Most of the applications you create in Android will fall into one of the following categories:

- **Foreground**—An application that's useful only when it's in the foreground and is effectively suspended when it's not visible. Games are the most common examples.
- **Background**—An application with limited interaction that, apart from when being configured, spends most of its lifetime hidden. These applications are less common, but good examples include call screening applications, SMS auto-responders, and alarm clocks.
- **Intermittent**—Most well-designed applications fall into this category. At one extreme are applications that expect limited interactivity but do most of their work in the background. A common example would be a media player. At the other extreme are applications that are typically used as foreground applications but that do important work in the background. Email and news applications are great examples.
- **Widgets and Live Wallpapers**—Some applications are represented only as a home-screen Widget or as

a Live Wallpaper.

Complex applications are often difficult to pigeonhole into a single category and usually include elements of each of these types. When creating your application, you need to consider how it's likely to be used and then design it accordingly. The following sections look more closely at some of the design considerations for each application type.

Foreground Applications

When creating foreground applications, you need to consider carefully the Activity lifecycle (described in Chapter 3) so that the Activity switches seamlessly between the background and the foreground.

Applications have little control over their lifecycles, and a background application with no running Services is a prime candidate for cleanup by Android's resource management. This means that you need to save the state of the application when it leaves the foreground, and then present the same state when it returns to the front.

It's also particularly important for foreground applications to present a slick and intuitive user experience. You'll learn more about creating well-behaved and attractive foreground Activities in Chapters 3, 4, 10, and 11.

Background Applications

These applications run silently in the background with little user input. They often listen for messages or actions caused by the hardware, system, or other applications, rather than relying on user interaction.

You can create completely invisible services, but in practice it's better to provide at least a basic level of user control. At a minimum you should let users confirm that the service is running and let them configure, pause, or terminate it, as needed.

Services and Broadcast Receivers, the driving forces of background applications, are covered in depth in Chapter 5, “Intents and Broadcast Receivers,” and Chapter 9, “Working in the Background.”

Intermittent Applications

Often you'll want to create an application that can accept user input and that also reacts to events when it's not the active foreground Activity. Chat and e-mail applications are typical examples. These applications are generally a union of visible Activities and invisible background Services and Broadcast Receivers.

Such an application needs to be aware of its state when interacting with the user. This might mean updating the

Activity UI when it's visible and sending notifications to keep the user updated when it's in the background, as described in the section “Using Notifications” in Chapter 10.

You must be particularly careful to ensure that the background processes of applications of this type are well behaved and have a minimal impact on the device's battery life.

Widgets and Live Wallpapers

In some circumstances your application may consist entirely of a Widget or Live Wallpaper. By creating Widgets and Live Wallpapers, you provide interactive visual components that can add functionality to user's home screens.

Widget-only applications are commonly used to display dynamic information, such as battery levels, weather forecasts, or the date and time.

You'll learn how to create Widgets and Live Wallpapers in Chapter 14, “Invading the Home Screen.”

Developing for Mobile and Embedded Devices

Android does a lot to simplify mobile- or embedded-device software development, but you need to understand the reasons behind the conventions. There are several factors to account for when writing software for mobile and embedded devices, and when developing for Android in particular.

In this chapter you'll learn some of the techniques and best practices for writing efficient Android code. In later examples, efficiency is sometimes compromised for clarity and brevity when new Android concepts or functionality are introduced. In the best tradition of “Do as I say, not as I do,” the examples are designed to show the simplest (or easiest-to-understand) way of doing something, not necessarily the best way of doing it.

Hardware-Imposed Design Considerations

Small and portable, mobile devices offer exciting opportunities for software development. Their limited screen size and reduced memory, storage, and processor power are far less exciting, and instead present some unique challenges.

Compared to desktop or notebook computers, mobile devices have relatively:

- Low processing power
- Limited RAM
- Limited permanent storage capacity
- Small screens with low resolution
- High costs associated with data transfer
- Intermittent connectivity, slow data transfer rates, and high latency
- Unreliable data connections
- Limited battery life

Each new generation of phones improves many of these restrictions. In particular, newer phones have dramatically improved screen resolutions and significantly cheaper data costs.

The introduction of tablet devices and Android-powered televisions has expanded the range of devices on which

your application may be running and eliminating some of these restrictions. However, given the range of devices available, it's always good practice to design to accommodate the worst-case scenario to ensure your application provides a great user experience no matter what the hardware platform it's installed on.

Be Efficient

Manufacturers of embedded devices, particularly mobile devices, generally value small size and long battery life over potential improvements in processor speed. For developers, that means losing the head start traditionally afforded thanks to Moore's law (the doubling of the number of transistors placed on an integrated circuit every two years). In desktop and server hardware, this usually results directly in processor performance improvements; for mobile devices, it instead means thinner, more power-efficient mobiles, with brighter, higher resolution screens. By comparison, improvements in processor power take a back seat.

In practice, this means that you always need to optimize your code so that it runs quickly and responsively, assuming that hardware improvements over the lifetime of your software are unlikely to do you any favors.

Code efficiency is a big topic in software engineering, so I'm not going to try and cover it extensively here. Later in this chapter you'll learn some Android-specific efficiency

tips, but for now note that efficiency is particularly important for resource-constrained platforms.

Expect Limited Capacity

Advances in flash memory and solid-state disks have led to a dramatic increase in mobile-device storage capacities. (MP3 collections still tend to expand to fill the available storage.) Although an 8GB flash drive or SD card is no longer uncommon in mobile devices, optical disks offer more than 32GB, and terabyte drives are now commonly available for PCs. Given that most of the available storage on a mobile device is likely to be used to store music and movies, many devices offer relatively limited storage space for your applications.

Android lets you specify that your application can be installed on the SD card as an alternative to using internal memory (described in detail in Chapter 3), but there are significant restrictions to this approach and it isn't suitable for all applications. As a result, the compiled size of your application is an important consideration, though more important is ensuring that your application is polite in its use of system resources.

You should carefully consider how you store your application data. To make life easier, you can use the Android databases and Content Providers to persist, reuse, and share large quantities of data, as described in Chapter 8, “Databases and Content Providers.” For

smaller data storage, such as preferences or state settings, Android provides an optimized framework, as described in Chapter 7, “Files, Saving State, and Preferences.”

Of course, these mechanisms won't stop you from writing directly to the filesystem when you want or need to, but in those circumstances always consider how you're structuring these files, and ensure that yours is an efficient solution.

Part of being polite is cleaning up after yourself. Techniques such as caching, pre-fetching, and lazy loading are useful for limiting repetitive network lookups and improving application responsiveness, but don't leave files on the filesystem or records in a database when they're no longer needed.

Design for Different Screens

The small size and portability of mobiles are a challenge for creating good interfaces, particularly when users are demanding an increasingly striking and information-rich graphical user experience. Combined with the wide range of screen sizes that make up the Android device ecosystem, creating consistent, intuitive, and pleasing user interfaces can be a significant challenge.

Write your applications knowing that users will often only glance at the screen. Make your applications intuitive and easy to use by reducing the number of controls and putting the most important information front and center.

Graphical controls, such as the ones you'll create in Chapter 4, are an excellent means of displaying a lot of information in a way that's easy to understand. Rather than a screen full of text with a lot of buttons and text-entry boxes, use colors, shapes, and graphics to convey information.

You'll also need to consider how touch input is going to affect your interface design. The time of the stylus has passed; now it's all about finger input, so make sure your Views are big enough to support interaction using a finger on the screen. To support accessibility and non-touch screen devices such as Google TV, you need to ensure your application is navigable without relying purely on touch.

Android devices are now available with a variety of screen sizes, from small-screen QVGA phones to 10.1” tablets and 46” Google TVs. As display technology advances and new Android devices are released, screen sizes and resolutions will be increasingly varied. To ensure that your application looks good and behaves well on all the possible host devices, you need to design and test your application on a variety of screens, optimizing for small screens and tablets, but also ensuring that your UIs scale well on any display.

You'll learn some techniques for optimizing your UI for different screen sizes in Chapters 3 and 4.

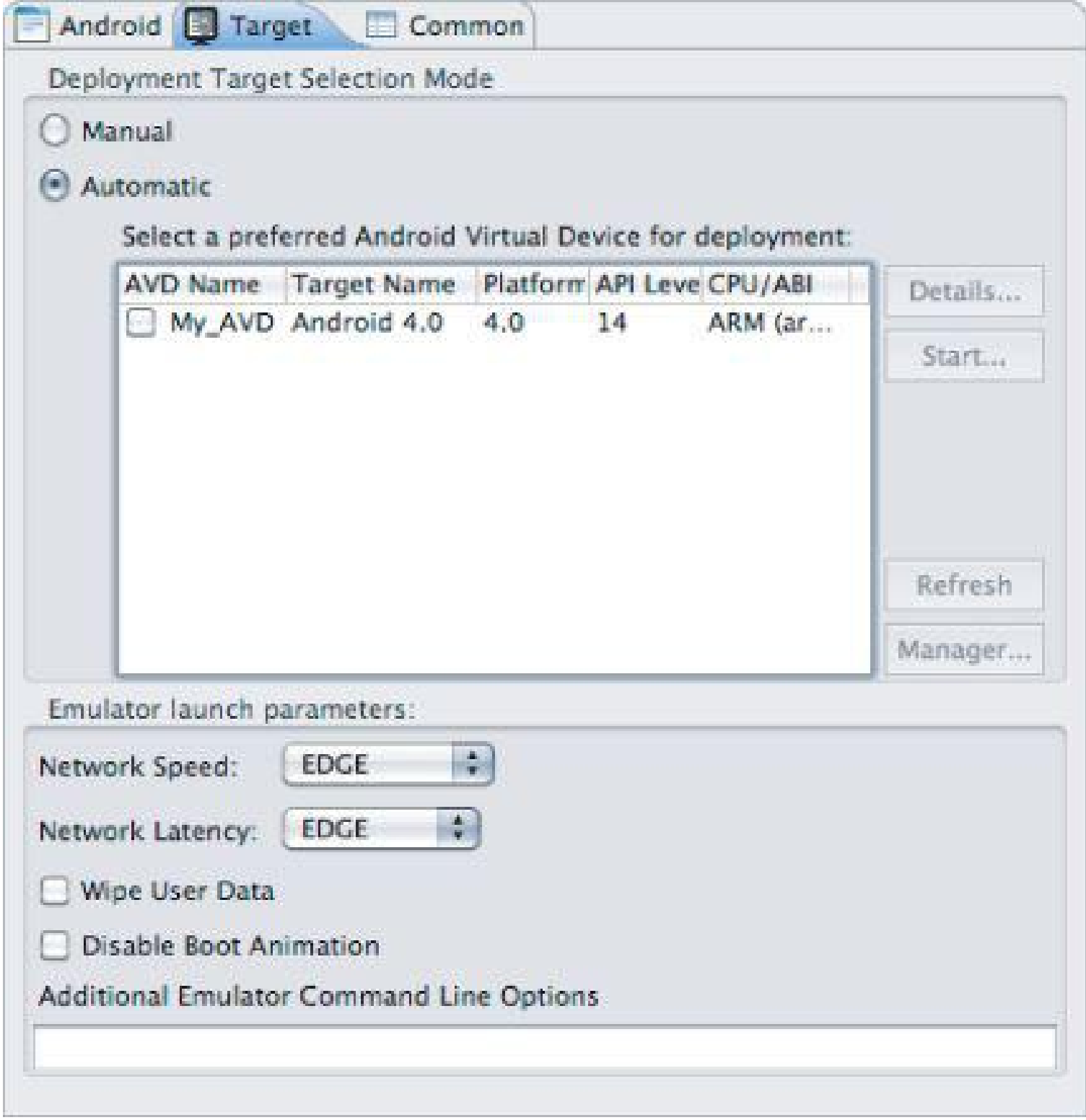
Expect Low Speeds, High Latency

The ability to incorporate some of the wealth of online information within your applications is incredibly powerful. Unfortunately, the mobile Web isn't as fast, reliable, or readily available as we would like; so, when you're developing your Internet-based applications, it's best to assume that the network connection will be slow, intermittent, and expensive.

With unlimited 4G data plans and citywide Wi-Fi, this is changing, but designing for the worst case ensures that you always deliver a high-standard user experience. This also means making sure that your applications can handle losing (or not finding) a data connection.

The Android Emulator enables you to control the speed and latency of your network connection. Figure 2.12 shows the Emulator's network connection speed and latency, simulating a distinctly suboptimal EDGE connection.

Figure 2.12



Experiment to ensure seamlessness and responsiveness no matter what the speed, latency, and availability of network access. Some techniques include limiting the functionality of your application, or reducing network lookups to cached bursts, when the available network connection supports only limited data transfer

capabilities.

In Chapter 6, “Using Internet Resources,” you'll learn how to use Internet resources in your applications.
Further details, including how to detect the kind of network connections available at run time, are included in Chapter 16, “Bluetooth, NFC, Networks, and Wi-Fi.”

At What Cost?

If you're a mobile device owner, you know all too well that some of your device's functionality can literally come at a price. Services such as SMS and data transfer can incur additional fees from your service provider.

It's obvious why any costs associated with functionality in your applications should be minimized, and that users should be made aware when an action they perform might result in their being charged.

It's a good approach to assume that there's a cost associated with any action involving an interaction with the outside world. In some cases (such as with GPS and data transfer), the user can toggle Android settings to disable a potentially costly action. As a developer, it's important that you use and respect those settings within your application.

In any case, it's important to minimize interaction costs by doing the following:

- Transferring as little data as possible
- Caching data and geocoding results to eliminate redundant or repetitive lookups
- Stopping all data transfers and GPS updates when your Activity is not visible in the foreground (provided they're only used to update the UI)
- Keeping the refresh/update rates for data transfers (and location lookups) as low as practicable
- Scheduling big updates or transfers at off-peak times or when connected via Wi-Fi by using Alarms and Broadcast Receivers, as shown in Chapter 9
- Respecting the user's preferences for background data transfers

Often the best solution is to use a lower-quality option that comes at a lower cost.

When using location-based services, as described in Chapter 13, “Maps, Geocoding, and Location-Based Services,” you can select a location provider based on whether there is an associated cost. Within your location-based applications, consider giving users the choice of lower cost or greater accuracy.

In some circumstances costs are either hard to define or different for different users. Charges for services vary between service providers and contract plans. Although some people will have free unlimited data transfers, others will have free SMS.

Rather than enforcing a particular technique based on which seems cheaper, consider letting your users choose.

For example, when users are downloading data from the Internet, ask them if they want to use any network available or limit their transfers to times when they're connected via Wi-Fi.

Considering the User's Environment

You can't assume that your users will think of your application as the most important feature of their device.

Although Android has already expanded beyond its roots as a mobile phone platform, most Android devices are phones or tablet devices. For most people, such a device is first and foremost a phone, secondly an SMS and email communicator, thirdly a camera, and fourthly an MP3 player. The applications you write will most likely be in the fifth category of “useful stuff.”

That's not a bad thing—they'll be in good company with others, including Google Maps and the web browser. That said, each user's usage model will be different; some people will never use their device to listen to music, some devices don't support telephony, and some don't include cameras — but the multitasking principle inherent in a device as ubiquitous as it is indispensable is an important consideration for usability design.

It's also important to consider when and how your users will use your applications. People use their mobiles all the time—on the train, walking down the street, or even while driving their cars. You can't make people use their phones appropriately, but you can make sure that your applications don't distract them any more than necessary.

What does this mean in terms of software design? Make

sure that your application:

- **Is predictable and well behaved**—Start by ensuring that your Activities suspend when they're not in the foreground. Android fires event handlers when your Activity is paused or resumed, so you can pause UI updates and network lookups when your application isn't visible—there's no point updating your UI if no one can see it. If you need to continue updating or processing in the background, Android provides a Service class designed for this purpose, without the UI overheads.
- **Switches seamlessly from the background to the foreground**—With the multitasking nature of mobile devices, it's likely that your applications will regularly move into and out of the background. It's important that they "come to life" quickly and seamlessly. Android's nondeterministic process management means that if your application is in the background, there's every chance it will get killed to free resources. This should be invisible to the user. You can ensure seamlessness by saving the application state and queuing updates so that your users don't notice a difference between restarting and resuming your application. Switching back to it should be seamless, with users being shown the UI and application state they last saw.
- **Is polite**—Your application should never steal focus

or interrupt a user's current Activity. Instead, use Notifications (detailed in Chapter 10) to request your user's attention when your application isn't in the foreground. There are several ways to alert users—for example, incoming calls are announced by a ringtone and/or vibration; when you have unread messages, the LED flashes; and when you have new voice mail, a small unread mail icon appears in the status bar. All these techniques and more are available to your application using the Notifications mechanism.

- **Presents an attractive and intuitive UI**—Your application is likely to be one of several in use at any time, so it's important that the UI you present is easy to use. Spend the time and resources necessary to produce a UI that is as attractive as it is functional, and don't force users to interpret and relearn your application every time they load it. Using it should be simple, easy, and obvious—particularly given the limited screen space and distracting user environment.
- **Is responsive**—Responsiveness is one of the most critical design considerations on a mobile device. You've no doubt experienced the frustration of a “frozen” piece of software; the multifunctional nature of a mobile makes this even more annoying. With the possibility of delays caused by slow and unreliable data connections, it's important that your application use worker threads and background Services to

keep your Activities responsive and, more important, to stop them from preventing other applications from responding promptly.

Developing for Android

Nothing covered so far is specific to Android; the preceding design considerations are just as important in developing applications for any mobile device. In addition to these general guidelines, Android has some particular considerations.

Take a few minutes to read the design best practices included in Google's Android Dev Guide at http://developer.android.com/guide/index.html.

The Android design philosophy demands that applications be designed for:

- Performance
- Responsiveness
- Freshness
- Security
- Seamlessness
- Accessibility

Being Fast and Efficient

In a resource-constrained environment, being fast means being efficient. A lot of what you already know about writing efficient code will be applicable to Android, but the limitations of embedded systems and the use of the Dalvik VM mean you can't take things for granted.

The smart bet for advice is to go to the source. The Android team has published some specific guidance on

writing efficient code for Android, so rather than reading a rehash of its advice, visit http://developer.android.com/guide/practices/design/performance.html for suggestions.

You may find that some of these performance suggestions contradict established design practices—for example, avoiding the use of internal setters and getters or preferring virtual classes over using interfaces. When writing software for resource-constrained systems such as embedded devices, there's often a compromise between conventional design principles and the demand for greater efficiency.

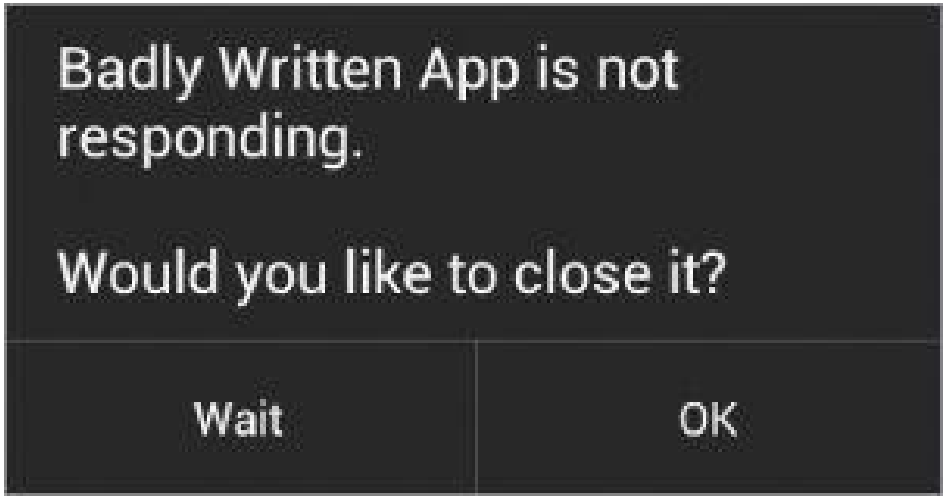
One of the keys to writing efficient Android code is not to carry over assumptions from desktop and server environments to embedded devices.

At a time when 2 to 4GB of memory is standard for most desktop and server rigs, typical smartphones feature approximately 200MB of SDRAM. With memory such a scarce commodity, you need to take special care to use it efficiently. This means thinking about how you use the stack and heap, limiting object creation, and being aware of how variable scope affects memory use.

Being Responsive

Android takes responsiveness very seriously. Android enforces responsiveness with the Activity Manager and Window Manager. If either service detects an unresponsive application, it will display an "[Application] is not responding" dialog—previously described as a *force close* error, as shown in Figure 2.13.

Figure 2.13



This alert is modal, steals focus, and won't go away until you press a button. It's pretty much the last thing you ever want to confront a user with.

Android monitors two conditions to determine responsiveness:

- An application must respond to any user action, such as a key press or screen touch, within five seconds.
- A Broadcast Receiver must return from its `onReceive` handler within 10 seconds.

The most likely culprit in cases of unresponsiveness is a lengthy task being performed on the main application thread. Network or database lookups, complex processing (such as the calculating of game moves), and file I/O should all be moved off the main thread to ensure responsiveness. There are a number of ways to ensure that these actions don't exceed the responsiveness conditions, in particular by using Services and worker threads, as shown in Chapter 9.

Android 2.3 (API level 9) introduced Strict Mode—an API that makes it easier for you to discover file I/O and network transfers being performed on the main application thread. Strict Mode is described in more detail in Chapter 18.

The “[Application] is not responding” dialog is a last resort of usability; the generous five-second limit is a worst-case scenario, not a target. Users will notice a regular pause of anything more than one-half second between key press and action. Happily, a side effect of the efficient code you're already writing will be more responsive applications.

Ensuring Data Freshness

The ability to multitask is a key feature in Android. One of the most important use cases for background Services is to keep your application updated while it's not in use.

Where a responsive application reacts quickly to user interaction, a fresh application quickly displays the data users want to see and interact with. From a usability perspective, the right time to update your application is immediately before the user plans to use it. In practice, you need to weigh the update frequency against its effect on the battery and data usage.

When designing your application, it's critical that you consider how often you will update the data it uses, minimizing the time users are waiting for refreshes or updates, while limiting the effect of these background updates on the battery life.

Developing Secure Applications

Android applications have access to networks and hardware, can be distributed independently, and are built on an open-source platform featuring open communication,

so it shouldn't be surprising that security is a significant consideration.

For the most part, users need to take responsibility for the applications they install and the permissions requests they accept. The Android security model sandboxes each application and restricts access to services and functionality by requiring applications to declare the permissions they require. During installation users are shown the application's required permissions before they commit to installing it.

You can learn more about Android's security model in Chapter 18, "Advanced Android Development," and at http://developer.android.com/resources/faq/security.html.

This doesn't get you off the hook. You not only need to make sure your application is secure for its own sake, but you also need to ensure that it doesn't "leak" permissions and hardware access to compromise the device. You can use several techniques to help maintain device security, and they'll be covered in more detail as you learn the technologies involved. In particular, you should do the following:

- Require permissions for any Services you publish or Intents you broadcast. Take special care when broadcasting an Intent that you aren't leaking secure information, such as location data.
- Take special care when accepting input to your application from external sources, such as the Internet, Bluetooth, NFC, Wi-Fi Direct, SMS messages, or instant messaging (IM). You can find out more about using Bluetooth, NFC, Wi-Fi Direct,

and SMS for application messaging in Chapters 16 and 17.

- Be cautious when your application may expose access to lower-level hardware to third-party applications.
- Minimize the data your application uses and which permissions it requires.

For reasons of clarity and simplicity, many of the examples in this book take a relaxed approach to security. When you're creating your own applications, particularly ones you plan to distribute, this is an area that should not be overlooked.

Ensuring a Seamless User Experience

The idea of a seamless user experience is an important, if somewhat nebulous, concept. What do we mean by *seamless*? The goal is a consistent user experience in which applications start, stop, and transition instantly and without perceptible delays or jarring transitions.

The speed and responsiveness of a mobile device shouldn't degrade the longer it's on. Android's process management helps by acting as a silent assassin, killing background applications to free resources as required. Knowing this, your applications should always present a consistent interface, regardless of whether they're being restarted or resumed.

With an Android device typically running several third-party applications written by different developers, it's particularly important that these applications interact

seamlessly. Using Intents, applications can provide functionality for each other. Knowing your application may provide, or consume, third-party Activities provides additional incentive to maintain a consistent look and feel.

Use a consistent and intuitive approach to usability. You can create applications that are revolutionary and unfamiliar, but even these should integrate cleanly with the wider Android environment.

Persist data between sessions, and when the application isn't visible, suspend tasks that use processor cycles, network bandwidth, or battery life. If your application has processes that need to continue running while your Activities are out of sight, use a Service, but hide these implementation decisions from your users.

When your application is brought back to the front, or restarted, it should seamlessly return to its last visible state. As far as your users are concerned, each application should be sitting silently, ready to be used but just out of sight.

You should also follow the best-practice guidelines for using Notifications and use generic UI elements and themes to maintain consistency among applications.

There are many other techniques you can use to ensure a seamless user experience, and you'll be introduced to some of them as you discover more of the possibilities available in Android in the upcoming chapters.

Providing Accessibility

When designing and developing your applications, it's important not to assume that every user will be exactly like

you. This has implications for internationalization and usability but is critical for providing accessible support for users with disabilities that require them to interact with their Android devices in different ways.

Android provides facilities to help these users navigate their devices more easily using text-to-speech, haptic feedback, and trackball or D-pad navigation.

To provide a good user experience for everyone—including people with visual, physical, or age-related disabilities that prevent them from fully using or seeing a touchscreen—you can leverage Android's accessibility layer.

Best practices for making your application accessible are covered in detail in Chapter 11, “Advanced User Experience.”

As a bonus, the same steps required to help make your touchscreen applications useful for users with disabilities will also make your applications easier to use on non-touch screen devices, such as GoogleTV.

Android Development Tools

The Android SDK includes several tools and utilities to help you create, test, and debug your projects. A detailed examination of each developer tool is outside the scope of this book, but it's worth briefly reviewing what's available. For additional details, check out the Android documentation at http://developer.android.com/guide/developing/tools/index.html.

As mentioned earlier, the ADT plug-in conveniently incorporates many of these tools into the Eclipse IDE, where you can access them from the DDMS perspective, including the following:

- **The Android Virtual Device and SDK Managers**—Used to create and manage AVDs and to download SDK packages, respectively. The AVD hosts an Emulator running a particular build of Android, letting you specify the supported SDK version, screen resolution, amount of SD card storage available, and available hardware capabilities (such as touchscreens and GPS).
- **The Android Emulator**—An implementation of the Android VM designed to run within an AVD on your

development computer. Use the Emulator to test and debug your Android applications.

- **Dalvik Debug Monitoring Service (DDMS)**—Use the DDMS to monitor and control the Emulators on which you're debugging your applications.
- **Android Debug Bridge (ADB)**—A client-server application that provides a link to virtual and physical devices. It lets you copy files, install compiled application packages (`.apk`), and run shell commands.
- **Logcat**—A utility used to view and filter the output of the Android logging system.
- **Android Asset Packaging Tool (AAPT)**—Constructs the distributable Android package files (`.apk`).

The following additional tools are also available:

- **SQLite3**—A database tool that you can use to access the SQLite database files created and used by Android.
- **Traceview and dmtracedump**—Graphical analysis tools for viewing the trace logs from your Android application.
- **Hprof-conv**—A tool that converts HPROF profiling output files into a standard format to view in your preferred profiling tool.
- **MkSDCard**—Creates an SD card disk image that can be used by the Emulator to simulate an external

storage card.

- **Dx**—Converts Java `.class` bytecode into Android `.dex` bytecode.
- **Hierarchy Viewer**—Provides both a visual representation of a layout's View hierarchy to debug and optimize your UI, and a magnified display to get your layouts pixel-perfect.
- **Lint**—A tool that analyzes your application and its resources to suggest improvements and optimizations.
- **Draw9patch:** A handy utility to simplify the creation of NinePatch graphics using a WYSIWYG editor.
- **Monkey and Monkey Runner:** Monkey runs within the VM, generating pseudo-random user and system events. Monkey Runner provides an API for writing programs to control the VM from outside your application.
- **ProGuard**—A tool to shrink and obfuscate your code by replacing class, variable, and method names with semantically meaningless alternatives. This is useful to make your code more difficult to reverse engineer.

Now take a look at some of the more important tools in more detail.

The Android Virtual Device Manager

The Android Virtual Device Manager is used to create and manage the virtual devices that will host instances of the Emulator.

AVDs are used to simulate the software builds and hardware configurations available on different physical devices. This lets you test your application on a variety of hardware platforms without needing to buy a variety of phones.

The Android SDK doesn't include any prebuilt virtual devices, so you will need to create at least one device before you can run your applications within an Emulator.

Each virtual device is configured with a name, a target build of Android (based on the SDK version it supports), an SD card capacity, and screen resolution, as shown in the Create new Android Virtual Device (AVD) dialog in Figure 2.14.

Figure 2.14


Create new Android Virtual Device (AVD)
Name:
Galaxy_Nexus
Target:
Google APIs (Google Inc.) - API Level 14
CPU/ABI:
ARM (armeabi-v7a)
SD Card:
Size:
32
MiB
File:
Browse...
Snapshot:
Enabled
Skin:
Built-in:
WXGA720
Resolution:
x
Hardware:
Property
Value
Hardware Back/Home keys
no
Abstracted LCD density
320
Keyboard lid support
no
Max VM application hea...
48
Device ram size
1024
New...
Delete
Override the existing AVD with the same name
Cancel
Create AVD

You can also choose to enable snapshots to save the Emulator state when it's closed. Starting a new Emulator from a snapshot is significantly faster.

Each virtual device also supports a number of specific hardware settings and restrictions that can be added in the form of name-value pairs (NVPs) in the hardware table. Selecting one of the built-in skins will automatically configure these additional settings corresponding to the device the skin represents.

The additional settings include the following:

- Maximum VM heap size
- Screen pixel density
- SD card support
- Existence of D-pad, touchscreen, keyboard, and trackball hardware
- Accelerometer, GPS, and proximity sensor support
- Available device memory
- Camera hardware (and resolution)
- Support for audio recording
- Existence of hardware back and home keys

Different hardware settings and screen resolutions will present alternative UI skins to represent the different hardware configurations. This simulates a variety of mobile device types. Some manufacturers have made hardware presets and virtual device skins available for their devices. Some, including Samsung, are available as SDK

packages.

Android SDK Manager

The Android SDK Manager can be used to see which version of the SDK you have installed and to install new SDKs when they are released.

Each platform release is displayed, along with the platform tools and a number of additional support packages. Each platform release includes the SDK platform, documentation, tools, and examples corresponding to that release.

The Android Emulator

The Emulator is available for testing and debugging your applications.

The Emulator is an implementation of the Dalvik VM, making it as valid a platform for running Android applications as any Android phone. Because it's decoupled from any particular hardware, it's an excellent baseline to use for testing your applications.

Full network connectivity is provided along with the ability to tweak the Internet connection speed and latency while debugging your applications. You can also simulate placing and receiving voice calls and SMS messages.

The ADT plug-in integrates the Emulator into Eclipse so that it's launched automatically within the selected AVD when you run or debug your projects. If you aren't using the plug-in or want to use the Emulator outside of Eclipse, you can telnet into the Emulator and control it from its console. (For more details on controlling the Emulator, check out the documentation at http://developer.android.com/guide/developing/tools/emulator.html.)

To execute the Emulator, you first need to create a virtual device, as described in the previous section. The Emulator will launch the virtual device and run a Dalvik instance within it.

At the time of this writing, the Emulator doesn't implement all the mobile hardware features supported by Android. For example, it

does not implement the camera, vibration, LEDs, actual phone calls, accelerometer, USB connections, audio capture, or battery charge level.

The Dalvik Debug Monitor Service

The Emulator enables you to see how your application will look, behave, and interact, but to actually see what's happening under the surface, you need the Dalvik Debug Monitoring Service. The DDMS is a powerful debugging tool that lets you interrogate active processes, view the stack and heap, watch and pause active threads, and explore the filesystem of any connected Android device.

The DDMS perspective in Eclipse also provides simplified access to screen captures of the Emulator and the logs generated by LogCat.

If you're using the ADT plug-in, the DDMS tool is fully integrated into Eclipse and is available from the DDMS perspective. If you aren't using the plug-in or Eclipse, you can run DDMS from the command line (it's available from the tools folder of the Android SDK), and it will automatically connect to any running device or Emulator.

The Android Debug Bridge

The *Android Debug Bridge* (ADB) is a client-service application that lets you connect with an Android device (virtual or actual). It's made up of three components:

- A daemon running on the device or Emulator
- A service that runs on your development computer
- Client applications (such as the DDMS) that communicate with the daemon through the service

As a communications conduit between your development hardware and the Android device/Emulator, the ADB lets you install applications, push and pull files, and run shell commands on the target device. Using the device shell, you can change logging settings and query or modify SQLite databases available on the device.

The ADT tool automates and simplifies a lot of the usual interaction with the ADB, including application installation and updating, file logging, and file transfer (through the DDMS perspective).

To learn more about what you can do with the ADB, check out the documentation at http://developer.android.com/guide/developing/tools/adb.html.

The Hierarchy Viewer and Lint Tool

To build applications that are fast and responsive, you need to optimize your UI. The Hierarchy Viewer and Lint tools help you analyze, debug, and optimize the XML layout definitions used within your application.

The Hierarchy Viewer displays a visual representation of the structure of your UI layout. Starting at the root node, the children of each nested View (including layouts) is displayed in a hierarchy. Each View node includes its name, appearance, and identifier.

To optimize performance, the performance of the layout, measure, and draw steps of creating the UI of each View at runtime is displayed. Using these values, you can learn the actual time taken to create each View within your hierarchy, with colored “traffic light” indicators showing the relative performance for each step. You can then search within your layout for Views that appear to be taking longer to render than they should.

The Lint tool helps you to optimize your layouts by checking them for a series of common inefficiencies that can have a negative impact on your application's performance. Common issues include a surplus of nested layouts, a surplus of Views within a layout, and unnecessary parent Views.

Although a detailed investigation into optimizing and debugging your UI is beyond the scope of this book, you can find further details at http://developer.android.com/guide/developing/debugging/debugging-ui.html.

Monkey and Monkey Runner

Monkey and Monkey Runner can be used to test your applications stability from a UI perspective.

Monkey works from within the ADB shell, sending a stream of pseudo-random system and UI events to your application. It's particularly useful to stress test your applications to investigate edge-cases you might not have anticipated through unconventional use of the UI.

Alternatively, Monkey Runner is a Python scripting API that lets you send specific UI commands to control an Emulator or device from outside the application. It's extremely useful for performing UI, functional, and unit tests in a predictable, repeatable fashion.

Chapter 3

Creating Applications and Activities

What's in this Chapter?

- Introducing the Android application components and the different types of applications you can build with them
- Understanding the Android application lifecycle
- Creating your application manifest
- Using external resources to provide dynamic support for locations, languages, and hardware configurations
- Implementing and using your own Application class
- Creating new Activities
- Understanding an Activity's state transitions and lifecycle

To write high-quality applications, it's important to understand the components they consist of and how those components are bound together by the Android manifest.

This chapter introduces each of the application components, with special attention paid to Activities.

Next, you'll see why and how you should use external resources and the resource hierarchy to create applications that can be customized and optimized for a variety of devices, countries, and languages.

In Chapter 2, “Getting Started,” you learned that each Android application runs in a separate process, in its own instance of the Dalvik virtual machine. In this chapter, you learn more about the application lifecycle and how the Android run time can manage your application. You are also introduced to the application and Activity states, state transitions, and event handlers. The application's state determines its priority, which, in turn, affects the likelihood of its being terminated when the system requires more resources.

You should always provide the best possible experience for users, no matter which country they're in or which of the wide variety of Android device types, form factors, and screen sizes they're using. In this chapter, you learn how to use the resource framework to provide optimized resources, ensuring your applications run seamlessly on different hardware (particularly different screen resolutions and pixel densities), in different countries, and supporting multiple languages.

The `Activity` class forms the basis for all your user interface (UI) screens. You learn how to create Activities and gain an understanding of their lifecycles and how they

affect the application lifetime and priority.

Finally, you are introduced to some of the `Activity` subclasses that simplify resource management for some common UI patterns, such as map- and list-based Activities.

What Makes an Android Application?

Android applications consist of loosely coupled components, bound by the application manifest that describes each component and how they interact. The manifest is also used to specify the application's metadata, its hardware and platform requirements, external libraries, and required permissions.

The following components comprise the building blocks for all your Android applications:

- **Activities**—Your application's presentation layer. The UI of your application is built around one or more extensions of the `Activity` class. Activities use Fragments and Views to layout and display information, and to respond to user actions. Compared to desktop development, Activities are equivalent to Forms. You'll learn more about Activities later in this chapter.
- **Services**—The invisible workers of your application. Service components run without a UI, updating your data sources and Activities, triggering Notifications, and broadcasting Intents. They're used to perform

long running tasks, or those that require no user interaction (such as network lookups or tasks that need to continue even when your application's Activities aren't active or visible.) You'll learn more about how to create and use services in Chapter 9, “Working in the Background.”

- **Content Providers**—Shareable persistent data storage. Content Providers manage and persist application data and typically interact with SQL databases. They're also the preferred means to share data across application boundaries. You can configure your application's Content Providers to allow access from other applications, and you can access the Content Providers exposed by others. Android devices include several native Content Providers that expose useful databases such as the media store and contacts. You'll learn how to create and use Content Providers in Chapter 8, “Databases and Content Providers.”
- **Intents**—A powerful interapplication message-passing framework. Intents are used extensively throughout Android. You can use Intents to start and stop Activities and Services, to broadcast messages system-wide or to an explicit Activity, Service, or Broadcast Receiver, or to request an action be performed on a particular piece of data. Explicit, implicit, and broadcast Intents are explored in more detail in Chapter 5, “Intents and Broadcast Receivers.”

- **Broadcast Receivers**—Intent listeners. Broadcast Receivers enable your application to listen for Intents that match the criteria you specify. Broadcast Receivers start your application to react to any received Intent, making them perfect for creating event-driven applications. Broadcast Receivers are covered with Intents in Chapter 5.
- **Widgets**—Visual application components that are typically added to the device home screen. A special variation of a Broadcast Receiver, widgets enable you to create dynamic, interactive application components for users to embed on their home screens. You'll learn how to create your own widgets in Chapter 14, “Invading the Home Screen.”
- **Notifications**—Notifications enable you to alert users to application events without stealing focus or interrupting their current Activity. They're the preferred technique for getting a user's attention when your application is not visible or active, particularly from within a Service or Broadcast Receiver. For example, when a device receives a text message or an email, the messaging and Gmail applications use Notifications to alert you by flashing lights, playing sounds, displaying icons, and scrolling a text summary. You can trigger these notifications from your applications, as discussed in Chapter 10, “Expanding the User Experience.”

By decoupling the dependencies between application components, you can share and use individual Content Providers, Services, and even Activities with other applications—both your own and those of third parties.

Introducing the Application Manifest File

Each Android project includes a manifest file, `AndroidManifest.xml`, stored in the root of its project hierarchy. The manifest defines the structure and metadata of your application, its components, and its requirements.

It includes nodes for each of the Activities, Services, Content Providers, and Broadcast Receivers that make up your application and, using Intent Filters and Permissions, determines how they interact with each other and with other applications.

The manifest can also specify application metadata (such as its icon, version number, or theme), and additional top-level nodes can specify any required permissions, unit tests, and define hardware, screen, or platform requirements (as described next).

The manifest is made up of a root `manifest` tag with a `package` attribute set to the project's package. It should also include an `xmlns:android` attribute that supplies several system attributes used within the file.

Use the `versionCode` attribute to define the current application version as an integer that increases with each

version iteration, and use the `versionName` attribute to specify a public version that will be displayed to users.

You can also specify whether to allow (or prefer) for your application be installed on external storage (usually an SD card) rather than internal storage using the `installLocation` attribute. To do this specify either `preferExternal` or `auto`, where the former installs to external storage whenever possible, and the latter asks the system to decide.

If your application is installed on external storage, it will be immediately killed if a user mounts the USB mass storage to copy files to/from a computer, or ejects or unmounts the SD card.

If you don't specify an install location attribute, your application will be installed in the internal storage and users won't be able to move it to external storage. The total amount of internal storage is generally limited, so it's good practice to let your application be installed on external storage whenever possible.

There are some applications for which installation to external storage is not appropriate due to the consequences of unmounting or ejecting the external storage, including:

- **Applications with Widgets, Live Wallpapers, and Live Folders**—Your Widgets, Live Wallpapers, and Live Folders will be removed from the home screen

and may not be available until the system restarts.

- **Applications with ongoing Services**—Your application and its running Services will be stopped and won't be restarted automatically.
- **Input Method Engines**—Any IME installed on external storage will be disabled and must be reselected by the user after the external storage is once again available.
- **Device administrators**—Your `DeviceAdminReceiver` and any associated admin capabilities will be disabled.

A Closer Look at the Application Manifest

The following XML snippet shows a typical manifest node:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
          package="com.paad.myapp"
          android:versionCode="1"
          android:versionName="0.9 Beta"
          android:installLocation="preferExternal">
          [ ... manifest nodes ... ]
</manifest>
```

The `manifest` tag can include nodes that define the application components, security settings, test classes, and requirements that make up your application. The following list gives a summary of the available `manifest` sub-node tags and provides an XML snippet demonstrating how each tag is used:

- `uses-sdk`—This node enables you to define a minimum and maximum SDK version that must be available on a device for your application to function properly, and target SDK for which it has been designed using a combination of `minSDKVersion`, `maxSDKVersion`, and `targetSDKVersion` attributes, respectively.

The minimum SDK version specifies the lowest version of the SDK that includes the APIs you have used in your application. If you fail to specify a minimum version, it defaults to 1, and your application crashes when it attempts to access unavailable APIs.

The target SDK version attribute enables you to specify the platform against which you did your development and testing. Setting a target SDK version tells the system that there is no need to apply any forward- or backward-compatibility changes to support that particular version. To take advantage of the newest platform UI improvements, it's considered good practice to update the target SDK of your application to the latest platform release after you confirm it behaves as expected, even if you aren't

making use of any new APIs.

It is usually unnecessary to specify a maximum SDK, and doing so is strongly discouraged. The maximum SDK defines an upper limit you are willing to support and your application will not be visible on the Google Play Store for devices running a higher platform release. Devices running on platforms higher than Android 2.0.1 (API level 6) will ignore any maximum SDK values at installation time.

```
<uses-sdk android:minSdkVersion="6"
          android:targetSdkVersion="15"/>
```

The supported SDK version is not equivalent to the platform version and cannot be derived from it. For example, Android platform release 4.0 supports the SDK version 14. To find the correct SDK version for each platform, use the table at http://developer.android.com/guide/appendix/api-levels.html.

- `uses-configuration`— The `uses-configuration` nodes specify each combination of input mechanisms are supported by your application. You shouldn't normally need to include this node, though it can be useful for games that require particular input controls. You can specify any combination of input devices that include the following:
  - `reqFiveWayNav`—Specify `true` for this attribute if you require an input device capable of navigating up, down, left, and right and of clicking the current selection. This includes both trackballs and directional pads (D-pads).
  - `reqHardKeyboard`—If your application requires a hardware keyboard, specify `true`.
  - `reqKeyboardType`—Lets you specify the keyboard type as one of `nokeys`, `qwerty`, `twelvekey`, or `undefined`.
  - `reqNavigation`—Specify the attribute

value as one of `nonav`, `dpad`, `trackball`, `wheel`, or `undefined` as a required navigation device.

- `reqTouchScreen`—Select one of `notouch`, `stylus`, `finger`, or `undefined` to specify the required touchscreen input.

You can specify multiple supported configurations, for example, a device with a finger touchscreen, a trackball, and either a QUERTY or a twelve-key hardware keyboard, as shown here:

```
<uses-configuration
android:reqTouchScreen="finger"

 android:reqNavigation="trackball"

 android:reqHardKeyboard="true"

 android:reqKeyboardType="qwerty"/>
<uses-configuration
android:reqTouchScreen="finger"

 android:reqNavigation="trackball"

 android:reqHardKeyboard="true"

 android:reqKeyboardType="twelvekey"/>
```

When specifying required configurations, be aware that your application won't be installed on any device that does not have one of the combinations specified. In the preceding example, a device with a QWERTY keyboard and a D-pad (but no touchscreen or trackball) would not be supported. Ideally, you should develop your application to ensure it works with any input configuration, in which case no `uses-configuration` node is required.

- `uses-feature` —Android is available on a wide variety of hardware platforms. Use multiple `uses-feature` nodes to specify which hardware features your application requires. This prevents your application

from being installed on a device that does not include a required piece of hardware, such as NFC hardware, as follows:

```
<uses-feature
android:name="android.hardware.nfc" />
```

You can require support for any hardware that is optional on a compatible device. Currently, optional hardware features include the following:

- **Audio**—For applications that requires a low-latency audio pipeline. Note that at the time of writing this book, no Android devices satisfied this requirement.
- **Bluetooth**—Where a Bluetooth radio is required.
- **Camera**—For applications that require a camera. You can also require (or set as options) autofocus, flash, or a front-facing camera.
- **Location**—If you require location-based services. You can also specify either network or GPS support explicitly.
- **Microphone**—For applications that require audio input.
- **NFC**—Requires NFC (near-field communications) support.
- **Sensors**—Enables you to specify a requirement for any of the potentially available hardware sensors.
- **Telephony**—Specify that either telephony in general, or a specific telephony radio (GSM or CDMA) is required.
- **Touchscreen**—To specify the type of touch-screen your application requires.
- **USB**—For applications that

require either USB host or accessory mode support.
- **Wi-Fi**—Where Wi-Fi networking support is required.

As the variety of platforms on which Android is available increases, so too will the optional hardware. You can find a full list of `uses-feature` hardware at http://developer.android.com/guide/topics/manifest/uses-feature-element.html#features-reference.

To ensure compatibility, requiring some permissions implies a feature requirement. In particular, requesting permission to access Bluetooth, the camera, any of the location service permissions, audio recording, Wi-Fi, and telephony-related permissions implies the corresponding hardware features. You can override these implied requirements by adding a `required` attribute and setting it to `false`—for example, a note-taking application that supports recording an audio note:

```
<uses-feature
android:name="android.hardware.microphone"
              android:required="false"
/>
```

The camera hardware also represents a special case. For compatibility reasons requesting permission to use the camera, or adding a uses-feature node requiring it, implies a requirement for the camera to support autofocus. You can specify it as optional as appropriate:

```
<uses-feature
android:name="android.hardware.camera"
/>
<uses-feature
android:name="android.hardware.camera.autofocus"

 android:required="false" />
```

```
<uses-feature
android:name="android.hardware.camera.flash"

 android:required="false" />
```

You can also use the `uses-feature` node to specify the minimum version of OpenGL required by your application. Use the `glEsVersion` attribute, specifying the OpenGL ES version as an integer. The higher 16 bits represent the major number and the lower 16 bits represent the minor number, so version 1.1 would be represented as follows:

```
<uses-feature
android:glEsVersion="0x0010001"
/>
```

- `supports-screens`—The first Android devices were limited to 3.2” HVGA hardware. Since then, hundreds of new Android devices have been launched including tiny 2.55” QVGA phones, 10.1” tablets, and 42” HD televisions. The `supports-screen` node enables you to specify the screen sizes your application has been designed and tested to. On devices with supported screens, your application is laid out normally using the scaling properties associated with the layout files you supply. On unsupported devices the system may apply a “compatibility mode,” such as pixel scaling to display your application. It's best practice to create scalable layouts that adapt to all screen dimensions.

  You can use two sets of attributes when describing your screen support. The first set is used primarily for devices running Android versions prior to Honeycomb MR2 (API level 13). Each attribute takes a Boolean specifying support. As of SDK 1.6

(API level 4), the default value for each attribute is `true`, so use this node to specify the screen sizes you do not support.

- `smallScreens`—Screens with a resolution smaller than traditional HVGA (typically, QVGA screens).
- `normalScreens`—Used to specify typical mobile phone screens of at least HVGA, including WVGA and WQVGA.
- `largeScreens`—Screens larger than normal. In this instance a large screen is considered to be significantly larger than a mobile phone display.
- `xlargeScreens`—Screens larger than large-typically tablet devices.

Honeycomb MR2 (API level 13) introduced additional attributes that provide a finer level of control over the size of screen your application layouts can support. It is generally good practice to use these in combination with the earlier attributes if your application is available to devices running platform releases earlier than API level 13.

- `requiresSmallestWidthDp`—Enables you to specify a minimum supported screen width in device independent pixels. The smallest screen width for a device is the lower dimension of its screen height and width. This attribute can potentially be used to filter applications from the Google Play Store for devices with unsupported screens, so when used it should specify the absolute minimum number of pixels required for your layouts to provide a useable user experience.
- `compatibleWidthLimitDp`—Specifies the upper bound beyond which your application may not scale. This can cause the system to enable a

compatibility mode on devices with screen resolutions larger than you specify.

- `largestWidthLimitDp`—Specifies the absolute upper bound beyond which you know your application will not scale appropriately. Typically this results in the system forcing the application to run in compatibility mode (without the ability for users to disable it) on devices with screen resolutions larger than that specified.

It is generally considered a bad user experience to force your application into compatibility mode. Wherever possible, ensure that your layouts scale in a way that makes them usable on larger devices.

```
<supports-screens
android:smallScreens="false"

 android:normalScreens="true"
                 android:largeScreens="true"

 android:xlargeScreens="true"

 android:requiresSmallestWidthDp="480"

 android:compatibleWidthLimitDp="600"

 android:largestWidthLimitDp="720"/>
```

Where possible you should optimize your application for different screen resolutions and densities using the resources folder, as shown later in this chapter, rather than enforcing a subset of supported screens.

- `supports-gl-texture`—Declares that the application is capable of providing texture assets that are compressed using a particular GL texture compression format. You must use multiple supports-gl-texture elements if your application is capable of supporting multiple texture compression formats. You can find the most up-to-date

list of supported GL texture compression format values at http://developer.android.com/guide/topics/manifest/supports-gl-texture-element.html.

```
<supports-gl-texture
android:name="GL_OES_compressed_ETC1_RGB8_texture"
/>
```

- `uses-permission`—As part of the security model, `uses-permission` tags declare the user permissions your application requires. Each permission you specify will be presented to the user before the application is installed. Permissions are required for many APIs and method calls, generally those with an associated cost or security implication (such as dialing, receiving SMS, or using the location-based services).

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

- `permission`—Your application components can also create permissions to restrict access to shared application components. You can use the existing platform permissions for this purpose or define your own permissions in the manifest. To do this, use the `permission` tag to create a permission definition.

Your application components can then create permissions by adding an `android:permission` attribute. Then you can include a `uses-permission` tag in your manifest to use these protected components, both in the application that includes the protected component and any other application that wants to use it.

Within the `permission` tag, you can specify the level of access the permission permits (`normal`, `dangerous`, `signature`, `signatureOrSystem`), a label, and an external resource containing the description that explains the risks of granting the specified permission. More details on creating and using your own permissions can be found in Chapter 18, “Advanced Android Development.”

```
<permission android:name="com.paad.DETONATE_DEVICE"
            android:protectionLevel="dangerous"
            android:label="Self Destruct"

 android:description="@string/detonate_description">
</permission>
```

- `instrumentation`—Instrumentation classes provide a test framework for your application components at run time. They provide hooks to monitor your application and its interaction with the system resources. Create a new node for each of the test classes you've created for your application.

  ```
  <instrumentation android:label="My Test"

  android:name=".MyTestClass"

  android:targetPackage="com.paad.apackage">
  </instrumentation>
  ```

  Note that you can use a period (.) as shorthand for prepending the manifest package to a class within your package.

- `application`—A manifest can contain only one application node. It uses attributes to specify the metadata for your application (including its title, icon, and theme). During development you should include a `debuggable` attribute set to `true` to enable debugging, then be sure to disable it for your release builds.

  The `application` node also acts as a container for the Activity, Service, Content Provider, and Broadcast Receiver nodes that specify the application components. Later in this chapter you'll learn how to create and use your own Application class extension to manage application state. You specify the name of your custom application class using the `android:name` attribute.

  ```
  <application
  ```

```
android:icon="@drawable/icon"

android:logo="@drawable/logo"

android:theme="@android:style/Theme.Light"

android:name=".MyApplicationClass"
            android:debuggable="true">
            [ ... application nodes ...
]
</application>
```

- `activity`—An `activity` tag is required for every Activity within your application. Use the `android:name` attribute to specify the `Activity` class name. You must include the main launch Activity and any other Activity that may be displayed. Trying to start an Activity that's not defined in the manifest will throw a runtime exception. Each Activity node supports `intent-filter` child tags that define the Intents that can be used to start the Activity. Later in this chapter you'll explore the Activity manifest entry in more detail.

Note, again, that a period is used as shorthand for the application's package name when specifying the Activity's class name.

```
<activity
android:name=".MyActivity"
android:label="@string/app_name">
  <intent-filter>
    <action
android:name="android.intent.action.MAIN"
/>
    <category
android:name=»android.intent.category.LAUNCHER»
/>
  </intent-filter>
</activity>
```

- `service`—As with the `activity` tag, add a `service` tag for each

`Service` class used in your application. Service tags also support `intent-filter` child tags to allow late runtime binding.

```
<service
android:name=".MyService">
</service>
```

- `provider`—Provider tags specify each of your application's Content Providers. Content Providers are used to manage database access and sharing.

```
<provider
android:name=".MyContentProvider"

 android:authorities="com.paad.myapp.MyContentProvider"/>
```

- `receiver`—By adding a receiver tag, you can register a Broadcast Receiver without having to launch your application first. As you'll see in Chapter 5, Broadcast Receivers are like global event listeners that, when registered, will execute whenever a matching Intent is broadcast by the system or an application. By registering a Broadcast Receiver in the manifest you can make this process entirely autonomous. If a matching Intent is broadcast, your application will be

started automatically and the registered Broadcast Receiver will be executed. Each receiver node supports `intent-filter` child tags that define the Intents that can be used to trigger the receiver:

```
<receiver
android:name=".MyIntentReceiver">
  <intent-filter>
    <action
android:name="com.paad.mybroadcastaction"
/>
  </intent-filter>
</receiver>
```

- `uses-library`—Used to specify a shared library that this application requires. For example, the maps APIs described in Chapter 13, “Maps, Geocoding, and Location-Based Services,” are packaged as a separate library that is not automatically linked. You can specify that a particular package is required—which prevents the application from being installed on devices without the specified library—or optional, in which case your application must use reflection to check for the library before attempting to make use of it.

```
<uses-library
android:name="com.google.android.maps"

 android:required="false"/>
```

You can find a more detailed description of the manifest and each of these nodes at http://developer.android.com/guide/topics/manifest/manifest-intro.html.

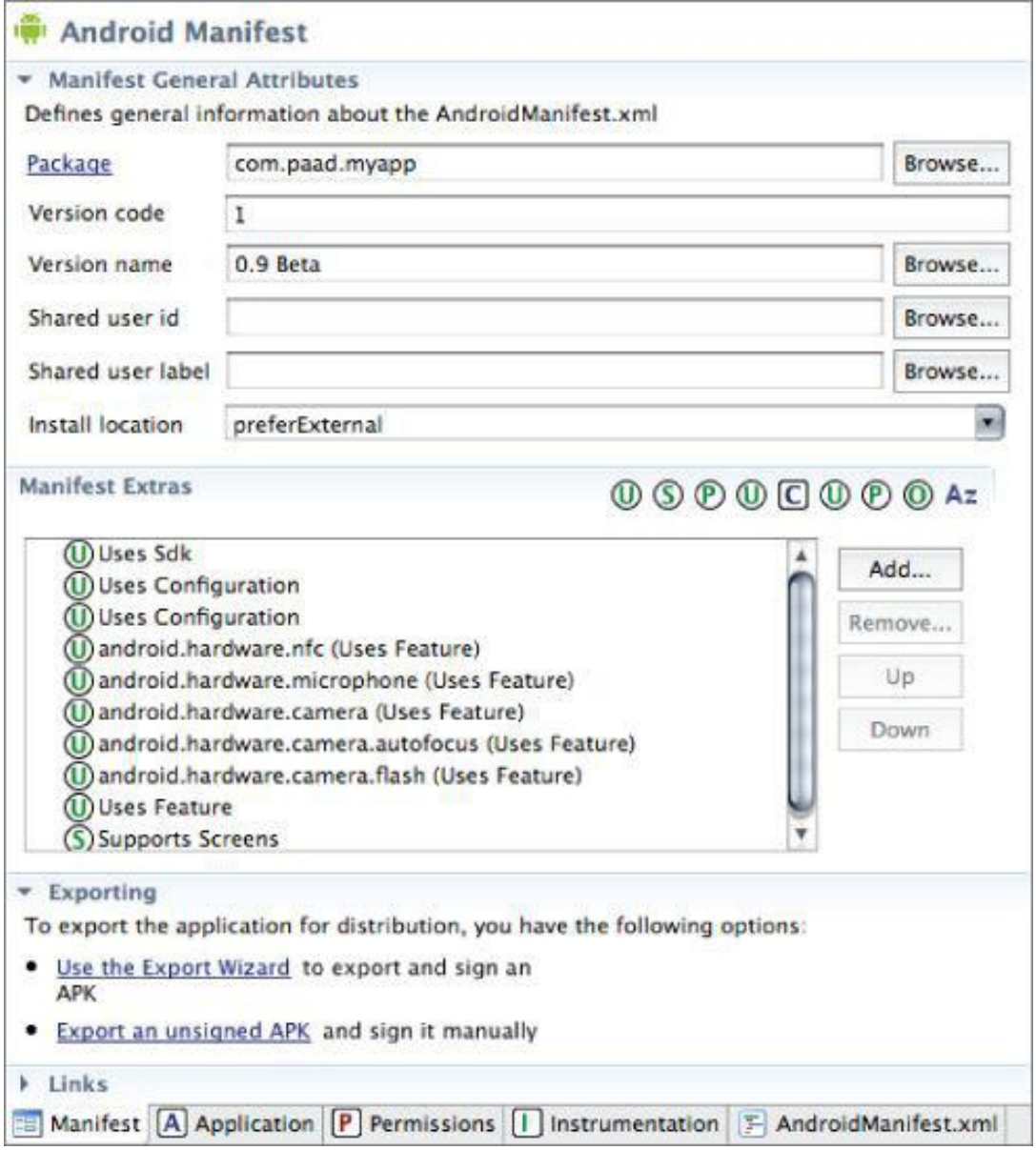
The ADT New Project Wizard automatically creates a new manifest file when it creates a new project. You'll return to the manifest as each of the application components is introduced and explored.

Using the Manifest Editor

The Android Development Tools (ADT) plug-in includes a Manifest Editor, so you don't have to manipulate the underlying XML directly.

To use the Manifest Editor in Eclipse, right-click the `AndroidManifest.xml` file in your project folder, and select Open With → Android Manifest Editor. This presents the Android Manifest Overview screen, as shown in Figure 3.1. This screen gives you a high-level view of your application structure, enabling you to set your application version information and root level manifest nodes, including `uses-sdk` and `uses-features`, as described previously in this chapter. It also provides shortcut links to the Application, Permissions, Instrumentation, and raw XML screens.

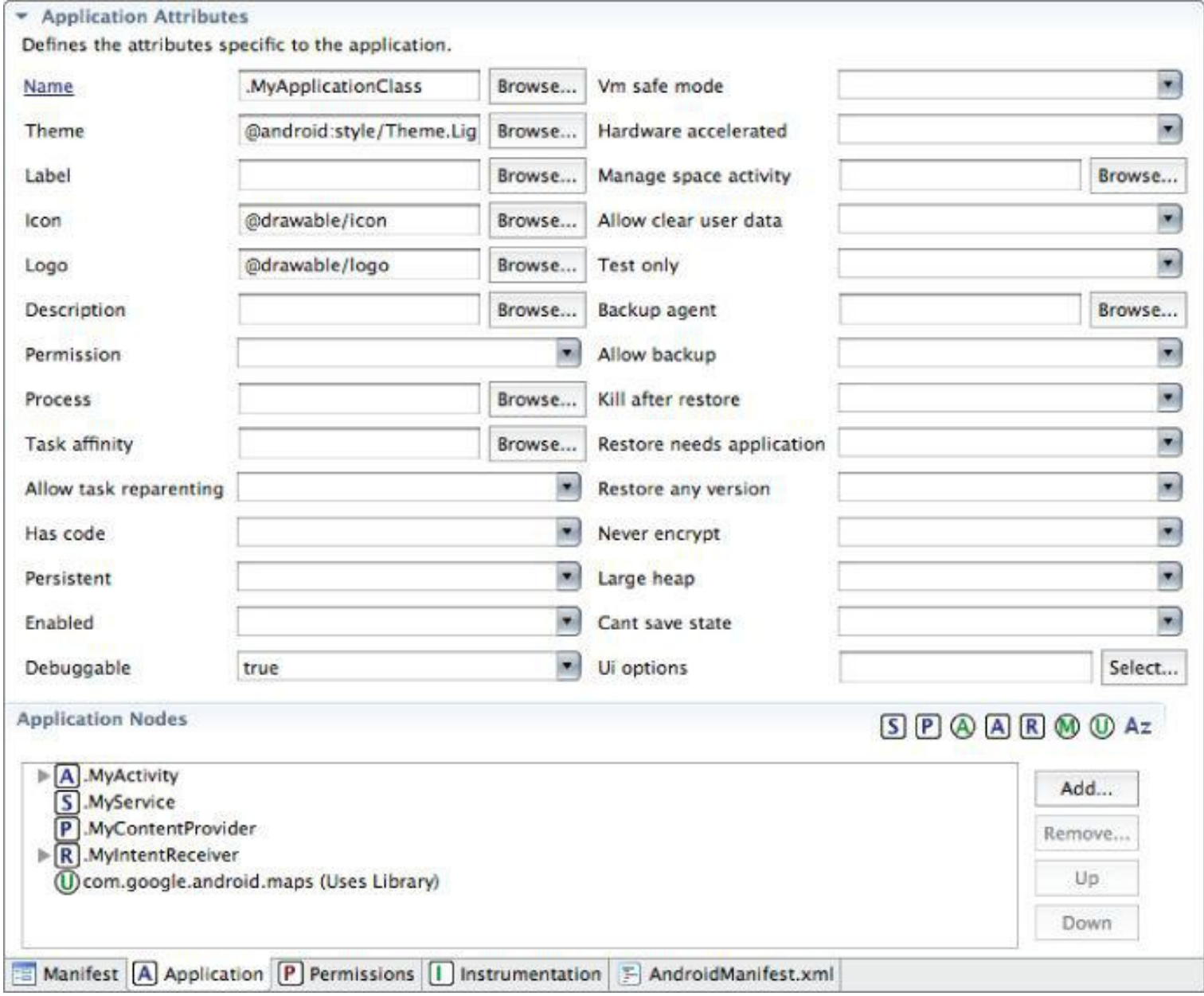
Figure 3.1



Each of the next three tabs contains a visual interface for managing the application, security, and instrumentation (testing) settings, while the last tab (labeled with the manifest's filename) gives access to the underlying XML.

Of particular interest is the Application tab, as shown in Figure 3.2. Use it to manage the application node and the application component hierarchy, where you specify each of your application's components.

Figure 3.2



You can specify an application's attributes—including its icon, label, and theme—in the Application Attributes panel. The Application Nodes tree beneath it lets you manage the application components, including their attributes and any associated Intent Filters.

Externalizing Resources

It's always good practice to keep non-code resources, such as images and string constants, external to your code. Android supports the externalization of resources, ranging from simple values such as strings and colors to more complex resources such as images (Drawables), animations, themes, and menus. Perhaps the most powerful externalizable resources are layouts.

By externalizing resources, you make them easier to maintain, update, and manage. This also lets you easily define alternative resource values for internationalization and to include different resources to support variations in hardware—particularly, screen size and resolution.

You'll see later in this section how Android dynamically selects resources from resource trees that contain different values for alternative hardware configurations, languages, and locations. When an application starts, Android automatically selects the correct resources without you having to write a line of code.

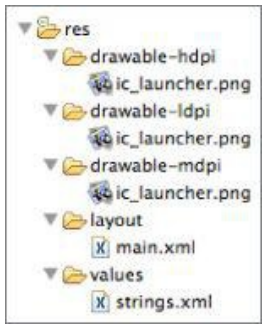
Among other things, this lets you change the layout based on the screen size and orientation, images based on screen density, and customize text prompts based on a user's language and country.

Creating Resources

Application resources are stored under the `res` folder in your project hierarchy. Each of the available resource types is stored in subfolders, grouped by resource type.

If you start a project using the ADT Wizard, it creates a `res` folder that contains subfolders for the `values`, `drawable-ldpi`, `drawable-mdpi`, `drawable-hdpi`, and `layout` resources that contain the default string resource definitions, application icon, and layouts respectively, as shown in Figure 3.3.

Figure 3.3



Note that three `drawable` resource folders contain three different icons: one each for low, medium, and high density displays respectively.

Each resource type is stored in a different folder: simple values, Drawables, colors, layouts, animations, styles, menus, XML files (including searchables), and raw resources. When your application is built, these resources will be compiled and compressed as efficiently as possible and included in your application package.

This process also generates an `R` class file that contains references to each of the resources you include in your project. This enables you to reference the resources in your code, with the advantage of design-time syntax checking.

The following sections describe many of the specific resource types available within these categories and how to create them for your applications.

In all cases, the resource filenames should contain only lowercase letters, numbers, and the period (.) and underscore (_) symbols.

Simple Values

Supported simple values include strings, colors, dimensions, styles, and string or integer arrays. All simple values are stored within XML files in the `res/values` folder.

Within each XML file, you indicate the type of value being stored using tags, as shown in the sample XML file in Listing 3.1.

Listing 3.1: Simple values XML

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">To Do List</string>
  <plurals name="androidPlural">
    <item quantity="one">One android</item>
    <item quantity="other">%d androids</item>
  </plurals>
  <color name="app_background">#FF0000FF</color>
  <dimen name="default_border">5px</dimen>
  <string-array name="string_array">
    <item>Item 1</item>
    <item>Item 2</item>
    <item>Item 3</item>
  </string-array>
  <array name="integer_array">
    <item>3</item>
    <item>2</item>
    <item>1</item>
  </array>
</resources>
```

code snippet
PA4AD_Ch03_Manifest_and_Resources/res/values/simple_values.xml

This example includes all the simple value types. By convention, resources are generally stored in separate files, one for each type; for example, `res/values/strings.xml` would contain only string resources.

The following sections detail the options for defining simple resources.

Strings

Externalizing your strings helps maintain consistency within your application and makes it much easier to internationalize them.

String resources are specified with the `string` tag, as shown in the following XML snippet:

```
<string name="stop_message">Stop.</string>
```

Android supports simple text styling, so you can use the HTML tags `<b>`, `<i>`, and `<u>` to apply bold, italics, or underlining, respectively, to parts of your text strings, as shown in the following example:

```
<string name="stop_message"><b>Stop.</b></string>
```

You can use resource strings as input parameters for the `String.format` method. However, `String.format` does not support the text styling previously described. To apply styling to a format string, you have to escape the HTML tags when creating your resource, as shown in the following snippet:

```
<string name="stop_message">&lt;b&gt;Stop&lt;/b&gt;.
%1$s</string>
```

Within your code, use the `Html.fromHtml` method to convert this back into a styled character sequence.

```
String rString =
getString(R.string.stop_message);
String fString = String.format(rString,
"Collaborate and listen.");
CharSequence styledString =
Html.fromHtml(fString);
```

You can also define alternative plural forms for your strings. This enables you to define different strings based on the number of items you refer to. For example, in English you would refer to “one Android” or “seven Androids.”

By creating a plurals resource, you can specify an alternative string for any of zero, one, multiple, few, many, or other quantities. In English only the singular is a special case, but some languages require finer detail:

```
<plurals name="unicornCount">
  <item quantity="one">One
unicorn</item>
  <item quantity="other">%d
unicorns</item>
</plurals>
```

To access the correct plural in code, use the `getQuantityString` method on your application's `Resources` object, passing in the resource ID of the plural resource, and specifying the number of objects you want to describe:

```
Resources resources =
getResources();
String unicornStr =
resources.getQuantityString(
  R.plurals.unicornCount,
unicornCount, unicornCount);
```

The object count is passed in twice—once to return the correct plural string, and again as an input parameter to complete the sentence.

Colors

Use the `color` tag to define a new color resource. Specify the color value using a `#` symbol followed by the (optional) alpha channel, and then the red, green, and blue values using one or two hexadecimal numbers with any of the following notations:

- #RGB
- #RRGGBB
- #ARGB
- #AARRGGBB

The following example shows how to specify a fully opaque blue and a partially transparent green:

```
<color
name="opaque_blue">#00F</color>
<color
name="transparent_green">#7700FF00</color>
```

Dimensions

Dimensions are most

commonly referenced within style and layout resources. They're useful for creating layout constants, such as borders and font heights.

To specify a dimension resource, use the `dimen` tag, specifying the dimension value, followed by an identifier describing the scale of your dimension:

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

Although you can use any of these measurements to define a dimension, it's best practice to use either density- or scale-independent pixels. These alternatives let you define a dimension using relative scales that account for different screen resolutions and densities to simplify scaling on different hardware.

Scale-independent pixels are particularly well suited when defining font sizes because they automatically scale if the user changes the system font size.

The following XML snippet shows how to specify dimension values for a large font

size and a standard border:

```
<dimen
name="standard_border">5dp</dimen>
<dimen
name="large_font_size">16sp</dimen>
```

Styles and Themes

Style resources let your applications maintain a consistent look and feel by enabling you to specify the attribute values used by Views. The most common use of themes and styles is to store the colors and fonts for an application.

To create a style, use a `style` tag that includes a `name` attribute and contains one or more `item` tags. Each `item` tag should include a `name` attribute used to specify the attribute (such as font size or color) being defined. The tag itself should then contain the value, as shown in the following skeleton code.

```
<?xml
version="1.0"
encoding="utf-
8"?>
<resources>
  <style
name="base_text">
      <item
name="android:textSize">14sp</item>
      <item
name="android:textColor">#111</item>
  </style>
</resources>
```

Styles support inheritance using the parent attribute on

the style tag, making it easy to create simple variations:

```
<?xml
version="1.0"
encoding="utf-
8"?>
<resources>
  <style
name="small_text"
parent="base_text">
    <item
name="android:textSize">8sp</item>
  </style>
</resources>
```

Drawables

Drawable resources include bitmaps and NinePatches (stretchable PNG images). They also include complex composite Drawables, such as `LevelListDrawable`**s** and `StateListDrawable`**s**, that can be defined in XML.

Both NinePatch Drawables and complex composite resources are covered in more detail in the next chapter.

All Drawables are stored as

individual files in the `res/drawable` folder. Note that it's good practice to store bitmap image assets in the appropriate drawable -ldpi, -mdpi, -hdpi, and -xhdpi folders, as described earlier in this chapter. The resource identifier for a Drawable resource is the lowercase file name without its extension.

The preferred format for a bitmap resource is PNG, although JPG and GIF files are also supported.

Layouts

Layout resources enable

you to decouple your presentation layer from your business logic by designing UI layouts in XML rather than constructing them in code.

You can use layouts to define the UI for any visual component, including Activities, Fragments, and Widgets. Once defined in XML, the layout must be "inflated" into the user interface. Within an Activity this is done using `setContentView` (usually within the `onCreate` method), whereas Fragment Views are inflated using the `inflate`

method from the `Inflator` object passed in to the Fragment's `onCreateView` handler.

For more detailed information on using and creating layouts in Activities and Fragments, see Chapter 4, “Building User Interfaces.”

Using layouts to construct your screens in XML is best practice in Android. The decoupling of the layout from the code enables you to create optimized layouts for different hardware configurations, such as varying screen sizes, orientation, or the presence of

keyboards and touchscreens.

Each layout definition is stored in a separate file, each containing a single layout, in the `res/layout` folder. The filename then becomes the resource identifier.

A thorough explanation of layout containers and View elements is included in the next chapter, but as an example Listing 3.2 shows the layout created by the New Project Wizard. It uses a Linear Layout (described in more detail in Chapter 4) as a layout

container for a TextView that displays the "Hello World" greeting.

Listing 3.2: Hello World layout

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

 android:layout_width="fill_parent"

 android:layout_height="wrap_content"

 android:text="@string/hello"
  />
</LinearLayout>
```

code snippet PA4AD_Ch03_Manifest_and_Resources/res/layout/main.xml

Animations

Android supports three types of animation:

- **Property animations**—A tweened animation that can be used to potentially animate any property on the target object by applying incremental changes between two values. This can be used for anything from changing the color or opacity of a View to gradually fade it in or out, to changing a font size, or increasing a character's hit points.
- **View animations**—

Tweened animations that can be applied to rotate, move, and stretch a View.

- **Frame animations**—Frame-by-frame "cell" animations used to display a sequence of Drawable images.

A comprehensive overview of creating, using, and applying animations can be found in Chapter 11, "Advanced User Experience."

Defining animations as external resources enables you to reuse the same sequence in multiple places and provides you with the opportunity to present different animations based on device hardware or orientation.

Property Animations

Property animators were introduced in Android 3.0 (API level 11).

It is a powerful framework that can be used to animate almost anything.

Each property animation is stored in a separate XML file in the project's `res/animator` folder. As with layouts and Drawable resources, the animation's filename is used as its resource identifier.

You can use a property animator to animate almost any property on a target object. You can define animators that

are tied to a specific property, or a generic value animator that can be allocated to any property and object.

Property animators are extremely useful and are used extensively for animating Fragments in Android. You will explore them in more detail in Chapter 11.

The following simple XML snippet shows a property animator that changes the opacity of the target object by

calling its `setAlpha` method incrementally between 0 and 1 over the course of a second:

```
<?xml version="1.0"
encoding="utf-8"?>
<objectAnimator
xmlns:android="http://schemas.android.com/apk/res/android"
   android:propertyName="alpha"
   android:duration="1000"
   android:valueFrom="0.0"
   android:valueTo="1.0"
/>
```

View Animations

Each view animation is stored in a separate XML file in the project's `res/anim` folder. As with layouts and Drawable resources, the animation's filename is used as its resource identifier.

An animation can be defined for changes in `alpha` (fading), `scale` (scaling), `translate` (movement), or `rotate` (rotation).

Table 3.1 shows the valid attributes, and attribute values, supported by each animation type.

Table 3.1: Animation type attributes

Animation Type	Attributes	Valid Values
Alpha	`fromAlpha`/`toAlpha`	Float from 0 to 1
Scale	`fromXScale`/`toXScale`	Float from 0 to 1
	`fromYScale`/`toYScale`	Float from 0 to 1
	`pivotX`/`pivotY`	String of the percentage of graphic width/height from 0% to 100%
		Float from

Translate	`fromX`/`toX`	Float from 0 to 1
	`fromY`/`toY`	Float from 0 to 1
Rotate	`fromDegrees`/`toDegrees`	Float from 0 to 360
	`pivotX`/`pivotY`	String of the percentage of graphic width/height from 0% to 100%

You can create a combination of animations using the `set` tag. An animation set contains one or more animation transformations and supports various additional tags and attributes to customize when and how each animation within the set is run.

The following list shows some of the `set` tags available:

- `duration`—Duration of the full animation in milliseconds.
- `startOffset`—Millisecond delay before the animation starts.
- `fillBeforetrue`—Applies the animation transformation before it begins.
- `fillAftertrue`—Applies the animation transformation after it ends.
- `interpolator`—Sets how the speed of this effect varies over time. Chapter 11 explores the interpolators available. To specify one, reference the system animation resources at `android:anim/interpolatorName`.

The following example shows an animation set that spins the target 360 degrees while it shrinks and fades out:

If you do not use the `startOffset` tag, all the animation effects within a set will execute simultaneously.

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

android:interpolator="@android:anim/accelerate_interpolator">
  <rotate
    android:fromDegrees="0"
    android:toDegrees="360"
    android:pivotX="50%"
    android:pivotY="50%"

 android:startOffset="500"
    android:duration="1000"
/>
  <scale

 android:fromXScale="1.0"
    android:toXScale="0.0"

 android:fromYScale="1.0"
    android:toYScale="0.0"
    android:pivotX="50%"
    android:pivotY="50%"

 android:startOffset="500"
    android:duration="500"
/>
  <alpha
    android:fromAlpha="1.0"
    android:toAlpha="0.0"

 android:startOffset="500"
    android:duration="500"
/>
</set>
```

Frame-by-Frame Animations

Frame-by-frame animations produce a sequence of Drawables, each of which is displayed for a specified duration.

Because frame-by-frame animations represent animated Drawables, they are stored in the `res/drawable` folder and use their filenames (without the `.xml` extension) as their resource Ids.

The following XML snippet shows a simple animation that cycles through a series of bitmap resources, displaying each one for half a second. To use this snippet, you need to create new image resources `android1` through `android3`:

```
    <animation-list
```

```
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:oneshot="false">
  <item
android:drawable="@drawable/android1"
android:duration="500"
/>
  <item
android:drawable="@drawable/android2"
android:duration="500"
/>
  <item
android:drawable="@drawable/android3"
android:duration="500"
/>
</animation-list>
```

Note that in many cases you should include multiple resolutions of each of the drawables used within the animation list in the drawable-ldpi, -mdi, -hdpi, and -xhdpi folders, as appropriate.

To play the animation, start by assigning the resource to a host View before getting a reference to the Animation Drawable object and starting it:

```
ImageView
androidIV =
(ImageView)findViewById(R.id.iv_android);
androidIV.setBackgroundResource(R.drawable.android_anim);

AnimationDrawable
androidAnimation
=
  (AnimationDrawable)
androidIV.getBackground();

androidAnimation.start();
```

Typically, this is done in two steps; assigning the resource to the background should be done within the `onCreate` handler.

Within this handler the animation is not fully attached to the window, so the animations can't be started; instead, this is

usually done as a result to user action (such as a button press) or within the `onWindowFocusChanged` handler.

Menus

Create menu resources to design your menu layouts in XML, rather than constructing them in code.

You can use menu resources to define both Activity and context menus within your applications, and provide the same options you would have when constructing your menus in code. When defined in XML, a menu is inflated within your application via the `inflate` method of the `MenuInflator` Service, usually within the `onCreateOptionsMenu` method. You examine menus in more detail in Chapter 10.

Each menu definition is stored in a separate file, each containing a single menu, in the `res/menu`

folder—the filename then becomes the resource identifier. Using XML to define your menus is best-practice design in Android.

A thorough explanation of menu options is included in Chapter 10, but Listing 3.3 shows a simple example.

Listing 3.3: Simple menu layout resource

```
<?xml
version="1.0"
encoding="utf-
8"?>
<menu
xmlns:android="http://schemas.android.com/apk/res/android">
  <item
android:id="@+id/menu_refresh"

 android:title="@string/refresh_mi"
/>
  <item
android:id="@+id/menu_settings"

 android:title="@string/settings_mi"
/>
</menu>
```

code snippet PA4AD_Snippets_Chapter3/res/menu/menu.xml

Using Resources

In addition to the resources you supply, the Android platform includes several system resources that you can use in your applications. All resources can be used directly from your application code and can also be referenced from within other resources. For example, a dimension resource might be referenced in a layout definition.

Later in this chapter you learn how to define alternative resource values for different languages, locations, and hardware. It's important to note that when using resources, you shouldn't choose a particular specialized version. Android will automatically select the most appropriate value for a given resource identifier based on the current hardware, device, and language configurations.

Using Resources in Code

Access resources in code using the static `R` class. `R` is a generated class based on your external resources, and created when your project is compiled. The `R` class contains static subclasses for each of the resource types for which you've defined at least one resource. For example, the default new project includes the `R.string` and `R.drawable` subclasses.

If you use the ADT plug-in in Eclipse, the `R` class will be created automatically when you make any change to an external resource file or folder. If you are not using the plug-in, use the AAPT tool to compile your project and generate the `R` class. `R` is a compiler-generated class, so don't make any manual modifications to it because they will be lost when the file is regenerated.

Each of the subclasses within `R` exposes its associated resources as variables, with the variable names matching the resource identifiers—for example, `R.string.app_name` or `R.drawable.icon`.

The value of these variables is an integer that represents each resource's location in the resource table, *not* an instance of the resource itself.

Where a constructor or method, such as `setContentView`, accepts a resource identifier, you can pass in the resource variable, as shown in the following code snippet:

```
// Inflate a layout resource.
setContentView(R.layout.main);
// Display a transient dialog box that displays the
// error message string resource.
Toast.makeText(this, R.string.app_error,
Toast.LENGTH_LONG).show();
```

When you need an instance of the resource itself, you need to use helper methods to extract them from the resource table. The resource table is represented within your application as an instance of the `Resources` class.

These methods perform lookups on the application's current resource table, so these helper methods can't be static. Use the `getResources` method on your application context, as shown in the following snippet, to access your application's `Resources` instance:

```
Resources myResources = getResources();
```

The `Resources` class includes getters for each of the available resource types and generally works by passing in the resource ID you'd like an instance of. The following code snippet shows an example of using the helper methods to return a selection of resource values:

```
Resources myResources = getResources();

CharSequence styledText =
myResources.getText(R.string.stop_message);
Drawable icon =
myResources.getDrawable(R.drawable.app_icon);

int opaqueBlue =
myResources.getColor(R.color.opaque_blue);

float borderWidth =
myResources.getDimension(R.dimen.standard_border);

Animation tranOut;
tranOut = AnimationUtils.loadAnimation(this,
R.anim.spin_shrink_fade);
```

```
ObjectAnimator animator =
  (ObjectAnimator)AnimatorInflater.loadAnimator(this,
  R.anim.my_animator);

String[] stringArray;
stringArray =
myResources.getStringArray(R.array.string_array);

int[] intArray =
myResources.getIntArray(R.array.integer_array);
```

Frame-by-frame animated resources are inflated into `AnimationResources`. You can return the value using `getDrawable` and casting the return value, as shown here:

```
AnimationDrawable androidAnimation;
androidAnimation =
  (AnimationDrawable)myResources.getDrawable(R.drawable.frame_by_frame);
```

Referencing Resources Within Resources

You can also use resource references as attribute values in other XML resources.

This is particularly useful for layouts and styles, letting you create specialized variations on themes and localized strings and image assets. It's also a useful way to support different images and spacing for a layout to ensure that it's optimized for different screen sizes and resolutions.

To reference one resource from another, use the `@` notation, as shown in the following snippet:

```
attribute="@[packagename:]resourcetype/resourceidentifier"
```

Android assumes you use a resource from the same package, so you only need to fully qualify the package name if you use a resource from a different

package.

Listing 3.4 shows a layout that uses color, dimension, and string resources.

Listing 3.4: Using resources in a layout

```
<?xml version="1.0" encoding="utf-
8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="@dimen/standard_border">
  <EditText
    android:id="@+id/myEditText"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"

 android:text="@string/stop_message"

 android:textColor="@color/opaque_blue"
  />
</LinearLayout>
```

code snippet PA4AD_Ch03_Manifest_and_Resources/res/layout/reslayout.xml

Using System Resources

The Android framework makes many native resources available, providing you with various strings, images, animations, styles, and layouts to use in your applications.

Accessing the system resources in code is similar to

using your own resources. The difference is that you use the native Android resource classes available from `android.R`, rather than the application-specific `R` class. The following code snippet uses the `getString` method available in the application context to retrieve an error message available from the system resources:

```
CharSequence httpError =
getString(android.R.string.httpErrorBadUrl);
```

To access system resources in XML, specify `android` as the package name, as shown in this XML snippet:

```
<EditText
  android:id="@+id/myEditText"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@android:string/httpErrorBadUrl"
  android:textColor="@android:color/darker_gray"
/>
```

Referring to Styles in the Current Theme

Using themes is an excellent way to ensure consistency for your application's UI. Rather than fully define each style, Android provides a shortcut to enable you to use styles from the

currently applied theme.

To do this, use `?android:` rather than `@` as a prefix to the resource you want to use. The following example shows a snippet of the preceding code but uses the current theme's text color rather than a system resource:

```
<EditText
  android:id="@+id/myEditText"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="@android:string/httpErrorBadUrl"
  android:textColor="?
android:textColor"
/>
```

This technique enables you to create styles that change if the current theme changes, without you modifying each individual style resource.

Creating Resources for Different Languages and Hardware

Using the directory structure described here, you can create different resource values for specific languages, locations, and hardware configurations. Android chooses from among these values dynamically at run time using its dynamic resource-selection mechanism.

You can specify alternative resource values using a parallel directory structure within the `res` folder. A hyphen (-) is used to separate qualifiers that specify the conditions you provide alternatives for.

The following example hierarchy shows a folder structure that features default string values, with French language and French Canadian location variations:

```
Project/
  res/
    values/
      strings.xml
    values-fr/
      strings.xml
    values-fr-rCA/
      strings.xml
```

The following list gives the qualifiers you can use to customize your resource values:

- **Mobile Country Code and Mobile Network Code (MCC/MNC)**—The country, and optionally

the network, associated with the SIM currently used in the device. The MCC is specified by `mcc` followed by the three-digit country code. You can optionally add the MNC using `mnc` and the two- or three-digit network code (for example, `mcc234-mnc20` or `mcc310`). You can find a list of MCC/MNC codes on Wikipedia at http://en.wikipedia.org/wiki/MobileNetworkCode.

- **Language and Region**—Language specified by the lowercase two-letter ISO 639-1 language code, followed optionally by a region specified by a lowercase `r` followed by the uppercase two-letter ISO 3166-1-alpha-2 language code (for example, `en`, `en-rUS`, or `en-rGB`).
- **Smallest Screen Width**—The lowest of the device's screen dimensions (height and width) specified in the form `sw<Dimension value>dp` (for example, `sw600dp`, `sw320dp`, or `sw720dp`). This is generally used when providing multiple layouts, where the value specified should be the smallest screen width that your layout requires in order to render correctly. Where you supply multiple directories with different smallest screen width qualifiers, Android selects the largest value that doesn't exceed the smallest dimension available on the device.
- **Available Screen Width**—The minimum screen width required to use the contained resources,

specified in the form `w<Dimension value>dp` (for example, `w600dp`, `w320dp`, or `w720dp`). Also used to supply multiple layouts alternatives, but unlike smallest screen width, the available screen width changes to reflect the current screen width when the device orientation changes. Android selects the largest value that doesn't exceed the currently available screen width.

- **Available Screen Height**—The minimum screen height required to use the contained resources, specified in the form `h<Dimension value>dp` (for example, `h720dp`, `h480dp`, or `h1280dp`). Like available screen width, the available screen height changes when the device orientation changes to reflect the current screen height. Android selects the largest value that doesn't exceed the currently available screen height.
- **Screen Size**—One of `small` (smaller than HVGA), `medium` (at least HVGA and typically smaller than VGA), `large` (VGA or larger), or `xlarge` (significantly larger than HVGA). Because each of these screen categories can include devices with significantly different screen sizes (particularly tablets), it's good practice to use the more specific smallest screen size, and available screen width and height whenever possible. Because they precede this screen size

qualifier, where both are specified, the more specific qualifiers will be used in preference where supported.

- **Screen Aspect Ratio**—Specify `long` or `notlong` for resources designed specifically for wide screen. (For example, WVGA is `long`; QVGA is `notlong`.)
- **Screen Orientation:** One of `port` (portrait), `land` (landscape), or `square` (square).
- **Dock Mode**—One of `car` or `desk`. Introduced in API level 8.
- **Night Mode**—One of `night` (night mode) or `notnight` (day mode). Introduced in API level 8. Used in combination with the dock mode qualifier, this provides a simple way to change the theme and/or color scheme of an application to make it more suitable for use at night in a car dock.
- **Screen Pixel Density**—Pixel density in dots per inch (dpi). Best practice is to supply `ldpi`, `mdpi`, `hdpi`, or `xhdpi` to specify low (120 dpi), medium (160 dpi), high (240 dpi), or extra high (320 dpi) pixel density assets, respectively. You can specify `nodpi` for bitmap resources you don't want scaled to support an exact screen density. To better support applications targeting televisions running Android, you can also use the `tvdpi` qualifier for assets of approximately

213dpi. This is generally unnecessary for most applications, where including medium- and high-resolution assets is sufficient for a good user experience. Unlike with other resource types, Android does not require an exact match to select a resource. When selecting the appropriate folder, it chooses the nearest match to the device's pixel density and scales the resulting Drawables accordingly.

- **Touchscreen Type**—One of `notouch`, `stylus`, or `finger`, allowing you to provide layouts or dimensions optimized for the style of touchscreen input available on the host device.
- **Keyboard Availability**—One of `keysexposed`, `keyshidden`, or `keyssoft`.
- **Keyboard Input Type**—One of `nokeys`, `qwerty`, or `12key`.
- **Navigation Key Availability**—One of `navexposed` or `navhidden`.
- **UI Navigation Type**—One of `nonav`, `dpad`, `trackball`, or `wheel`.
- **Platform Version**—The target API level, specified in the form `v<API Level>` (for example, `v7`). Used for resources restricted to devices running at the specified API level or higher.

You can specify multiple qualifiers for any resource type, separating each qualifier with a hyphen. Any

combination is supported; however, they must be used in the order given in the preceding list, and no more than one value can be used per qualifier.

The following example shows valid and invalid directory names for alternative layout resources.

```
Valid
    layout-large-land
    layout-xlarge-port-keyshidden
    layout-long-land-notouch-nokeys


Invalid
    values-rUS-en (out of order)
    values-rUS-rUK (multiple values for a single
qualifier)
```

When Android retrieves a resource at run time, it finds the best match from the available alternatives. Starting with a list of all the folders in which the required value exists, it selects the one with the greatest number of matching qualifiers. If two folders are an equal match, the tiebreaker is based on the order of the matched qualifiers in the preceding list.

If no resource matches are found on a given device, your application throws an exception when attempting to access that resource. To avoid this, you should always include default values for each resource type in a folder that includes no qualifiers.

Runtime Configuration Changes

Android handles runtime changes to the language, location, and hardware by terminating and restarting the active Activity. This forces the resource resolution for the Activity to be reevaluated and the most appropriate resource values for the new configuration to be selected.

In some special cases this default behavior may be inconvenient, particularly for applications that don't want to present a different UI based on screen orientation changes. You can customize your application's response to such changes by detecting and reacting to them yourself.

To have an Activity listen for runtime configuration changes, add an `android:configChanges` attribute to its manifest node, specifying the configuration changes you want to handle.

The following list describes some of the configuration changes you can specify:

- `mcc` and `mnc`—A SIM has been detected and the mobile country or network code (respectively) has changed.
- `locale`—The user has changed the device's language settings.
- `keyboardHidden`—The keyboard, d-pad, or other input mechanism has been exposed or hidden.

- `keyboard`—The type of keyboard has changed; for example, the phone may have a 12-key keypad that flips out to reveal a full keyboard, or an external keyboard might have been plugged in.
- `fontScale`—The user has changed the preferred font size.
- `uiMode`—The global UI mode has changed. This typically occurs if you switch between car mode, day or night mode, and so on.
- `orientation`—The screen has been rotated between portrait and landscape.
- `screenLayout`—The screen layout has changed; typically occurs if a different screen has been activated.
- `screenSize`—Introduced in Honeycomb MR2 (API level 12), occurs when the available screen size has changed, for example a change in orientation between landscape and portrait.
- `smallestScreenSize`—Introduced in Honeycomb MR2 (API level 12), occurs when the physical screen size has changed, such as when a device has been connected to an external display.

In certain circumstances multiple events will be triggered simultaneously. For example, when the user slides out a keyboard, most devices fire both the `keyboardHidden` and `orientation` events, and connecting an external display on a post-Honeycomb MR2 device is likely to trigger

`orientation`, `screenLayout`, `screenSize`, and `smallestScreenSize` events.

You can select multiple events you want to handle yourself by separating the values with a pipe (`|`), as shown in Listing 3.5, which shows an activity node declaring that it will handle changes in screen size and orientation, and keyboard visibility.

Listing 3.5: Activity definition for handling dynamic resource changes

```
<activity
  android:name=".MyActivity"
  android:label="@string/app_name"
  android:configChanges="screenSize|orientation|keyboardHidden">
  <intent-filter >
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
```

code snippet PA4AD_Ch03_Config_Changes/AndroidManifest.xml

Adding an `android:configChanges` attribute suppresses the restart for the specified configuration changes, instead triggering the `onConfigurationChanged` handler in the associated Activity. Override this method to handle the configuration changes yourself, using the passed-in

`Configuration` object to determine the new configuration values, as shown in Listing 3.6. Be sure to call back to the superclass and reload any resource values that the Activity uses, in case they've changed.

Listing 3.6: Handling configuration changes in code

```
@Override
public void onConfigurationChanged(Configuration
newConfig) {
  super.onConfigurationChanged(newConfig);

  // [ ... Update any UI based on resource values ... ]

  if (newConfig.orientation ==
Configuration.ORIENTATION_LANDSCAPE) {
    // [ ... React to different orientation ... ]
  }

  if (newConfig.keyboardHidden ==
Configuration.KEYBOARDHIDDEN_NO) {
    // [ ... React to changed keyboard visibility ... ]
  }
}
```

code snippet PA4AD_Ch03_Config_Changes/src/MyActivity.java

When `onConfigurationChanged` is called, the Activity's Resource variables have already been updated with the new values, so they'll be safe to use.

Any configuration change that you don't explicitly flag as being handled by your application will cause your Activity to restart, without a call to `onConfigurationChanged`.

The Android Application Lifecycle

Unlike many traditional application platforms, Android applications have limited control over their own lifecycles. Instead, application components must listen for changes in the application state and react accordingly, taking particular care to be prepared for untimely termination.

By default, each Android application runs in its own process, each of which is running a separate instance of Dalvik. Memory and process management is handled exclusively by the run time.

You can force application components within the same application to run in different processes or to have multiple applications share the same process using the `android:process` attribute on the affected component nodes within the manifest.

Android aggressively manages its resources, doing whatever's necessary to ensure a smooth and stable user experience. In practice that means that processes (and their hosted applications) will be killed, in some case without warning, to free resources for higher-priority applications.

Understanding an Application's Priority and Its Process' States

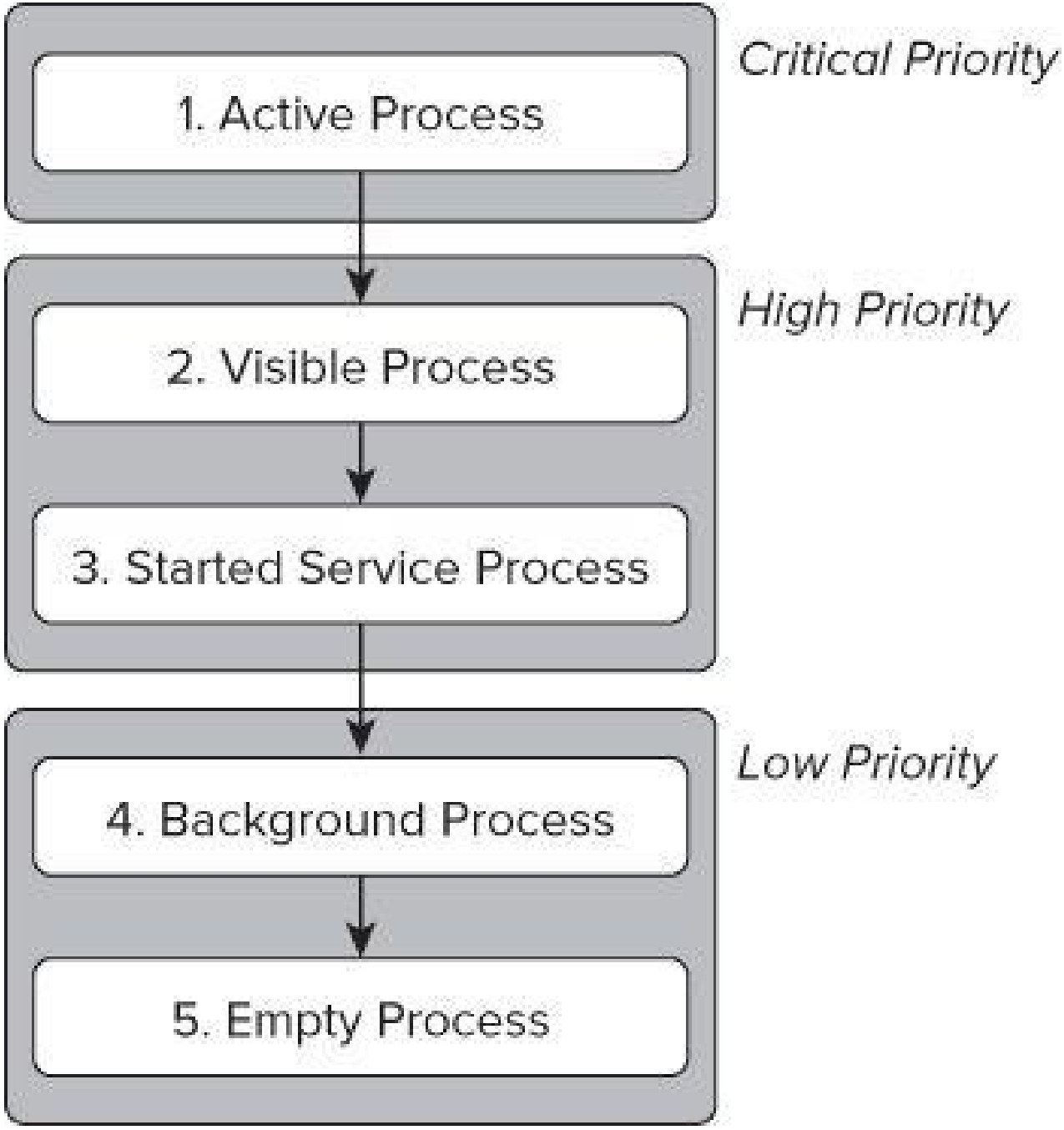
The order in which processes are killed to reclaim resources is determined by the priority of their hosted applications. An application's priority is equal to that of its highest-priority component.

If two applications have the same priority, the process that has been at that priority longest will be killed first. Process priority is also affected by interprocess dependencies; if an application has a dependency on a Service or Content Provider supplied by a second application, the secondary application has at least as high a priority as the application it supports.

All Android applications continue running and in memory until the system needs resources for other applications.

Figure 3.4 shows the priority tree used to determine the order of application termination.

Figure 3.4



It's important to structure your application to ensure that its priority is appropriate for the work it's doing. If you don't, your application could be killed while it's in the middle of something important, or it could remain running when it is no longer needed.

The following list details each of the application states shown in Figure 3.4, explaining how the state is determined by the application components of which it comprises:

- **Active processes**—Active (foreground) processes have application components the user is interacting with. These are the processes Android tries to keep responsive by reclaiming resources from other applications. There are generally very few of these processes, and they will be killed only as a last resort.
- Active processes include the following:
    - Activities in an active state—that is, those in the foreground responding to user events. You will explore Activity states in greater detail later in this chapter.
    - Broadcast Receivers executing `onReceive` event handlers as described in Chapter 5.
    - Services executing `onStart`, `onCreate`, or `onDestroy` event handlers as described in Chapter 9.
    - Running Services that have been flagged to run in the foreground (also described in Chapter 9.)
- **Visible processes**—Visible but inactive processes are those hosting “visible” Activities. As the name suggests, visible Activities are visible, but they aren't in the foreground or responding to user events. This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity). There are generally very few visible processes, and they'll be killed only under extreme circumstances to allow active processes to continue.

- **Started Service processes**—Processes hosting Services that have been started. Because these Services don't interact directly with the user, they receive a slightly lower priority than visible Activities or foreground Services. Applications with running Services are still considered foreground processes and won't be killed unless resources are needed for active or visible processes. When the system terminates a running Service it will attempt to restart them (unless you specify that it shouldn't) when resources become available. You'll learn more about Services in Chapter 9.
- **Background processes**—Processes hosting Activities that aren't visible and that don't have any running Services. There will generally be a large number of background processes that Android will kill using a last-seen-first-killed pattern in order to obtain resources for foreground processes.
- **Empty processes**—To improve overall system performance, Android will often retain an application in memory after it has reached the end of its lifetime. Android maintains this cache to improve the start-up time of applications when they're relaunched. These processes are routinely killed, as required.

Introducing the Android Application Class

Your application's `Application` object remains instantiated whenever your application runs. Unlike Activities, the Application is not restarted as a result of configuration changes. Extending the `Application` class with your own implementation enables you to do three things:

- Respond to application level events broadcast by the Android run time such as low memory conditions.
- Transfer objects between application components.
- Manage and maintain resources used by several application components.

Of these, the latter two can be better achieved using a separate singleton class. When your `Application` implementation is registered in the manifest, it will be instantiated when your application process is created. As a result, your `Application` implementation is by nature a singleton and should be implemented as such to provide access to its methods and member variables.

Extending and Using the Application Class

Listing 3.7 shows the skeleton code for extending the `Application` class and implementing it as a singleton.

Listing 3.7: Skeleton Application class

```
import android.app.Application;
import android.content.res.Configuration;

public class MyApplication extends Application {

  private static MyApplication singleton;

  // Returns the application instance
  public static MyApplication getInstance() {
    return singleton;
  }

  @Override
  public final void onCreate() {
    super.onCreate();
    singleton = this;
  }
}
```

code snippet PA4AD_Ch03_Config_Changes/src/MyApplication.java

When created, you must register your new `Application` class in the manifest's `application` node using a `name` attribute, as shown in the following snippet:

```
<application android:icon="@drawable/icon"
             android:name=".MyApplication">
  [... Manifest nodes ...]
```

```
</application>
```

Your `Application` implementation will be instantiated when your application is started. Create new state variables and global resources for access from within the application components:

```
MyObject value =
MyApplication.getInstance().getGlobalStateValue();
MyApplication.getInstance().setGlobalStateValue(myObjectValue);
```

Although this can be an effective technique for transferring objects between your loosely coupled application components, or for maintaining application state or shared resources, it is often better to create your own static singleton class rather than extending the `Application` class specifically unless you are also handling the lifecycle events described in the following section.

Overriding the Application Lifecycle Events

The `Application` class provides event handlers for application creation and termination, low memory conditions, and configuration changes (as described in the previous section).

By overriding these methods, you can implement your own application-specific behavior for each of these circumstances:

- `onCreate`—Called when the application is created. Override this method to initialize your application singleton and create and initialize any application state variables or shared resources.
- `onLowMemory`—Provides an opportunity for well-behaved applications to free additional memory when the system is running low on resources. This will generally only be called when background processes have already been terminated and the current foreground applications are still low on memory. Override this handler to clear caches or release unnecessary resources.
- `onTrimMemory`—An application specific alternative to the `onLowMemory` handler introduced in Android 4.0 (API level 13). Called when the run time determines

that the current application should attempt to trim its memory overhead – typically when it moves to the background. It includes a level parameter that provides the context around the request.

- `onConfigurationChanged`—Unlike Activities `Application` objects are not restarted due to configuration changes. If your application uses values dependent on specific configurations, override this handler to reload those values and otherwise handle configuration changes at an application level.

As shown in Listing 3.8, you must always call through to the superclass event handlers when overriding these methods.

Listing 3.8: Overriding the Application Lifecycle Handlers

```
public class MyApplication extends Application {

  private static MyApplication singleton;

  // Returns the application instance
  public static MyApplication getInstance() {
    return singleton;
  }

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

```
    singleton = this;
  }

  @Override
  public final void onLowMemory() {
    super.onLowMemory();
  }

  @Override
  public final void onTrimMemory(int level) {
    super.onTrimMemory(level);
  }

  @Override
  public final void onConfigurationChanged(Configuration
newConfig) {
    super.onConfigurationChanged(newConfig);
  }
}
```

code snippet PA4AD_Snippets_Chapter3/MyApplication.java

A Closer Look at Android Activities

Each Activity represents a screen that an application can present to its users. The more complicated your application, the more screens you are likely to need.

Typically, this includes at least a primary interface screen that handles the main UI functionality of your application. This primary interface generally consists of a number of Fragments that make up your UI and is generally supported by a set of secondary Activities. To move between screens you start a new Activity (or return from one).

Most Activities are designed to occupy the entire display, but you can also create semitransparent or floating Activities.

Creating Activities

Extend `Activity` to create a new `Activity` class. Within this new class you must define the UI and implement your functionality. Listing 3.9 shows the basic skeleton code for a new `Activity`.

Listing 3.9: Activity skeleton code

```
package com.paad.activities;

import android.app.Activity;
import android.os.Bundle;

public class MyActivity extends Activity {

  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
  }
}
```

code snippet PA4AD_Ch03_Activities/src/MyActivity.java

The base `Activity` class presents an empty screen that encapsulates the window display handling. An empty `Activity` isn't particularly useful, so the first thing you'll want to do is create the UI with Fragments, layouts, and Views.

Views are the UI controls that display data and

provide user interaction. Android provides several layout classes, called *View Groups*, which can contain multiple Views to help you layout your UIs. Fragments are used to encapsulate segments of your UI, making it simple to create dynamic interfaces that can be rearranged to optimize your layouts for different screen sizes and orientations.

Chapter 4 discusses Views, View Groups, layouts, and Fragments in detail, examining what's available, how to use them, and how to create your own.

To assign a UI to an Activity, call `setContentView` from the `onCreate` method of your `Activity`.

In this first snippet, an instance of a `TextView` is used as the `Activity`'s UI:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  TextView textView = new TextView(this);
  setContentView(textView);
}
```

Usually, you'll want to use a more complex UI design. You can create a layout in code using layout View Groups, or you can use the standard Android convention of passing a resource ID for a layout defined in an external resource, as shown in the following snippet:

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

To use an `Activity` in your application, you need to register it in the manifest. Add a new `activity` tag within the `application` node of the manifest; the `activity` tag includes attributes for metadata, such as the label, icon, required permissions, and themes used by the `Activity`. An `Activity` without a corresponding `activity` tag can't be displayed—attempting to do so will result in a runtime exception.

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

Within the `activity` tag you can add `intent-filter` nodes that specify the Intents that can be used to start your Activity. Each Intent Filter defines one or more actions and categories that your `Activity` supports. Intents and Intent Filters are covered in depth in Chapter 5, but it's worth noting that for an `Activity` to be available from the application launcher, it must include an Intent Filter listening for the `MAIN` action and the `LAUNCHER` category, as highlighted in Listing 3.10.

Listing 3.10: Main Application Activity Definition

```
<activity
android:label="@string/app_name"
          android:name=".MyActivity">
  <intent-filter>
    <action
android:name="android.intent.action.MAIN"
/>
    <category
android:name="android.intent.category.LAUNCHER"
/>
  </intent-filter>
</activity>
```

code snippet PA4AD_Ch03_Activities/AndroidManifest.xml

The Activity Lifecycle

A good understanding of the Activity lifecycle is vital to ensure that your application provides a seamless user experience and properly manages its resources.

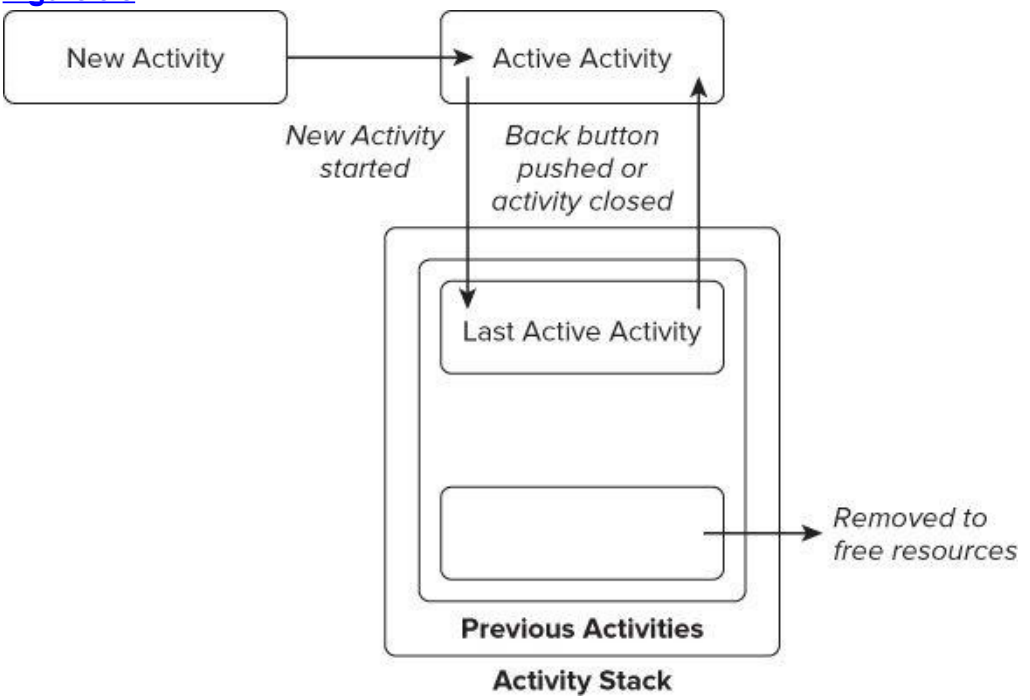
As explained earlier, Android applications do not control their own process lifetimes; the Android run time manages the process of each application, and by extension that of each Activity within it.

Although the run time handles the termination and management of an Activity's process, the Activity's state helps determine the priority of its parent application. The application priority, in turn, influences the likelihood that the run time will terminate it and the Activities running within it.

Activity Stacks

The state of each Activity is determined by its position on the Activity stack, a last-in–first-out collection of all the currently running Activities. When a new Activity starts, it becomes active and is moved to the top of the stack. If the user navigates back using the Back button, or the foreground Activity is otherwise closed, the next Activity down on the stack moves up and becomes active. Figure 3.5 illustrates this process.

Figure 3.5



As described previously in this chapter, an application's priority is influenced by its highest-priority Activity. When the Android memory manager is deciding which application to terminate to free resources, it uses this Activity stack to determine the priority of applications.

Activity States

As Activities are created and destroyed, they move in and out of the stack, as shown in Figure 3.5. As they do so, they transition through four possible states:

- **Active**—When an Activity is at the top of the stack it is the visible, focused, foreground Activity that is receiving user input. Android will attempt to keep it alive at all costs, killing Activities further down the stack as needed, to ensure that it has the resources it needs. When another Activity becomes active, this one will be paused.
- **Paused**—In some cases your Activity will be visible but will not have focus; at this point it's paused. This state is reached if a transparent or non-full-screen Activity is active in front of it. When paused, an Activity is treated as if it were active; however, it doesn't receive user input events. In extreme cases Android will kill a paused Activity to recover resources for the active Activity. When an Activity becomes totally obscured, it is stopped.
- **Stopped**—When an Activity isn't visible, it "stops." The Activity will remain in memory, retaining all state information; however, it is now a candidate for termination when the system requires memory elsewhere. When an Activity is in a stopped state, it's important to save data and the current UI state, and to stop any non-critical operations. Once an Activity has exited or closed, it becomes inactive.
- **Inactive**—After an Activity has been killed, and before it's been launched, it's inactive. Inactive Activities have been removed from the Activity stack and need to be restarted before they can be displayed and used.

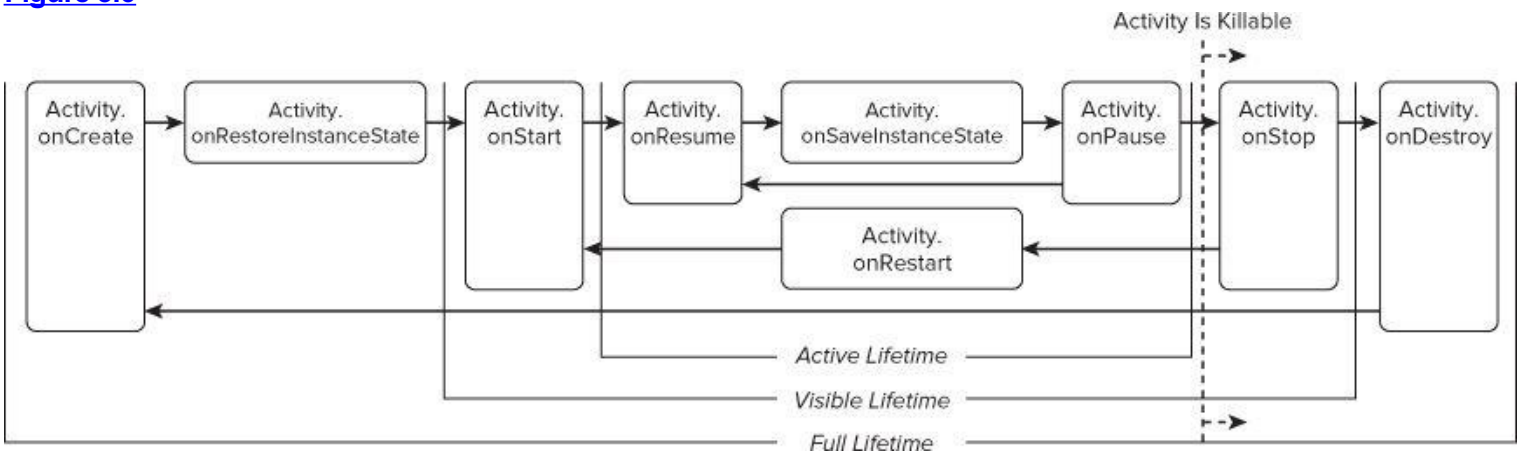
State transitions are nondeterministic and are handled entirely by the Android memory manager. Android will start by closing applications that contain inactive Activities, followed by those that are stopped. In extreme cases, it will remove those that are paused.

To ensure a seamless user experience, transitions between states should be invisible to the user. There should be no difference in an Activity moving from a paused, stopped, or inactive state back to active, so it's important to save all UI state and persist all data when an Activity is paused or stopped. Once an Activity does become active, it should restore those saved values.
Similarly, apart from changes to the Activity's priority, transitions between the active, paused, and stopped states have little direct impact on the Activity itself. It's up to you to use these signals to pause and stop your Activities accordingly

Monitoring State Changes

To ensure that Activities can react to state changes, Android provides a series of event handlers that are fired when an Activity transitions through its full, visible, and active lifetimes. Figure 3.6 summarizes these lifetimes in terms of the Activity states described in the previous section.

Figure 3.6



The skeleton code in Listing 3.11 shows the stubs for the state change method handlers available in an Activity. Comments within each stub describe the actions you should consider taking on each state change event.

Listing 3.11: Activity state event handlers

```
package com.paad.activities;
import android.app.Activity;
import android.os.Bundle;

public class MyStateChangeActivity extends Activity {

  // Called at the start of the full lifetime.
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Initialize Activity and inflate the UI.
  }

  // Called after onCreate has finished, use to restore UI
state
  @Override
  public void onRestoreInstanceState(Bundle
savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    // Restore UI state from the savedInstanceState.
    // This bundle has also been passed to onCreate.
    // Will only be called if the Activity has been
    // killed by the system since it was last visible.
  }

  // Called before subsequent visible lifetimes
  // for an Activity process.
  @Override
  public void onRestart(){
    super.onRestart();
    // Load changes knowing that the Activity has already
    // been visible within this process.
  }

  // Called at the start of the visible lifetime.
  @Override
  public void onStart(){
    super.onStart();
    // Apply any required UI change now that the Activity is
visible.
  }

  // Called at the start of the active lifetime.
  @Override
  public void onResume(){
    super.onResume();
    // Resume any paused UI updates, threads, or processes
required
    // by the Activity but suspended when it was inactive.
  }

  // Called to save UI state changes at the
  // end of the active lifecycle.
  @Override
```

```
  public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save UI state changes to the savedInstanceState.
    // This bundle will be passed to onCreate and
    // onRestoreInstanceState if the process is
    // killed and restarted by the run time.
    super.onSaveInstanceState(savedInstanceState);
  }

  // Called at the end of the active lifetime.
  @Override
  public void onPause(){
    // Suspend UI updates, threads, or CPU intensive processes
    // that don't need to be updated when the Activity isn't
    // the active foreground Activity.
    super.onPause();
  }

  // Called at the end of the visible lifetime.
  @Override
  public void onStop(){
    // Suspend remaining UI updates, threads, or processing
    // that aren't required when the Activity isn't visible.
    // Persist all edits or state changes
    // as after this call the process is likely to be killed.
    super.onStop();
  }

  // Sometimes called at the end of the full lifetime.
  @Override
  public void onDestroy(){
    // Clean up any resources including ending threads,
    // closing database connections etc.
    super.onDestroy();
  }
}
```

code snippet
PA4AD_Ch03_Activities/src/MyStateChangeActivity.java

As shown in the preceding code, you should always call back to the superclass when overriding these event handlers.

Understanding Activity Lifetimes

Within an Activity's full lifetime, between creation and destruction, it goes through one or more iterations of the active and visible lifetimes. Each transition triggers the method handlers previously described. The following sections provide a closer look at each of these lifetimes and the events that bracket them.

The Full Lifetime

The full lifetime of your Activity occurs between the first call to `onCreate` and the final call to `onDestroy`. It's not uncommon for an Activity's process to be terminated *without* the `onDestroy` method being called.

Use the `onCreate` method to initialize your Activity: inflate the user interface, get references to Fragments, allocate references to class variables, bind data to controls, and start Services and Timers. If the Activity was terminated unexpectedly by the runtime, the `onCreate` method is passed a `Bundle` object containing the state saved in the last call to `onSaveInstanceState`. You should use this `Bundle` to restore the UI to its previous state, either within the `onCreate` method or `onRestoreInstanceState`.

Override `onDestroy` to clean up any resources created in `onCreate`, and ensure that all external connections, such as network or database links, are

closed.

As part of Android's guidelines for writing efficient code, it's recommended that you avoid the creation of short-term objects. The rapid creation and destruction of objects force additional garbage collection, a process that can have a direct negative impact on the user experience. If your Activity creates the same set of objects regularly, consider creating them in the `onCreate` method instead, as it's called only once in the Activity's lifetime.

The Visible Lifetime

An Activity's visible lifetimes are bound between calls to `onStart` and `onStop`. Between these calls your Activity will be visible to the user, although it may not have focus and may be partially obscured. Activities are likely to go through several visible lifetimes during their full lifetime because they move between the foreground and background. Although it's unusual, in extreme cases the Android run time will kill an Activity during its visible lifetime without a call to `onStop`.

The `onStop` method should be used to pause or stop animations, threads, Sensor listeners, GPS lookups, Timers, Services, or other processes that are used exclusively to update the UI. There's little value in consuming resources (such as CPU cycles or network bandwidth) to update the UI when it isn't visible. Use the `onStart` (or `onRestart`) methods to resume or restart these processes when the UI is visible again.

The `onRestart` method is called immediately prior to all but the first call to `onStart`. Use it to implement special processing that you want done only when the Activity restarts within its full lifetime.

The `onStart`/`onStop` methods are also used to register and unregister Broadcast Receivers used exclusively to update the UI.

You'll learn more about using Broadcast Receivers in Chapter 5.

The Active Lifetime

The active lifetime starts with a call to `onResume` and ends with a corresponding call to `onPause`.

An active Activity is in the foreground and is receiving user input events. Your Activity is likely to go through many active lifetimes before it's destroyed, as the active lifetime will end when a new Activity is displayed, the device goes to sleep, or the Activity loses focus. Try to keep code in the `onPause` and `onResume` methods relatively fast and lightweight to ensure that your application remains responsive when moving in and out of the foreground.

Immediately before `onPause`, a call is made to

`onSaveInstanceState`. This method provides an opportunity to save the Activity's UI state in a `Bundle` that may be passed to the `onCreate` and `onRestoreInstanceState` methods. Use `onSaveInstanceState` to save the UI state (such as checkbox states, user focus, and entered but uncommitted user input) to ensure that the Activity can present the same UI when it next becomes active. You can safely assume that during the active lifetime `onSaveInstanceState` and `onPause` will be called before the process is terminated.

Most Activity implementations will override at least the `onSaveInstanceState` method to commit unsaved changes, as it marks the point beyond which an Activity may be killed without warning. Depending on your application architecture you may also choose to suspend threads, processes, or Broadcast Receivers while your Activity is not in the foreground.

The `onResume` method can be lightweight. You do not need to reload the UI state here because this is handled by the `onCreate` and `onRestoreInstanceState` methods when required. Use `onResume` to reregister any Broadcast Receivers or other processes you may have suspended in `onPause`.

Android Activity Classes

The Android SDK includes a selection of `Activity` subclasses that wrap up the use of common UI widgets. Some of the more useful ones are listed here:

- `MapActivity`—Encapsulates the resource handling required to support a `MapView` widget within an Activity. Learn more about `MapActivity` and `MapView` in Chapter 13.
- `ListActivity`—Wrapper class for Activities that feature a `ListView` bound to a data source as the primary UI metaphor, and expose event handlers for list item selection.
- `ExpandableListActivity`—Similar to the `ListActivity` but supports an `ExpandableListView`.

Chapter 4

Understanding Fragments

What's in this Chapter?

- Using Views and layouts
- Optimizing layouts
- Creating resolution-independent user interfaces
- Extending, grouping, creating, and using Views
- Using Adapters to bind data to Views

To quote Stephen Fry on the role of style as part of substance in the design of digital devices:

> *As if a device can function if it has no style. As if a device can be called stylish that does not function superbly…. Yes, beauty matters. Boy, does it matter. It is not surface, it is not an extra, it is the thing itself.*
>
> —*Stephen Fry*, *The Guardian* (*October 27*, *2007*)

Although Fry was describing the style of the devices themselves, the same can be said of the applications that run on them. Bigger, brighter, and higher resolution

displays with multitouch support have made applications increasingly visual. The introduction of devices optimized for a more immersive experience—including tablets and televisions—into the Android ecosystem has only served to increase the importance of an application's visual design.

In this chapter you'll discover the Android components used to create UIs. You'll learn how to use layouts, Fragments, and Views to create functional and intuitive UIs for your Activities.

The individual elements of an Android UI are arranged on screen by means of a variety of Layout Managers derived from the `ViewGroup` class. This chapter introduces several native layout classes and demonstrates how to use them, how to create your own, and how to ensure your use of layouts is as efficient as possible.

The range of screen sizes and display resolutions your application may be used on has expanded along with the range of Android devices now available to buy. Android 3.0 introduced the Fragment API to provide better support for creating dynamic layouts that can be optimized for tablets as well as a variety of different smartphone displays.

You'll learn how to use Fragments to create layouts that scale and adapt to accommodate a variety of screen sizes and resolutions, as well as the best practices for developing and testing your UIs so that they look great on all screens.

After being introduced to some of the visual controls available from the Android SDK, you'll learn how to extend

and customize them. Using View Groups, you'll combine Views to create atomic, reusable UI elements made up of interacting subcontrols. You'll also create your own Views, to display data and interact with users in creative new ways.

Finally, you'll examine Adapters and learn how to use them to bind your presentation layer to the underlying data sources.

Fundamental Android UI Design

User interface (UI) design, user experience (UX), human computer interaction (HCI), and usability are huge topics that can't be covered in the depth they deserve within the confines of this book. Nonetheless, the importance of creating a UI that your users will understand and enjoy using can't be overstated.

Android introduces some new terminology for familiar programming metaphors that will be explored in detail in the following sections:

- **Views**—Views are the base class for all visual interface elements (commonly known as *controls* or *widgets*). All UI controls, including the layout classes, are derived from `View`.
- **View Groups**—View Groups are extensions of the View class that can contain multiple child Views. Extend the `ViewGroup` class to create compound controls made up of interconnected child Views. The `ViewGroup` class is also extended to provide the Layout Managers that help you lay out controls within your Activities.

- **Fragments**—Fragments, introduced in Android 3.0 (API level 11), are used to encapsulate portions of your UI. This encapsulation makes Fragments particularly useful when optimizing your UI layouts for different screen sizes and creating reusable UI elements. Each Fragment includes its own UI layout and receives the related input events but is tightly bound to the Activity into which each must be embedded. Fragments are similar to UI View Controllers in iPhone development.
- **Activities**—Activities, described in detail in the previous chapter, represent the window, or screen, being displayed. Activities are the Android equivalent of Forms in traditional Windows desktop development. To display a UI, you assign a View (usually a layout or Fragment) to an Activity.

Android provides several common UI controls, widgets, and Layout Managers.

For most graphical applications, it's likely that you'll need to extend and modify these standard Views—or create composite or entirely new Views—to provide your own user experience.

Android User Interface Fundamentals

All visual components in Android descend from the `View` class and are referred to generically as *Views*. You'll often see Views referred to as *controls* or *widgets* (not to be confused with home screen App Widgets described in Chapter 14, “Invading the Home Screen)—terms you're probably familiar with if you've previously done any GUI development.

The `ViewGroup` class is an extension of View designed to contain multiple Views. View Groups are used most commonly to manage the layout of child Views, but they can also be used to create atomic reusable components. View Groups that perform the former function are generally referred to as *layouts*.

In the following sections you'll learn how to put together increasingly complex UIs, before being introduced to Fragments, the Views available in the SDK, how to extend these Views, build your own compound controls, and create your own custom Views from scratch.

Assigning User Interfaces to Activities

A new Activity starts with a temptingly empty screen onto which you place your UI. To do so, call `setContentView`, passing in the View instance, or layout resource, to display. Because empty screens aren't particularly inspiring, you will almost always use `setContentView` to assign an Activity's UI when overriding its `onCreate` handler.

The `setContentView` method accepts either a layout's resource ID or a single View instance. This lets you define your UI either in code or using the preferred technique of external layout resources.

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

  setContentView(R.layout.main);
}
```

Using layout resources decouples your presentation layer from the application logic, providing the flexibility to change the presentation without changing code. This makes it possible to specify different layouts optimized for different hardware configurations, even changing them at run time based on hardware changes (such as screen orientation changes).

You can obtain a reference to each of the Views within a layout using the `findViewById` method:

```
TextView myTextView =
```

```
(TextView)findViewById(R.id.myTextView);
```

If you prefer the more *traditional* approach, you can construct the UI in code:

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

  TextView myTextView = new TextView(this);
  setContentView(myTextView);

  myTextView.setText("Hello, Android");
}
```

The `setContentView` method accepts a single View instance; as a result, you use layouts to add multiple controls to your Activity.

If you're using Fragments to encapsulate portions of your Activity's UI, the View inflated within your Activity's `onCreate` handler will be a layout that describes the relative position of each of your Fragments (or their containers). The UI used for each Fragment is defined in its own layout and inflated within the Fragment itself, as described later in this chapter.

Note that once a Fragment has been inflated into an Activity, the Views it contains become part of that Activity's View hierarchy. As a result you can find any of its child Views from within the parent Activity, using

`findViewById` as described previously.

Introducing Layouts

Layout Managers (or simply *layouts*) are extensions of the `ViewGroup` class and are used to position child Views within your UI. Layouts can be nested, letting you create arbitrarily complex UIs using a combination of layouts.

The Android SDK includes a number of layout classes. You can use these, modify them, or create your own to construct the UI for your Views, Fragments, and Activities. It's up to you to select and use the right combination of layouts to make your UI aesthetically pleasing, easy to use, and efficient to display.

The following list includes some of the most commonly used layout classes available in the Android SDK:

- `FrameLayout`—The simplest of the Layout Managers, the Frame Layout pins each child view within its frame. The default position is the top-left corner, though you can use the `gravity` attribute to alter its location. Adding multiple children stacks each new child on top of the one before, with each new View potentially obscuring the previous ones.
- `LinearLayout`—A Linear Layout aligns each child View in either a vertical or a horizontal line. A vertical

layout has a column of Views, whereas a horizontal layout has a row of Views. The Linear Layout supports a `weight` attribute for each child View that can control the relative size of each child View within the available space.

- `RelativeLayout`—One of the most flexible of the native layouts, the Relative Layout lets you define the positions of each child View relative to the others and to the screen boundaries.
- `GridLayout`—Introduced in Android 4.0 (API level 14), the Grid Layout uses a rectangular grid of infinitely thin lines to lay out Views in a series of rows and columns. The Grid Layout is incredibly flexible and can be used to greatly simplify layouts and reduce or eliminate the complex nesting often required to construct UIs using the layouts described above. It's good practice to use the Layout Editor to construct your Grid Layouts rather than relying on tweaking the XML manually.

Each of these layouts is designed to scale to suit the host device's screen size by avoiding the use of absolute positions or predetermined pixel values. This makes them particularly useful when designing applications that work well on a diverse set of Android hardware.

The Android documentation describes the features and properties of each layout class in detail; so, rather than repeat that information here, I'll refer you to

http://developer.android.com/guide/topics/ui/layout-objects.html.

You'll see practical example of how these layouts should be used as they're introduced in the examples throughout this book. Later in this chapter you'll also learn how to create compound controls by using and/or extending these layout classes.

Defining Layouts

The preferred way to define a layout is by using XML external resources.

Each layout XML must contain a single root element. This root node can contain as many nested layouts and Views as necessary to construct an arbitrarily complex UI.

The following snippet shows a simple layout that places a `TextView` above an `EditText` control using a vertical `LinearLayout`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Enter Text Below"
  />
  <EditText
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Text Goes Here!"
  />
</LinearLayout>
```

For each of the layout elements, the constants `wrap_content` and `match_parent` are used rather than an exact height or width in pixels. These constants, combined with layouts that scale (such as the Linear Layout, Relative Layout, and Grid Layout) offer the

simplest, and most powerful, technique for ensuring your layouts are screen-size and resolution independent.

The `wrap_content` constant sets the size of a View to the minimum required to contain the contents it displays (such as the height required to display a wrapped text string). The `match_parent` constant expands the View to match the available space within the parent View, Fragment, or Activity.

Later in this chapter you'll learn how to set the minimum height and width for your own controls, as well as further best practices for resolution independence.

Implementing layouts in XML decouples the presentation layer from the View, Fragment, and Activity controller code and business logic. It also lets you create hardware configuration-specific variations that are dynamically loaded without requiring code changes.

When preferred, or required, you can implement layouts in code. When assigning Views to layouts in code, it's important to apply `LayoutParameters` using the `setLayoutParams` method, or by passing them in to the `addView` call:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);

TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);
```

```
myTextView.setText("Enter Text Below");
myEditText.setText("Text Goes Here!");

int lHeight = LinearLayout.LayoutParams.MATCH_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;

ll.addView(myTextView, new
LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myEditText, new
LinearLayout.LayoutParams(lHeight, lWidth));
setContentView(ll);
```

Using Layouts to Create Device Independent User Interfaces

A defining feature of the layout classes described previously, and the techniques described for using them within your apps, is their ability to scale and adapt to a wide range of screen sizes, resolutions, and orientations.

The variety of Android devices is a critical part of its success. For developers, this diversity introduces a challenge for designing UIs to ensure that they provide the best possible experience for users, regardless of which Android device they own.

Using a Linear Layout

The Linear Layout is one of the simplest layout classes. It allows you to create simple UIs (or UI elements) that align a sequence of child Views in either a vertical or a horizontal line.

The simplicity of the Linear Layout makes it easy to use but limits its flexibility. In most cases you will use Linear Layouts to construct UI elements that will be nested within other layouts, such as the Relative Layout.

Listing 4.1 shows two nested Linear Layouts—a horizontal layout of two equally sized buttons within a vertical layout that places the buttons above a List View.

Listing 4.1: Linear Layout

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">
  <LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="5dp">
    <Button
      android:text="@string/cancel_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
    <Button
      android:text="@string/ok_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>
  <ListView
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
</LinearLayout>
```

code snippet PA4AD_Ch4_Layouts/res/layout/linear_layout.xml

If you find yourself creating increasingly complex nesting patterns of Linear Layouts, you will likely be better served using a more flexible Layout Manager.

Using a Relative Layout

The Relative Layout provides a great deal of flexibility

for your layouts, allowing you to define the position of each element within the layout in terms of its parent and the other Views.

Listing 4.2 modifies the layout described in Listing 4.1 to move the buttons below the List View.

Listing 4.2: Relative Layout

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <LinearLayout
    android:id="@+id/button_bar"
    android:layout_alignParentBottom="true"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal"
    android:padding="5dp">
    <Button
      android:text="@string/cancel_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
    <Button
      android:text="@string/ok_button_text"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>
  <ListView
    android:layout_above="@id/button_bar"
    android:layout_alignParentLeft="true"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
  </ListView>
</RelativeLayout>
```

code snippet
PA4AD_Ch4_Layouts/res/layout/relative_layout.xml

Using a Grid Layout

The Grid Layout was introduced in Android 3.0 (API level 11) and provides the most flexibility of any of the Layout Managers.

The Grid Layout uses an arbitrary grid to position Views. By using row and column spanning, the Space View, and Gravity attributes, you can create complex without resorting to the often complex nesting required to construct UIs using the Relative Layout described previously.

The Grid Layout is particularly useful for constructing layouts that require alignment in two directions—for example, a form whose rows and columns must be aligned but which also includes elements that don't fit neatly into a standard grid pattern.

It's also possible to replicate all the functionality provided by the Relative Layout by using the Grid Layout and Linear Layout in combination. For performance reasons it's good practice to use the Grid Layout in preference to creating the same UI using a combination of nested layouts.

Listing 4.3 shows the same layout as described in Listing 4.2 using a Grid Layout to replace the Relative Layout.

Listing 4.3: Grid Layout

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:orientation="vertical">
   <ListView
    android:background="#FF444444"
    android:layout_gravity="fill">
  </ListView>
  <LinearLayout
    android:layout_gravity="fill_horizontal"
    android:orientation="horizontal"
    android:padding="5dp">
    <Button
      android:text="Cancel"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
    <Button
      android:text="OK"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:layout_weight="1"/>
  </LinearLayout>
</GridLayout>
```

code snippet
PA4AD_Ch4_Layouts/res/layout/grid_layout.xml

Note that the Grid Layout elements do not require width and height parameters to be set. Instead, each element wraps its content by default, and the `layout_gravity` attribute is used to determine in which directions each element should expand.

Optimizing Layouts

Inflating layouts is an expensive process; each additional nested layout and included View directly impacts on the performance and responsiveness of your application.

To keep your applications smooth and responsive, it's important to keep your layouts as simple as possible and to avoid inflating entirely new layouts for relatively small UI changes.

Redundant Layout Containers Are Redundant

A Linear Layout within a Frame Layout, both of which are set to `MATCH_PARENT`, does nothing but add extra time to inflate. Look for redundant layouts, particularly if you've been making significant changes to an existing layout or are adding child layouts to an existing layout.

Layouts can be arbitrarily nested, so it's easy to create complex, deeply nested hierarchies. Although there is no hard limit, it's good practice to restrict nesting to fewer than 10 levels.

One common example of unnecessary nesting is a Frame Layout used to create the single root node required for a layout, as shown in the following snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <ImageView
    android:id="@+id/myImageView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/myimage"
  />
  <TextView
    android:id="@+id/myTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    android:gravity="center_horizontal"
    android:layout_gravity="bottom"
  />
</FrameLayout>
```

In this example, when the Frame Layout is added to a parent, it will become redundant. A better alternative is to use the `Merge` tag:

```
<?xml version="1.0" encoding="utf-8"?>
<merge
  xmlns:android="http://schemas.android.com/apk/res/android">
  <ImageView
    android:id="@+id/myImageView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:src="@drawable/myimage"
  />
  <TextView
    android:id="@+id/myTextView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/hello"
    android:gravity="center_horizontal"
    android:layout_gravity="bottom"
  />
</merge>
```

When a layout containing a `merge` tag is added

to another layout, the `merge` node is removed and its child Views are added directly to the new parent.

The `merge` tag is particularly useful in conjunction with the `include` tag, which is used to insert the contents of one layout into another:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <include android:id="@+id/my_action_bar"
             layout="@layout/actionbar"/>
  <include android:id="@+id/my_image_text_layout"
             layout="@layout/image_text_layout"/>
</LinearLayout>
```

Combining the `merge` and `include` tags enables you to create flexible, reusable layout definitions that don't create deeply nested layout hierarchies. You'll learn more about creating and using simple and reusable layouts later in this chapter.

Avoid Using Excessive Views

Each additional View takes time and resources to inflate. To maximize the speed and responsiveness of your application, none of its layouts should include more than 80 Views. When you exceed this limit, the time taken to inflate the layout becomes significant.

To minimize the number of Views inflated within a complex layout, you can use a `ViewStub`.

A View Stub works like a lazy include—a stub that represents the specified child Views within the parent layout—but the stub is only inflated explicitly via the `inflate` method or when it's made visible.

```
// Find the stub
View stub = findViewById(R.id.
download_progress_panel_stub);
// Make it visible, causing it to inflate the
child layout
stub.setVisibility(View.VISIBLE);

// Find the root node of the inflated stub
layout
View downloadProgressPanel =
findViewById(R.id.download_progress_panel);
```

As a result, the Views contained within the child layout aren't created until they are required—minimizing the time and resource cost of inflating complex UIs.

When adding a View Stub to your layout, you can override the `id` and `layout` parameters of the root View of the layout it represents:

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

```
  <ListView
    android:id="@+id/myListView"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
  <ViewStub

 android:id="@+id/download_progress_panel_stub"


 android:layout="@layout/progress_overlay_panel"

 android:inflatedId="@+id/download_progress_panel"

    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="bottom"
  />
</FrameLayout>
```

This snippet modifies the width, height, and gravity of the imported layout to suit the requirements of the parent layout. This flexibility makes it possible to create and reuse the same generic child layouts in a variety of parent layouts.

An ID has been specified for both the stub and the View Group it will become when inflated using the `id` and `inflatedId` attribute, respectively.

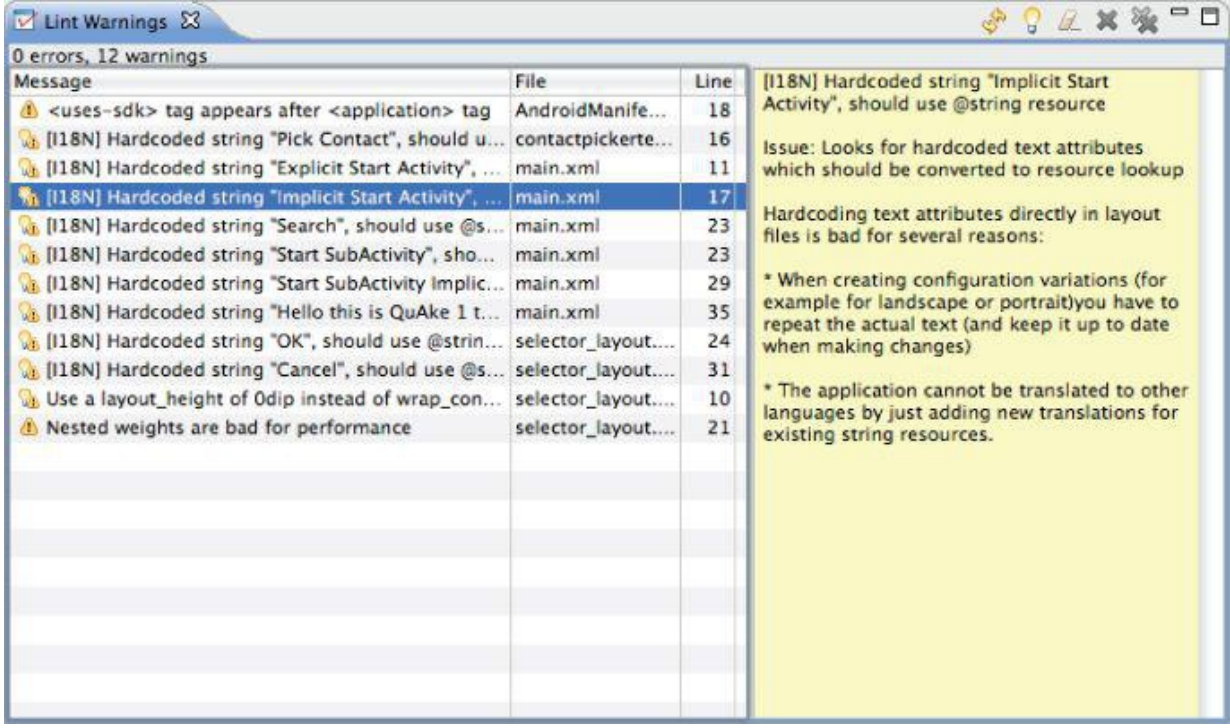
When the View Stub is inflated, it is removed from the hierarchy and replaced by the root node of the View it imported. If you need to modify the visibility of the imported Views, you must either use the reference to their root node (returned by the `inflate` call) or find the View by using `findViewById`, using the layout ID assigned to it within the corresponding View Stub node.

Using Lint to Analyze Your Layouts

To assist you in optimizing your layout hierarchies, the Android SDK includes `lint`—a powerful tool that can be used to detect problems within you application, including layout performance issues.

The lint tool is available as a command-line tool or as a window within Eclipse supplied as part of the ADT plug-in, as shown in Figure 4.1.

Figure 4.1



In addition to using Lint to detect each optimization issue described previously in this section, you can also use Lint to detect missing translations, unused resources, inconsistent array sizes, accessibility and internationalization problems, missing or duplicated image assets, usability problems, and manifest errors.

Lint is a constantly evolving tool, with new rules added regularly. A full list of the tests performed by the Lint tool can be found at http://tools.android.com/tips/lint-checks.

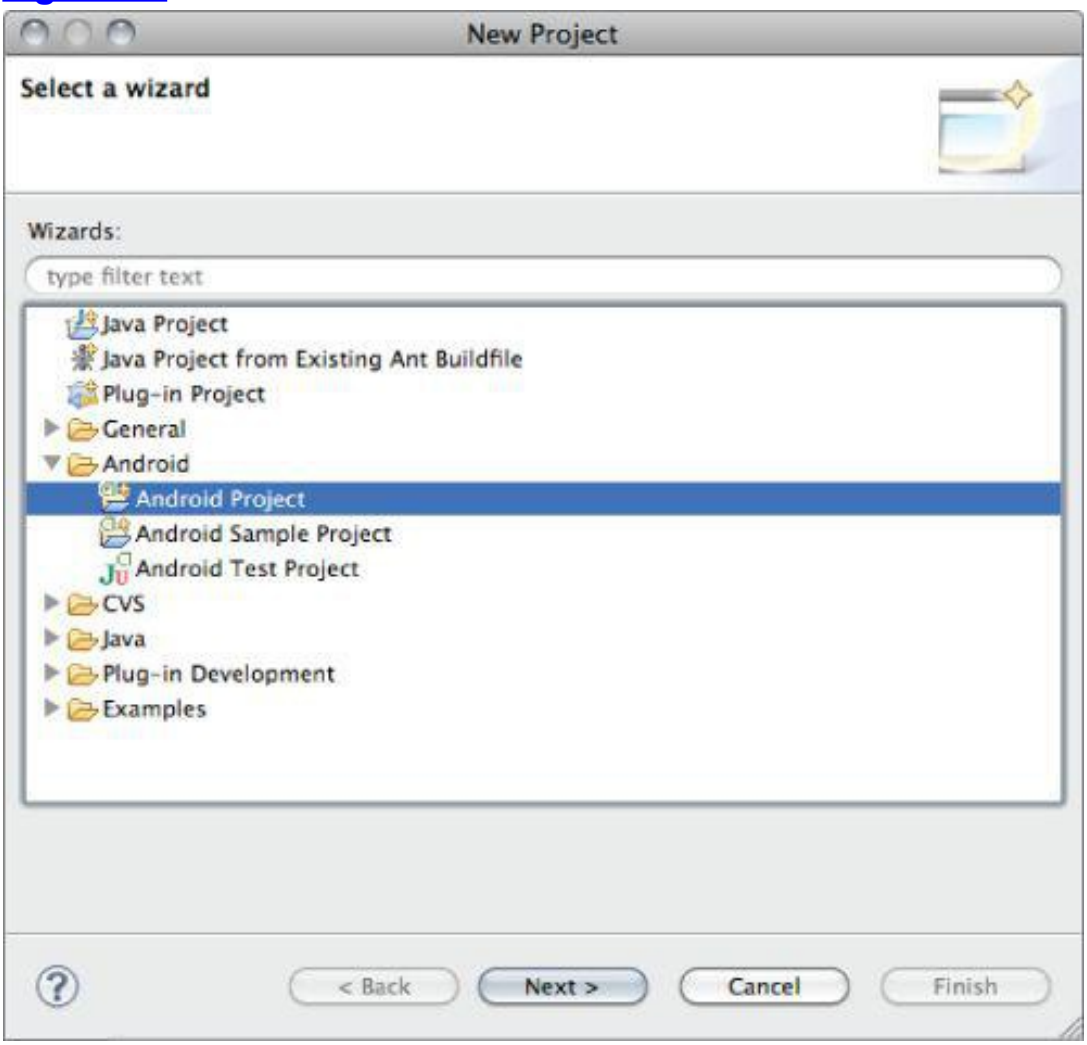
To-Do List Example

In this example you'll be creating a new Android application from scratch. This simple example creates a new to-do list application using native Android Views and layouts.

Don't worry if you don't understand everything that happens in this example. Some of the features used to create this application, including `ArrayAdapters`, `ListViews`, and `KeyListeners`, won't be introduced properly until later in this and subsequent chapters, where they'll be explained in detail. You'll also return to this example later to add new functionality as you learn more about Android.

1. Create a new Android project. Within Eclipse, select File → New → Project, and then choose Android Project within the Android node (as shown in Figure 4.2) before clicking Next.

Figure 4.2



2. Specify the project details for your new project.

1. Start by providing a project name, as shown in Figure 4.3, and then click Next.

2. Select the build target. Select the newest platform release, as shown in Figure 4.4, and then click Next.

3. Enter the details for your new project, as shown in Figure 4.5. The Application name is the friendly name of your application, and the Create Activity field lets you name your Activity (ToDoListActivity). When the remaining details are entered, click Finish to create your new project.

Figure 4.3

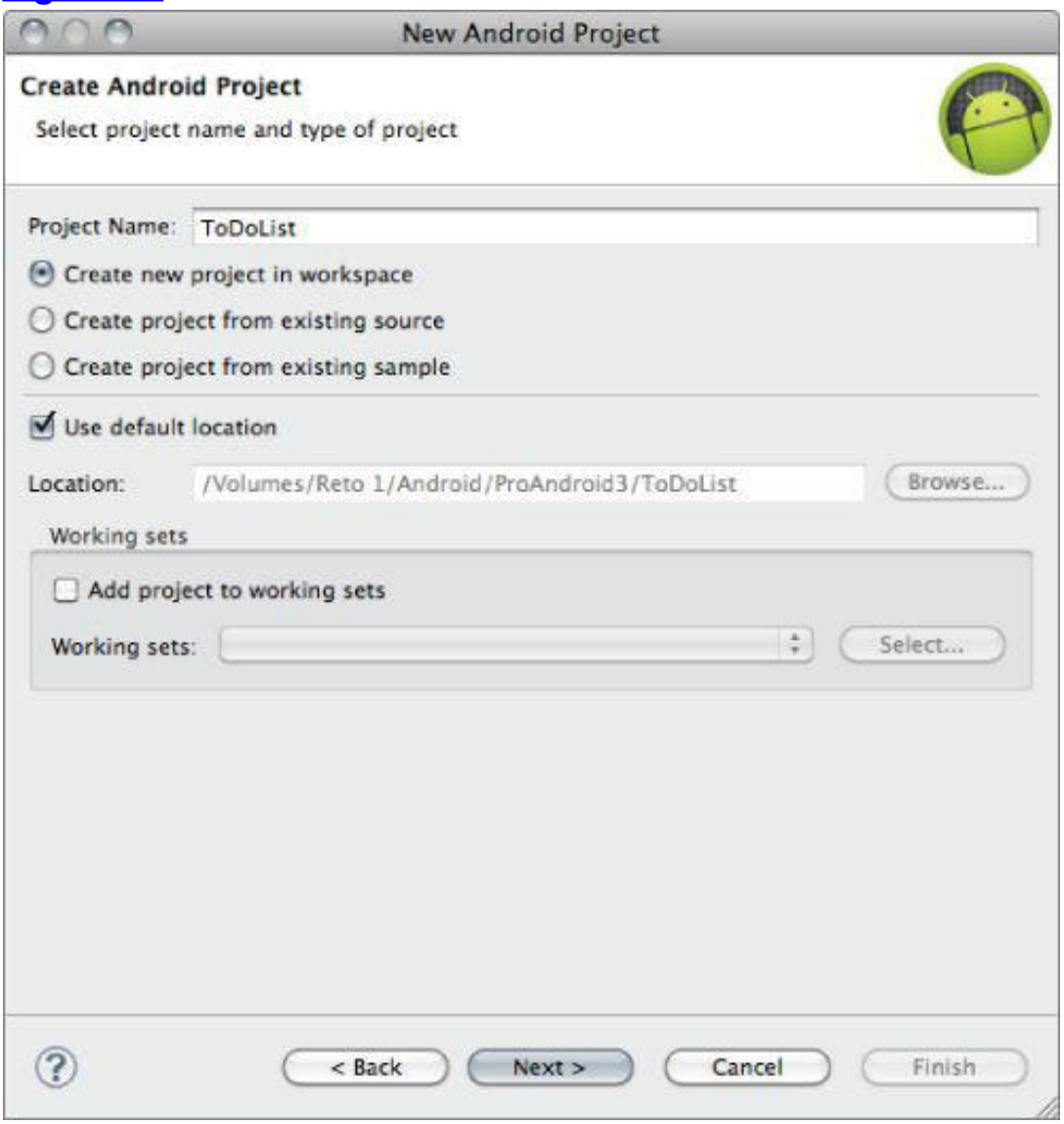
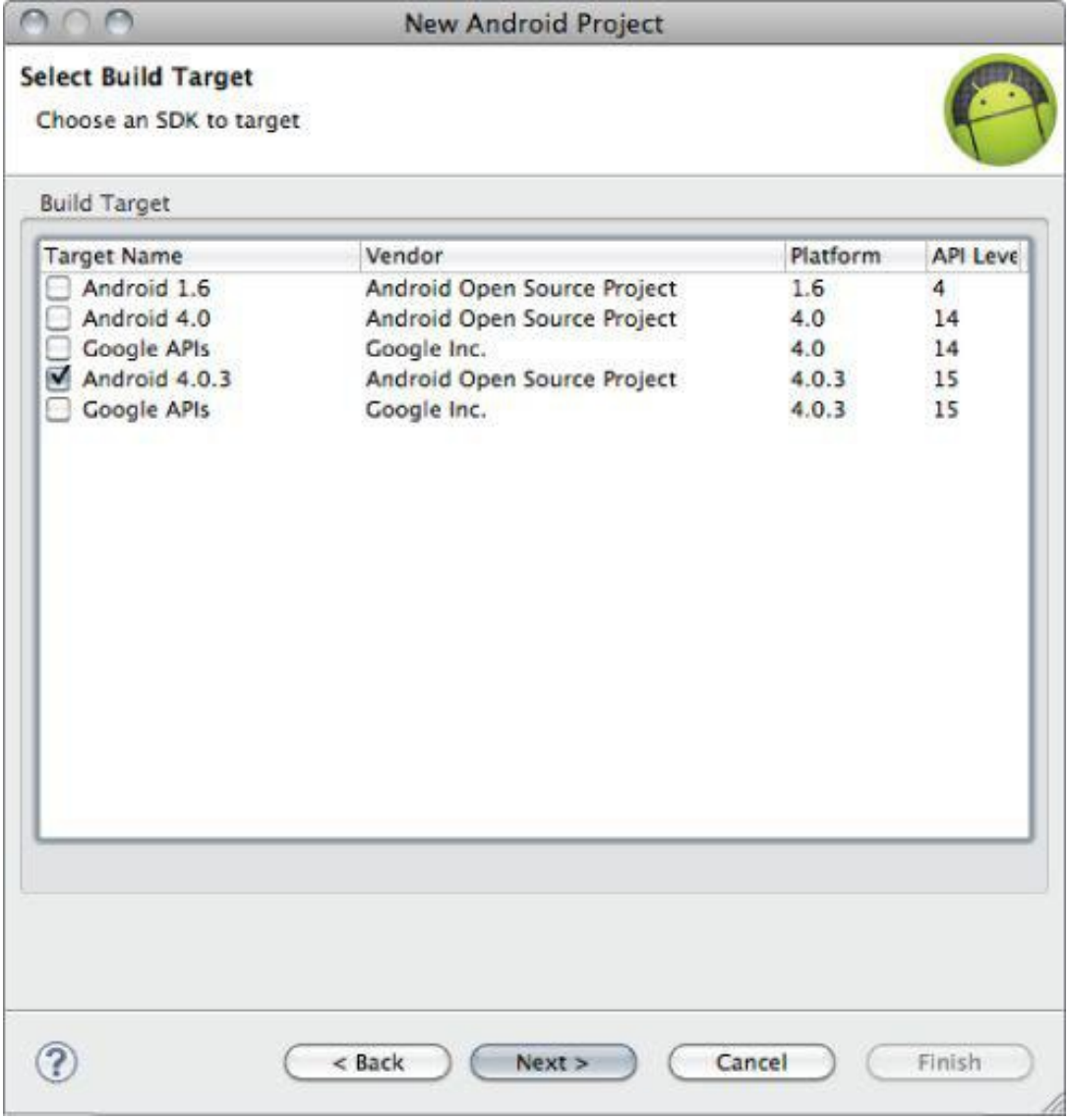
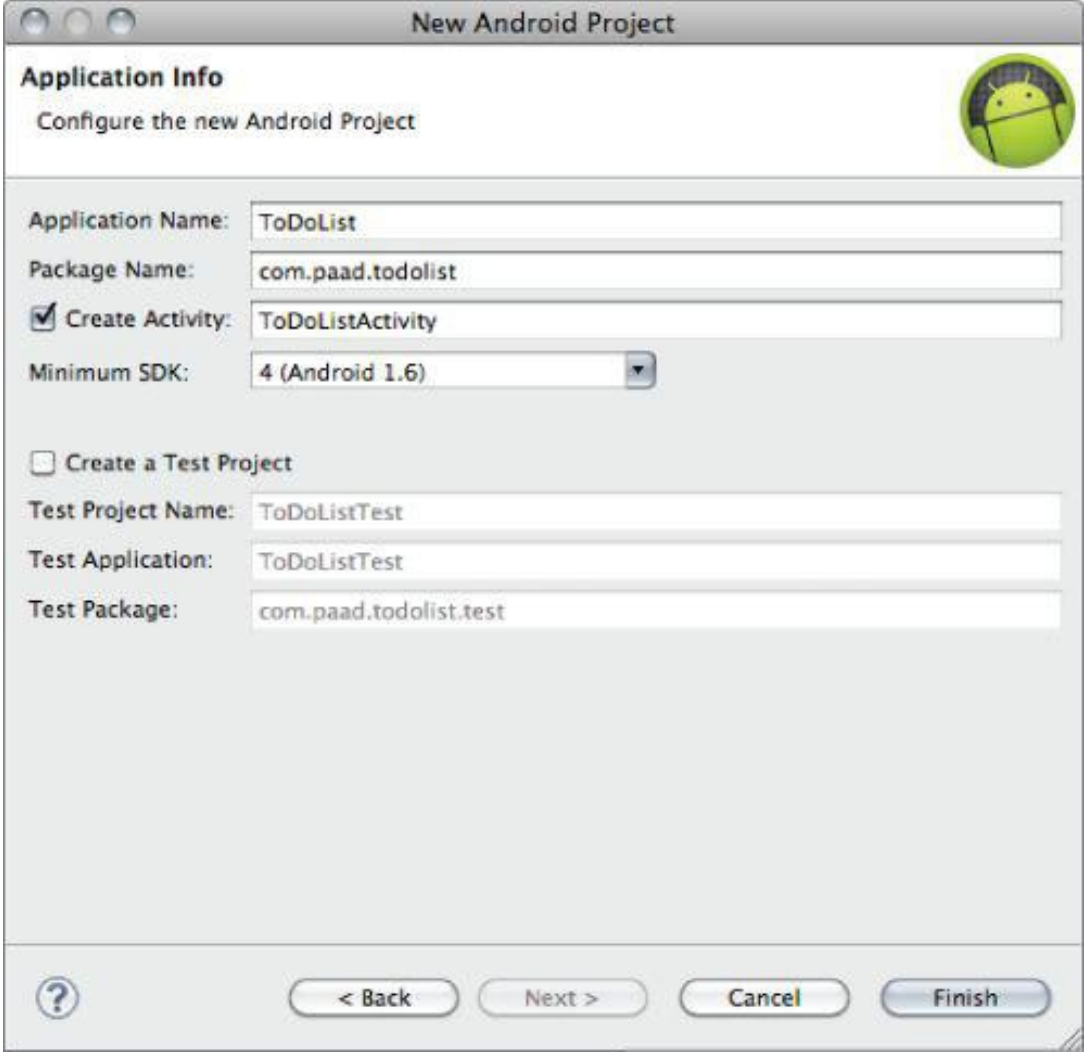


Figure 4.4


New Android Project
Select Build Target
Choose an SDK to target
Build Target
Target Name
Vendor
Platform
API Leve
Android 1.6
Android Open Source Project
1.6
4
Android 4.0
Android Open Source Project
4.0
14
Google APIs
Google Inc.
4.0
14
Android 4.0.3
Android Open Source Project
4.0.3
15
Google APIs
Google Inc.
4.0.3
15
< Back
Next >
Cancel
Finish

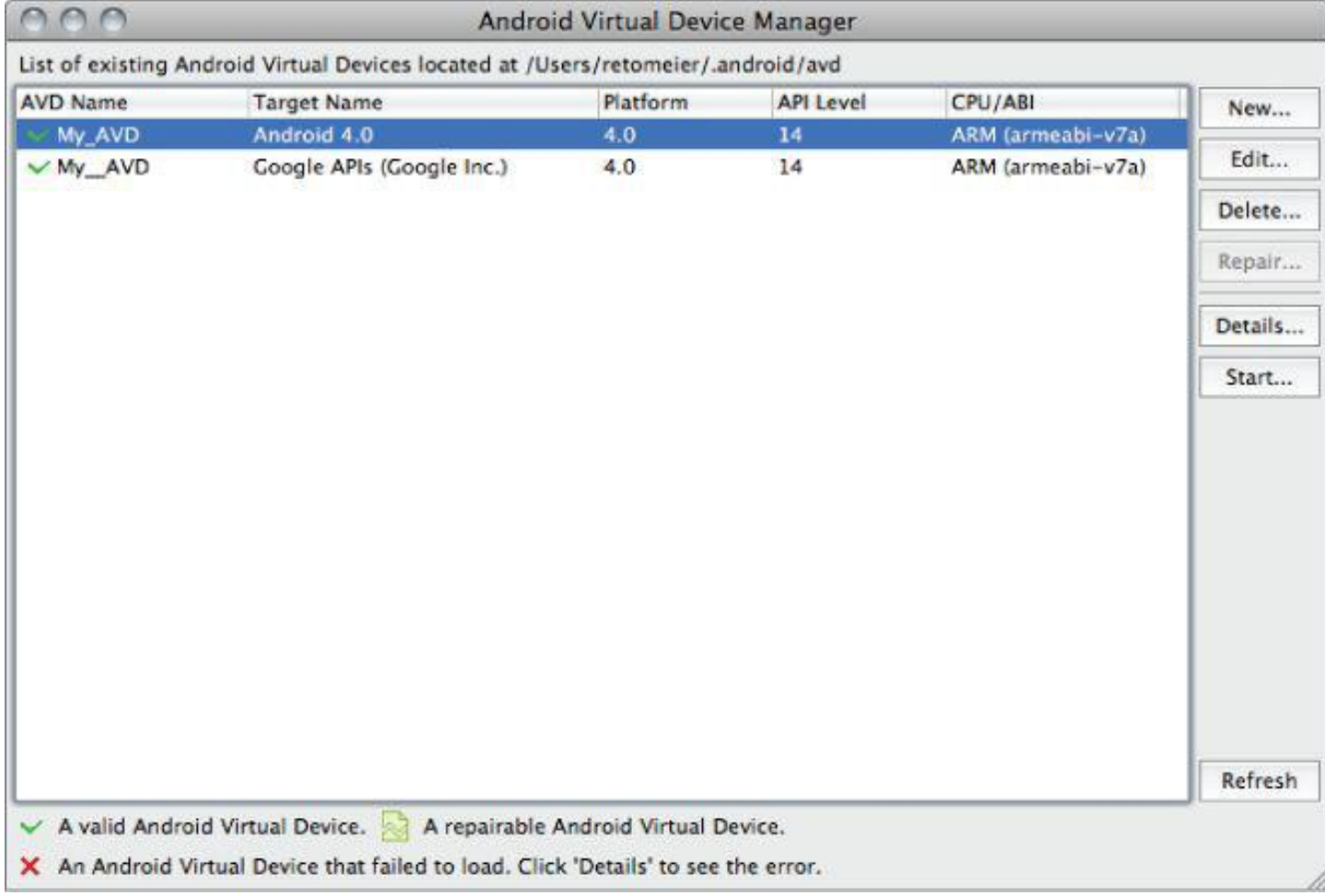
[Figure 4.5]



3. Before creating your debug and run configurations, take this opportunity to create a virtual device for testing your applications.

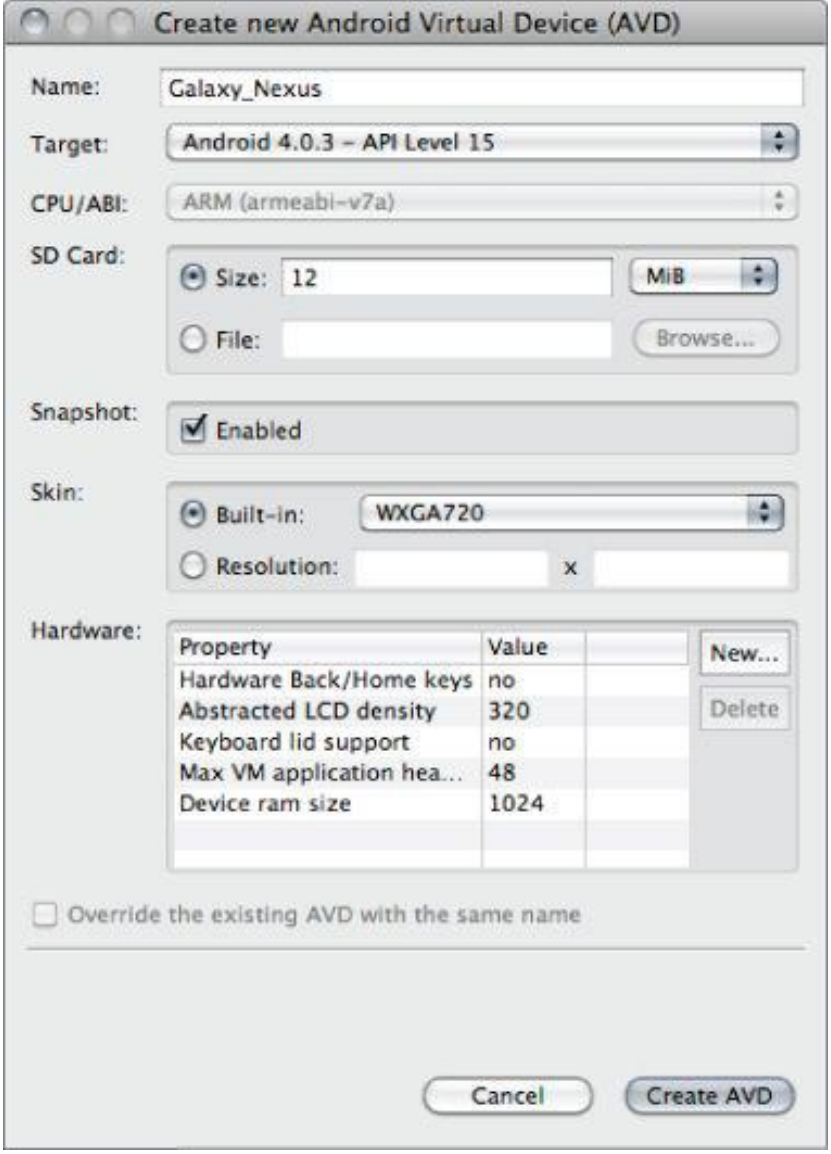
1. Select Window → AVD Manager. In the resulting dialog (see Figure 4.6), click the New button.

Figure 4.6



2. In the dialog displayed in Figure 4.7, enter a name for your device and choose an SDK target (use the same platform target as you selected for your project in step 2.2) and the screen resolution. Set the SD Card size to larger than 8MB, enable snapshots, and then press Create AVD.

Figure 4.7



4. Now create your debug and run configurations. Select Run → Debug Configurations and then Run → Run Configurations, creating a new configuration for each specifying the `TodoList` project. If you want to debug using a virtual device, you can select the one you created in step 3 here; alternatively, if you want to debug on a device, you can select it here if it's plugged in and has debugging enabled. You can either leave the launch action as Launch Default Activity or explicitly set it to launch the new `ToDoListActivity`.

5. In this example you want to present users with a list of to-do items and a text entry box to add new ones. There's both a list and a text-entry control available from the Android libraries. (You'll learn more about the Views available in Android, and how to create new ones, later in this Chapter.)

The preferred method for laying out your UI is to create a layout resource. Open the `main.xml` layout file in the `res/layout` project folder and modify it layout to

include a `ListView` and an `EditText` within a `LinearLayout`. You must give both the `EditText` and `ListView` an ID so that you can get references to them both in code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <EditText
    android:id="@+id/myEditText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="@string/addItemHint"

 android:contentDescription="@string/addItemContentDescription"
  />
  <ListView
    android:id="@+id/myListView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

6. You'll also need to add the string resources that provide the hint text and content description included in step 5 to the `strings.xml` resource stored in the project's `res/values` folder. You can take this opportunity to remove the default “hello” string value:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">ToDoList</string>
  <string name="addItemHint">New To Do Item</string>
  <string name="addItemContentDescription">New To Do
Item</string>
</resources>
```

7. With your UI defined, open the `ToDoListActivity` Activity from your project's `src` folder. Start by ensuring your UI is inflated using `setContentView`. Then get references to the `ListView` and `EditText` using `findViewById`:

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Inflate your View
  setContentView(R.layout.main);

  // Get references to UI widgets
  ListView myListView =
(ListView)findViewById(R.id.myListView);
  final EditText myEditText =
(EditText)findViewById(R.id.myEditText);
}
```

When you add the code from step 7 into the `ToDoListActivity`, or when you try to compile your project, your IDE or compiler will complain that the `ListView` and `EditText` classes cannot be resolved into a type.

You need to add import statements to your class to include the libraries that contain these Views (in this case, `android.widget.EditText` and `android.widget.ListView`). To ensure

the code snippets and example applications listed in this book remain concise and readable, not all the necessary import statements within the code listings are included within the text (however they are all included in the downloadable source code). If you are using Eclipse, classes with missing import statements are highlighted with a red underline. Clicking each highlighted class will display a list of "quick fixes," which include adding the necessary import statements on your behalf.

8. Still within `onCreate`, define an `ArrayList` of Strings to store each to-do list item. You can bind a `ListView` to an `ArrayList` using an `ArrayAdapter`. (This process is described in more detail later in this chapter.) Create a new `ArrayAdapter` instance to bind the to-do item array to the `ListView`.

```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Inflate your View
  setContentView(R.layout.main);

  // Get references to UI widgets
  ListView myListView =
(ListView)findViewById(R.id.myListView);
  final EditText myEditText =
(EditText)findViewById(R.id.myEditText);

  // Create the Array List of to do items
  final ArrayList<String> todoItems = new
ArrayList<String>();

  // Create the Array Adapter to bind the array to the
List View
  final ArrayAdapter<String> aa;

  aa = new ArrayAdapter<String>(this,

 android.R.layout.simple_list_item_1,
                                todoItems);

  // Bind the Array Adapter to the List View
  myListView.setAdapter(aa);
}
```

9. Let users add new to-do items. Add an `onKeyListener` to the `EditText` that listens for either a "D-pad center button" click or the Enter key being pressed. (You'll learn more about listening for key presses later in this chapter.) Either of these actions should add the contents of the `EditText` to the to-do list array created in step 8, and notify the `ArrayAdapter` of the change. Finally, clear the `EditText` to prepare for the next item.

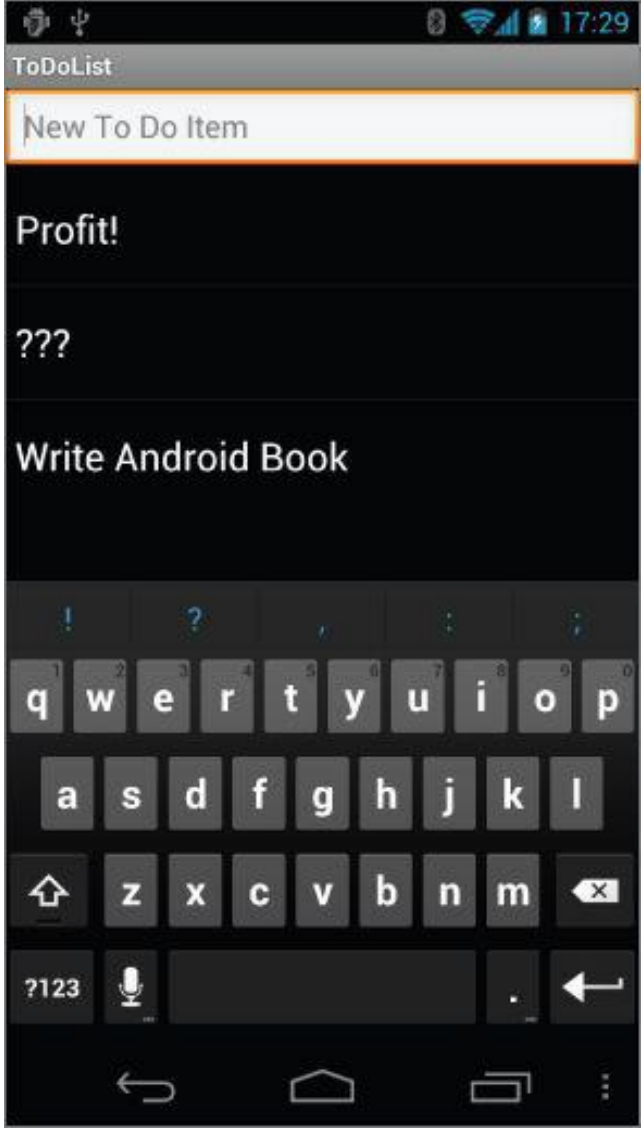
```
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Inflate your View
  setContentView(R.layout.main);

  // Get references to UI widgets
  ListView myListView =
(ListView)findViewById(R.id.myListView);
  final EditText myEditText =
(EditText)findViewById(R.id.myEditText);
```

```
  // Create the Array List of to do items
  final ArrayList<String> todoItems = new
ArrayList<String>();

  // Create the Array Adapter to bind the array to the
List View
  final ArrayAdapter<String> aa;

  aa = new ArrayAdapter<String>(this,

 android.R.layout.simple_list_item_1,
                                todoItems);

  // Bind the Array Adapter to the List View
  myListView.setAdapter(aa);

  myEditText.setOnKeyListener(new View.OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent
event) {
      if (event.getAction() == KeyEvent.ACTION_DOWN)
        if ((keyCode == KeyEvent.KEYCODE_DPAD_CENTER) ||
            (keyCode == KeyEvent.KEYCODE_ENTER)) {
          todoItems.add(0,
myEditText.getText().toString());
          aa.notifyDataSetChanged();
          myEditText.setText("");
          return true;
        }
      return false;
    }
  });
}
```

10. Run or debug the application and you'll see a text entry box above a list, as shown in Figure 4.8.

11. You've now finished your first Android application. Try adding breakpoints to the code to test the debugger and experiment with the DDMS perspective.

Figure 4.8



All code snippets in this example are part of the Chapter 4 To-Do List Part 1 project, available for download at www.wrox.com.

As it stands, this to-do list application isn't spectacularly useful. It doesn't save to-do list items between sessions; you can't edit or remove an item from the list; and typical task-list items, such as due dates and task priorities, aren't recorded or displayed. On balance, it fails most of the criteria laid out so far for a good mobile application design. You'll rectify some of these deficiencies when you return to this example.

Introducing Fragments

Fragments enable you to divide your Activities into fully encapsulated reusable components, each with its own lifecycle and UI.

The primary advantage of Fragments is the ease with which you can create dynamic and flexible UI designs that can be adapted to suite a range of screen sizes—from small-screen smartphones to tablets.

Each Fragment is an independent module that is tightly bound to the Activity into which it is placed. Fragments can be reused within multiple activities, as well as laid out in a variety of combinations to suit multipane tablet UIs and added to, removed from, and exchanged within a running Activity to help build dynamic UIs.

Fragments provide a way to present a consistent UI optimized for a wide variety of Android device types, screen sizes, and device densities.

Although it is not necessary to divide your Activities (and their corresponding layouts) into Fragments, doing so will drastically improve the flexibility of your UI and make it easier for you to adapt your user experience for new device configurations.

Fragments were introduced to Android as part of the Android 3.0 Honeycomb (API level 11) release. They are now also available as part of the Android support library, making it possible to take advantage of Fragments on platforms from Android 1.6 (API level 4) onward.

To use Fragments using the support library, you must make your Activity extend the `FragmentActivity` class:

```
public class MyActivity extends FragmentActivity
```

If you are using the compatibility library within a project that has a build target of API level 11 or above, it's critical that you ensure that all your Fragmentrelated imports and class references are using only the support library classes. The native and support library set of Fragment packages are closely related, but their classes are not interchangeable.

```
public class MyActivity extends FragmentActivity
```

If you are using the compatibility library within a project that has a build target of API level 11 or above, it's critical that you ensure that all your Fragment-related imports and class references are using only the support library classes. The native and support library set of Fragment packages are closely related, but their classes are not interchangeable.

Creating New Fragments

Extend the `Fragment` class to create a new Fragment, (optionally) defining the UI and implementing the functionality it encapsulates.

In most circumstances you'll want to assign a UI to your Fragment. It is possible to create a Fragment that *doesn't* include a UI but instead provides background behavior for an Activity. This is explored in more detail later in this chapter.

If your Fragment does require a UI, override the `onCreateView` handler to inflate and return the required View hierarchy, as shown in the Fragment skeleton code in Listing 4.4.

Listing 4.4: Fragment skeleton code

```
package com.paad.fragments;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class MySkeletonFragment extends Fragment {
  @Override
```

```
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    // Create, or inflate the Fragment's UI, and return it.
    // If this Fragment has no UI then return null.
    return inflater.inflate(R.layout.my_fragment, container,
false);
  }
}
```

code snippet PA4AD_Ch04_Fragments/src/MySkeletonFragment.java

You can create a layout in code using layout View Groups; however, as with Activities, the preferred way to design Fragment UI layouts is by inflating an XML resource.

Unlike Activities, Fragments don't need to be registered in your manifest. This is because Fragments can exist only when embedded into an Activity, with their lifecycles dependent on that of the Activity to which they've been added.

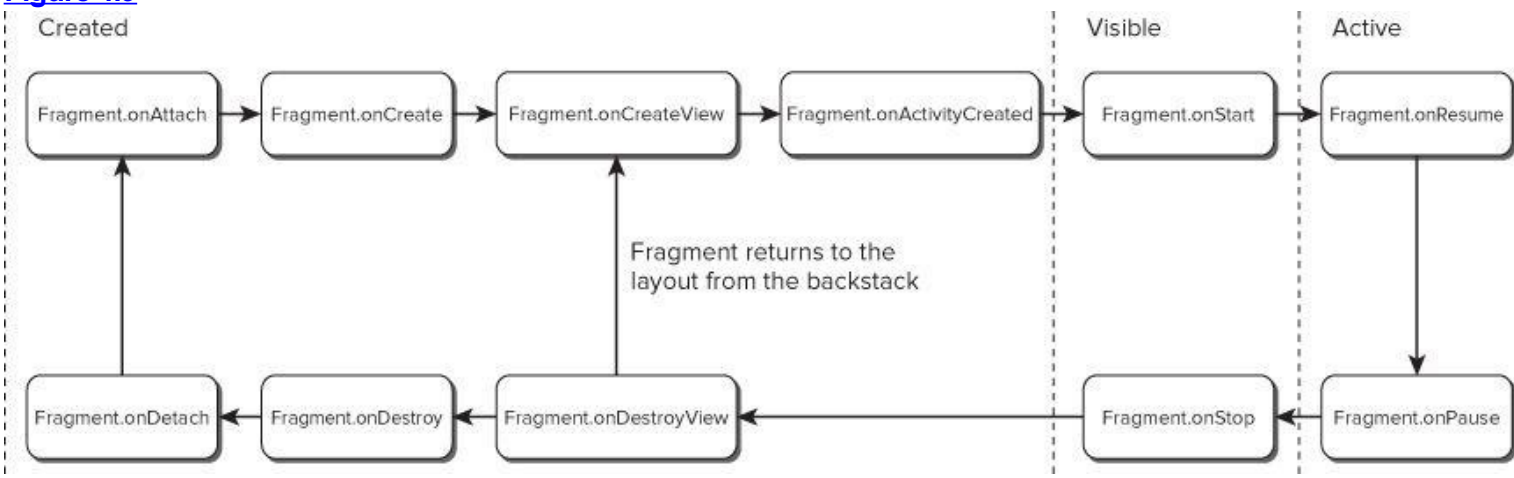
The Fragment Lifecycle

The lifecycle events of a Fragment mirror those of its parent Activity; however, after the containing Activity is in its active —resumed—state adding or removing a Fragment will affect its lifecycle independently.

Fragments include a series of event handlers that mirror those in the Activity class. They are triggered as the Fragment is created, started, resumed, paused, stopped, and destroyed. Fragments also include a number of additional callbacks that signal binding and unbinding the Fragment from its parent Activity, creation (and destruction) of the Fragment's View hierarchy, and the completion of the creation of the parent Activity.

Figure 4.9 summarizes the Fragment lifecycle.

Figure 4.9



The skeleton code in Listing 4.5 shows the stubs for the lifecycle handlers available in a Fragment. Comments within each stub describe the actions you should consider taking on each state change event.

You must call back to the superclass when overriding most of these event handlers.

Listing 4.5: Fragment lifecycle event handlers

```
package com.paad.fragments;

import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class MySkeletonFragment extends Fragment {

  // Called when the Fragment is attached to its parent
Activity.
  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);
    // Get a reference to the parent Activity.
  }
```

```
  // Called to do the initial creation of the Fragment.
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // Initialize the Fragment.
  }

  // Called once the Fragment has been created in order for it
to
  // create its user interface.
  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
    // Create, or inflate the Fragment's UI, and return it.
    // If this Fragment has no UI then return null.
    return inflater.inflate(R.layout.my_fragment, container,
false);
  }


  // Called once the parent Activity and the Fragment's UI
have
  // been created.
  @Override
  public void onActivityCreated(Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    // Complete the Fragment initialization - particularly
anything
    // that requires the parent Activity to be initialized or
the
    // Fragment's view to be fully inflated.
  }

  // Called at the start of the visible lifetime.
  @Override
  public void onStart(){
    super.onStart();
    // Apply any required UI change now that the Fragment is
visible.
  }

  // Called at the start of the active lifetime.
  @Override
  public void onResume(){
    super.onResume();
    // Resume any paused UI updates, threads, or processes
required
    // by the Fragment but suspended when it became inactive.
  }

  // Called at the end of the active lifetime.
  @Override
  public void onPause(){
    // Suspend UI updates, threads, or CPU intensive processes
    // that don't need to be updated when the Activity isn't
    // the active foreground activity.
    // Persist all edits or state changes
    // as after this call the process is likely to be killed.
    super.onPause();
  }

  // Called to save UI state changes at the
  // end of the active lifecycle.
  @Override
  public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save UI state changes to the savedInstanceState.
    // This bundle will be passed to onCreate, onCreateView,
and
    // onCreateView if the parent Activity is killed and
restarted.
    super.onSaveInstanceState(savedInstanceState);
  }

  // Called at the end of the visible lifetime.
  @Override
  public void onStop(){
    // Suspend remaining UI updates, threads, or processing
    // that aren't required when the Fragment isn't visible.
    super.onStop();
  }

  // Called when the Fragment's View has been detached.
  @Override
  public void onDestroyView() {
    // Clean up resources related to the View.
    super.onDestroyView();
  }

  // Called at the end of the full lifetime.
  @Override
  public void onDestroy(){
    // Clean up any resources including ending threads,
    // closing database connections etc.
```

```
    super.onDestroy();
  }

  // Called when the Fragment has been detached from its
parent Activity.
  @Override
  public void onDetach() {
    super.onDetach();
  }
}
```

code snippet
PA4AD_Ch04_Fragments/src/MySkeletonFragment.java

Fragment-Specific Lifecycle Events

Most of the Fragment lifecycle events correspond to their equivalents in the Activity class, which were covered in detail in Chapter 3. Those that remain are specific to Fragments and the way in which they're inserted into their parent Activity.

Attaching and Detaching Fragments from the Parent Activity

The full lifetime of your Fragment begins when it's bound to its parent Activity and ends when it's been detached. These events are represented by the calls to `onAttach` and `onDetach`, respectively.

As with any handler called after a Fragment/Activity has become paused, it's possible that `onDetach` will not be called if the parent Activity's process is terminated *without* completing its full lifecycle.

The `onAttach` event is triggered before the Fragment's UI has been created, before the Fragment itself or its parent Activity have finished their initialization. Typically, the `onAttach` event is used to gain a reference to the parent Activity in preparation for further initialization tasks.

Creating and Destroying Fragments

The created lifetime of your Fragment occurs between the first call to `onCreate` and the final call to `onDestroy`. As it's not uncommon for an Activity's process to be terminated *without* the corresponding `onDestroy` method being called, so a Fragment can't rely on its `onDestroy` handler being triggered.

As with Activities, you should use the `onCreate` method to initialize your Fragment. It's good practice to create any class scoped objects here to ensure they're created only once in the Fragment's lifetime.

Unlike Activities, the UI is *not* initialized within `onCreate`.

Creating and Destroying User Interfaces

A Fragment's UI is initialized (and destroyed) within a

new set of event handlers: `onCreateView` and `onDestroyView`, respectively.

Use the `onCreateView` method to initialize your Fragment: Inflate the UI, get references (and bind data to) the Views it contains, and then create any required Services and Timers.

Once you have inflated your View hierarchy, it should be returned from this handler:

```
return inflater.inflate(R.layout.my_fragment, container,
false);
```

If your Fragment needs to interact with the UI of its parent Activity, wait until the `onActivityCreated` event has been triggered. This signifies that the containing Activity has completed its initialization and its UI has been fully constructed.

Fragment States

The fate of a Fragment is inextricably bound to that of the Activity to which it belongs. As a result, Fragment state transitions are closely related to the corresponding Activity state transitions.

Like Activities, Fragments are active when they belong to an Activity that is focused and in the foreground. When an Activity is paused or stopped, the Fragments it contains are also paused and stopped, and the Fragments contained by an inactive Activity are also inactive. When an Activity is finally destroyed, each Fragment it contains is likewise destroyed.

As the Android memory manager nondeterministically closes applications to free resources, the Fragments within those Activities are also destroyed.

While Activities and their Fragments are tightly bound, one of the advantages of using Fragments to compose your Activity's UI is the flexibility to dynamically add or remove Fragments from an active Activity. As a result, each Fragment can progress through its full, visible, and active lifecycle several times within the active lifetime of its parent Activity.

Whatever the trigger for a Fragment's transition through its lifecycle, managing its state transitions is critical in ensuring a seamless user experience. There should be no difference in a Fragment moving from a paused, stopped, or inactive state back to active, so it's important to save all UI state and persist all data when a Fragment is paused or stopped. Like an Activity, when a Fragment becomes active again, it should restore that saved state.

Introducing the Fragment Manager

Each Activity includes a Fragment Manager to manage the Fragments it contains. You can access the Fragment Manager using the `getFragmentManager` method:

```
FragmentManager fragmentManager = getFragmentManager();
```

The Fragment Manager provides the methods used to access the Fragments currently added to the Activity, and to perform Fragment Transaction to add, remove, and replace Fragments.

Adding Fragments to Activities

The simplest way to add a Fragment to an Activity is by including it within the Activity's layout using the `fragment` tag, as shown in Listing 4.6.

Listing 4.6: Adding Fragments to Activities using XML layouts

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <fragment
android:name="com.paad.weatherstation.MyListFragment"
    android:id="@+id/my_list_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
  />
  <fragment
android:name="com.paad.weatherstation.DetailsFragment"
    android:id="@+id/details_fragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="3"
  />
</LinearLayout>
```

code snippet
PA4AD_Ch04_Fragments/res/layout/fragment_layout.xml

Once the Fragment has been inflated, it becomes a View Group, laying out and managing its UI within the Activity.

This technique works well when you use Fragments to define a set of static layouts based on various screen sizes. If you plan to dynamically modify your layouts by adding, removing, and replacing Fragments at run time, a better approach is to create layouts that use container Views into which Fragments can be placed at runtime, based on the current application state.

Listing 4.7 shows an XML snippet that you could use to support this latter approach.

Listing 4.7: Specifying Fragment layouts using container views

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout
    android:id="@+id/ui_container"
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
    android:layout_weight="1"
  />
  <FrameLayout
    android:id="@+id/details_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="3"
  />
</LinearLayout>
```

code snippet
PA4AD_Ch04_Fragments/res/layout/fragment_container_layout.xml

You then need to create and add the corresponding Fragments to their appropriate parent containers within the `onCreate` handler of your Activity using Fragment Transactions, as described in the next section.

Using Fragment Transactions

Fragment Transactions can be used to add, remove, and replace Fragments within an Activity at run time. Using Fragment Transactions, you can make your layouts dynamic—that is, they will adapt and change based on user interactions and application state.

Each Fragment Transaction can include any combination of supported actions, including adding, removing, or replacing Fragments. They also support the specification of the transition animations to display and whether to include the Transaction on the back stack.

A new Fragment Transaction is created using the `beginTransaction` method from the Activity's Fragment Manager. Modify the layout using the `add`, `remove`, and `replace` methods, as required, before setting the animations to display, and setting the appropriate back-stack behavior. When you are ready to execute the change, call `commit` to add the transaction to the UI queue.

```
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();

// Add, remove, and/or replace Fragments.
// Specify animations.
// Add to back stack if required.

fragmentTransaction.commit();
```

Each of these transaction types and options will be explored in the following sections.

Adding, Removing, and Replacing Fragments

When adding a new UI Fragment, specify the Fragment instance to add, along with the container View into which the Fragment will

be placed. Optionally, you can specify a tag that can later be used to find the Fragment by using the `findFragmentByTag` method:

```
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
fragmentTransaction.add(R.id.ui_container, new
MyListFragment());
fragmentTransaction.commit();
```

To remove a Fragment, you first need to find a reference to it, usually using either the Fragment Manager's `findFragmentById` or `findFragmentByTag` methods. Then pass the found Fragment instance as a parameter to the `remove` method of a Fragment Transaction:

```
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();
Fragment fragment =
fragmentManager.findFragmentById(R.id.details_fragment);
fragmentTransaction.remove(fragment);
fragmentTransaction.commit();
```

You can also replace one Fragment with another. Using the `replace` method, specify the container ID containing the Fragment to be replaced, the Fragment with which to replace it, and (optionally) a tag to identify the newly inserted Fragment.

```
FragmentTransaction
fragmentTransaction =
fragmentManager.beginTransaction();

fragmentTransaction.replace(R.id.details_fragment,
                            new
DetailFragment(selected_index));
fragmentTransaction.commit();
```

Using the Fragment Manager to Find Fragments

To find Fragments within your Activity, use the Fragment Manager's `findFragmentById` method. If you have added your Fragment to the Activity layout in XML, you can use the Fragment's resource identifier:

```
MyFragment myFragment =
  (MyFragment)fragmentManager.findFragmentById(R.id.MyFragment);
```

If you've added a Fragment using a Fragment Transaction, you should specify the resource identifier of the container View to which you added the Fragment

you want to find. Alternatively, you can use the `findFragmentByTag` method to search for the Fragment using the tag you specified in the Fragment Transaction:

```
MyFragment myFragment =
  (MyFragment)fragmentManager.findFragmentByTag(MY_FRAGMENT_TAG);
```

Later in this chapter you'll be introduced to Fragments that don't include a UI. The `findFragmentByTag` method is essential for interacting with these Fragments. Because they're not part of the Activity's View hierarchy, they don't have a resource identifier or a container resource identifier to pass in to the `findFragmentById` method.

Populating Dynamic Activity Layouts with Fragments

If you're dynamically changing the composition and layout of your Fragments at run time, it's good practice to define only the parent containers within your XML layout and populate it exclusively using Fragment Transactions at run time to ensure consistency when configuration changes (such as screen rotations) cause the UI to be re-created.

Listing 4.8 shows

the skeleton code used to populate an Activity's layout with Fragments at run time.

Listing 4.8: Populating Fragment layouts using container views

```
public void
onCreate(Bundle
savedInstanceState)
{
  super.onCreate(savedInstanceState);

  // Inflate the
layout containing
the Fragment
containers
  setContentView(R.layout.fragment_container_layout);

  FragmentManager fm
=
getFragmentManager();

  // Check to see if
the Fragment back
stack has been
populated
  // If not, create
and populate the
layout.
  DetailsFragment
detailsFragment =

 (DetailsFragment)fm.findFragmentById(R.id.details_container);

  if
(detailsFragment ==
null) {

FragmentTransaction
ft =
fm.beginTransaction();

ft.add(R.id.details_container,
new
DetailsFragment());

ft.add(R.id.ui_container,
new
MyListFragment());
     ft.commit();
   }
}
```

code snippet PA4AD_Ch04_Fragments/src/MyFragmentActivity.java

You should first check if the UI has already been populated based on the previous state. To ensure a consistent user experience, Android persists

the Fragment layout and associated back stack when an Activity is restarted due to a configuration change.

For the same reason, when creating alternative layouts for run time configuration changes, it's considered good practice to include any view containers involved in any transactions in all the layout variations. Failing to do so may result in the Fragment Manager attempting to restore Fragments to containers that don't exist in the new layout.

To remove a Fragment container in a given orientation layout, simply mark its `visibility` attribute as `gone` in your layout definition, as shown in Listing 4.9.

Listing 4.9: Hiding Fragments in layout

variations

```
<?xml
version="1.0"
encoding="utf-
8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <FrameLayout

 android:id="@+id/ui_container"

 android:layout_width="match_parent"

 android:layout_height="match_parent"

 android:layout_weight="1"
  />
  <FrameLayout

 android:id="@+id/details_container"

 android:layout_width="match_parent"

 android:layout_height="match_parent"

 android:layout_weight="3"

 android:visibility="gone"
  />
</LinearLayout>
```

code snippet PA4AD_Ch04_Fragments/res/layout-port/fragment_container_layout.xml

Fragments and the Back Stack

Chapter 3 described the concept of Activity stacks—the logical stacking of Activities that are no longer visible—which allow users to navigate back to previous screens using the back button.

Fragments enable you to create dynamic

Activity layouts that can be modified to present significant changes in the UIs. In some cases these changes could be considered a new screen—in which case a user may reasonably expect the back button to return to the previous layout. This involves reversing previously executed Fragment Transactions.

Android provides a convenient technique for providing this functionality. To add the Fragment Transaction to the back stack, call `addToBackStack` on a Fragment Transaction before calling `commit`.

```
FragmentTransaction
fragmentTransaction
=
fragmentManager.beginTransaction();

fragmentTransaction.add(R.id.ui_container,
new
MyListFragment());

Fragment
fragment
=
```

```
fragmentManager.findFragmentById(R.id.details_fragment);
fragmentTransaction.remove(fragment);

String
tag =
null;
fragmentTransaction.addToBackStack(tag);

fragmentTransaction.commit();
```

Pressing the Back button will then reverse the previous Fragment Transaction and return the UI to the earlier layout.

When the Fragment Transaction shown above is committed, the Details Fragment is stopped and moved to the back stack, rather than simply destroyed. If the Transaction is reversed, the List Fragment is destroyed, and the Details Fragment

is
restarted.

Animating Fragment Transactions

To apply one of the default transition animations, use the `setTransition` method on any Fragment Transaction, passing in one of the `FragmentTransaction.TRANSIT_FRAGMENT_*` constants.

```
transaction.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_OPEN);
```

You can also apply custom animations to Fragment Transactions by using the `setCustomAnimations` method. This method accepts two animation XML resources: one for Fragments that are being added to

the layout by this transaction, and another for Fragments being removed:

```
fragmentTransaction.setCustomAnimations(R.animator.slide_in_left,

 R.animator.slide_out_right);
```

This is a particularly useful way to add seamless dynamic transitions when you are replacing Fragments within your layout.

The Android animation libraries were significantly improved in Android 3.0 (API level 11) with the inclusion of the `Animator` class. As a result, the animation resource passed in to the `setCustomAnimations` method is different for applications built using the support library. Applications built for devices running on API level 11 and above should use Animator resources, whereas those using the support library to support earlier platform releases should use the older View Animation resources.

You can find more details on creating custom Animator and Animation resources in Chapter 11, “Advanced User Experience.”

Interfacing Between Fragments and Activities

Use the `getActivity` method within any Fragment to return a reference to the Activity within which it's embedded. This is particularly useful for finding the current Context, accessing other Fragments using the Fragment Manager, and finding Views within the Activity's View hierarchy.

```
TextView textView =
(TextView)getActivity().findViewById(R.id.textview);
```

Although it's possible for Fragments to communicate directly using the host Activity's Fragment Manager, it's generally considered better practice to use the Activity as an intermediary. This allows the Fragments to be as independent and loosely coupled as possible, with the responsibility for deciding how an event in one Fragment should affect the overall UI falling to the host Activity.

Where your Fragment needs to share events with its host Activity (such as signaling UI selections), it's good practice to create a callback interface within the Fragment that a host Activity must implement.

Listing 4.10 shows a code snippet from within a Fragment class that defines a public event listener interface. The `onAttach` handler is overridden to obtain a reference to the host Activity, confirming that it

implements the required interface.

Listing 4.10: Defining Fragment event callback interfaces

```
public interface OnSeasonSelectedListener {
  public void onSeasonSelected(Season season);
}

private OnSeasonSelectedListener
onSeasonSelectedListener;
private Season currentSeason;

@Override
public void onAttach(Activity activity) {
  super.onAttach(activity);

  try {
    onSeasonSelectedListener =
(OnSeasonSelectedListener)activity;
  } catch (ClassCastException e) {
    throw new ClassCastException(activity.toString() +
              " must implement
OnSeasonSelectedListener");
  }
}

private void setSeason(Season season) {
  currentSeason = season;
  onSeasonSelectedListener.onSeasonSelected(season);
}
```

code snippet
PA4AD_Ch04_Fragments/src/SeasonFragment.java

Fragments Without User Interfaces

In most circumstances, Fragments are used to encapsulate modular components of the UI; however, you can also create a Fragment without a UI to provide background behavior that persists across Activity restarts. This is particularly well suited to background tasks that regularly touch the UI or where it's important to maintain state across Activity restarts caused by configuration changes.

You can choose to have an active Fragment retain its current instance when its parent Activity is re-created using the `setRetainInstance` method. After you call this method, the Fragment's lifecycle will change.

Rather than being destroyed and re-created with its parent Activity, the same Fragment instance is retained when the Activity restarts. It will receive the `onDetach` event when the parent Activity is destroyed, followed by the `onAttach`, `onCreateView`, and `onActivityCreated` events as the new parent Activity is instantiated.

The following snippet shows the skeleton code for a Fragment without a UI:

Although you use this technique on Fragments with a UI, this is generally not recommended. A better alternative is to move the associated background task or required state into a new Fragment, without a UI, and have the two Fragments interact as required.

```
public class NewItemFragment extends Fragment {
  @Override
  public void onAttach(Activity activity) {
    super.onAttach(activity);

    // Get a type-safe reference to the parent Activity.
```

```
  }

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

    // Create background worker threads and tasks.
  }

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

    // Initiate worker threads and tasks.
  }
}
```

To add this Fragment to your Activity, create a new Fragment Transaction, specifying a tag to use to identify it. Because the Fragment has no UI, it should not be associated with a container View and generally shouldn't be added to the back stack.

```
FragmentTransaction fragmentTransaction =
fragmentManager.beginTransaction();

fragmentTransaction.add(workerFragment, MY_FRAGMENT_TAG);

fragmentTransaction.commit();
```

Use the `findFragmentByTag` from the Fragment Manager to find a reference to it later.

```
MyFragment myFragment =
  (MyFragment)fragmentManager.findFragmentByTag(MY_FRAGMENT_TAG);
```

Android Fragment Classes

The Android SDK includes a number of Fragment subclasses that encapsulate some of the most common Fragment implementations. Some of the more useful ones are listed here:

- `DialogFragment`—A Fragment that you can use to display a floating Dialog over the parent Activity. You can customize the Dialog's UI and control its visibility directly via the Fragment API. Dialog Fragments are covered in more detail in Chapter 10, “Expanding the User Experience.”
- `ListFragment`—A wrapper class for Fragments that feature a `ListView` bound to a data source as the primary UI metaphor. It provides methods to set the Adapter to use and exposes the event handlers for list item selection. The List Fragment is used as part of the To-Do List example in the next section.
- `WebViewFragment`—A wrapper class that encapsulates a WebView within a Fragment. The child WebView will be paused and resumed when the Fragment is paused and resumed.

Using Fragments for Your To-Do List

The earlier to-do list example used a Linear Layout within an Activity to define its UI.

In this example you'll break the UI into a series of Fragments that represent its component pieces—the text entry box and the list of to-do items. This will enable you to easily create optimized layouts for different screen sizes.

1. Start by creating a new layout file, `new_item_fragment.xml` in the `res/layout` folder that contains the Edit Text node from the `main.xml`.

```
<?xml version="1.0" encoding="utf-8"?>
<EditText
xmlns:android="http://schemas.android.com/apk/res/android"
  android:id="@+id/myEditText"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:hint="@string/addItemHint"
  android:contentDescription="@string/addItemContentDescription"
/>
```

2. You'll need to create a new Fragment for each UI component. Start by creating a `NewItemFragment` that extends `Fragment`. Override the `onCreateView` handler to inflate the layout you created in step 1.

```
package com.paad.todolist;

import android.app.Activity;
import android.app.Fragment;
import android.view.KeyEvent;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.EditText;

public class NewItemFragment extends Fragment {
```

```
  @Override
  public View onCreateView(LayoutInflater inflater,
ViewGroup container,
    Bundle savedInstanceState) {
    return inflater.inflate(R.layout.new_item_fragment,
container, false);
  }

}
```

3. Each Fragment should encapsulate the functionality that it provides. In the case of the New Item Fragment, that's accepting new to-do items to add to your list. Start by defining an interface that the `ToDoListActivity` can implement to listen for new items being added.

```
public interface OnNewItemAddedListener {
  public void onNewItemAdded(String newItem);
}
```

4. Now create a variable to store a reference to the parent `ToDoListActivity` that will implement this interface. You can get the reference as soon as the parent Activity has been bound to the Fragment within the Fragment's `onAttach` handler.

```
private OnNewItemAddedListener onNewItemAddedListener;

@Override
public void onAttach(Activity activity) {
  super.onAttach(activity);

  try {
    onNewItemAddedListener =
(OnNewItemAddedListener)activity;
  } catch (ClassCastException e) {
    throw new ClassCastException(activity.toString() +
              " must implement OnNewItemAddedListener");
  }
}
```

5. Move the `editText.onClickListener`

implementation from the `ToDoListActivity` into your Fragment. When the user adds a new item, rather than adding the text directly to an array, pass it in to the parent Activity's `OnNewItemAddedListener.onNewItemAdded` implementation.

```
@Override
public View onCreateView(LayoutInflater inflater,
ViewGroup container,
  Bundle savedInstanceState) {
  View view =
inflater.inflate(R.layout.new_item_fragment, container,
false);

  final EditText myEditText =
    (EditText)view.findViewById(R.id.myEditText);

  myEditText.setOnKeyListener(new View.OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent
event) {
      if (event.getAction() == KeyEvent.ACTION_DOWN)
        if ((keyCode == KeyEvent.KEYCODE_DPAD_CENTER) ||
            (keyCode == KeyEvent.KEYCODE_ENTER)) {
          String newItem =
myEditText.getText().toString();
          onNewItemAddedListener.onNewItemAdded(newItem);
          myEditText.setText("");
          return true;
        }
      return false;
    }
  });

  return view;
}
```

6. Next, create the Fragment that contains the list of to-do items. Android provides a `ListFragment` class that you can use to easily create a simple List View based Fragment. Create a new class that Extends `ListFragment`.

```
package com.paad.todolist;
```

```
import android.app.ListFragment;

public class ToDoListFragment extends ListFragment {
}
```

The List Fragment class includes a default UI consisting of a single List View, which is sufficient for this example. You can easily customize the default List Fragment UI by creating your own custom layout and inflating it within the `onCreateView` handler. Any custom layout must include a List View node with the ID specified as `@android:id/list`.

7. With your Fragments completed, it's time to return to the Activity. Start by updating the `main.xml` layout, replacing the List View and Edit Text with the ToDo List Fragment and New Item Fragment, respectively.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <fragment
android:name="com.paad.todolist.NewItemFragment"
    android:id="@+id/NewItemFragment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
  <fragment
android:name="com.paad.todolist.ToDoListFragment"
    android:id="@+id/TodoListFragment"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```

8. Return to the `ToDoListActivity`. Within the `onCreate` method, use the Fragment Manager to get a reference to the ToDo List Fragment before creating

and assigning the adapter to it. Because the List View and Edit Text Views are now encapsulated within fragments, you no longer need to find references to them within your Activity. You'll need to expand the scope of the Array Adapter and Array List to class variables.

```
private ArrayAdapter<String> aa;
private ArrayList<String> todoItems;

public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Inflate your view
  setContentView(R.layout.main);

  // Get references to the Fragments
  FragmentManager fm = getFragmentManager();
  ToDoListFragment todoListFragment =

 (ToDoListFragment)fm.findFragmentById(R.id.TodoListFragment);

  // Create the array list of to do items
  todoItems = new ArrayList<String>();

  // Create the array adapter to bind the array to the
listview
  aa = new ArrayAdapter<String>(this,

 android.R.layout.simple_list_item_1,
                                todoItems);

  // Bind the array adapter to the listview.
  todoListFragment.setListAdapter(aa);
}
```

9. Your List View is now connected to your Array List using an adapter, so all that's left is to add any new items created within the New Item Fragment. Start by declaring that your class will implement the `OnNewItemAddedListener` you defined within the New Item Fragment in step 3.

```
public class ToDoList extends Activity
```

```
    implements NewItemFragment.OnNewItemAddedListener {
```

10. Finally, implement the listener by implementing an `onNewItemAdded` handler. Add the received string variable to the Array List before notifying the Array Adapter that the dataset has changed.

```
public void onNewItemAdded(String newItem) {
  todoItems.add(newItem);
  aa.notifyDataSetChanged();
}
```

All code snippets in this example are part of the Chapter 4 To-Do List Part 2 project, available for download at `www.wrox.com`.

The Android Widget Toolbox

Android supplies a toolbox of standard Views to help you create your UIs. By using these controls (and modifying or extending them, as necessary), you can simplify your development and provide consistency between applications.

The following list highlights some of the more familiar toolbox controls:

- `TextView`—A standard read-only text label that supports multiline display, string formatting, and automatic word wrapping.
- `EditText`—An editable text entry box that accepts multiline entry, word-wrapping, and hint text.
- `Chronometer`—A Text View extension that implements a simple count-up timer.
- `ListView`—A View Group that creates and manages a vertical list of Views, displaying them as rows within the list. The simplest List View displays the `toString` value of each object in an array, using a Text View for each item.
- `Spinner`—A composite control that displays a Text

View and an associated List View that lets you select an item from a list to display in the textbox. It's made from a Text View displaying the current selection, combined with a button that displays a selection dialog when pressed.

- `Button`—A standard push button.
- `ToggleButton`—A two-state button that can be used as an alternative to a check box. It's particularly appropriate where pressing the button will initiate an action as well as changing a state (such as when turning something on or off).
- `ImageButton`—A push button for which you can specify a customized background image (Drawable).
- `CheckBox`—A two-state button represented by a checked or unchecked box.
- `RadioButton`—A two-state grouped button. A group of these presents the user with a number of possible options, of which only one can be enabled at a time.
- `ViewFlipper`—A View Group that lets you define a collection of Views as a horizontal row in which only one View is visible at a time, and in which transitions between visible views can be animated.
- `VideoView`—Handles all state management and display Surface configuration for playing videos more simply from within your Activity.
- `QuickContactBadge`—Displays a badge showing the image icon assigned to a contact you specify using a phone number, name, email address, or URI. Clicking the image will display the quick contact bar, which

provides shortcuts for contacting the selected contact—including calling and sending an SMS, email, and IM.

- `ViewPager`—Released as part of the Compatibility Package, the View Pager implements a horizontally scrolling set of Views similar to the UI used in Google Play and Calendar. The View Pager allows users to swipe or drag left or right to switch between different Views.

This is only a selection of the widgets available. Android also supports several more advanced View implementations, including date-time pickers, auto-complete input boxes, maps, galleries, and tab sheets. For a more comprehensive list of the available widgets, head to http://developer.android.com/guide/tutorials/views/index.html.

Creating New Views

It's only a matter of time before you, as an innovative developer, encounter a situation in which none of the built-in controls meets your needs.

The ability to extend existing Views, assemble composite controls, and create unique new Views makes it possible to implement beautiful UIs optimized for your application's workflow. Android lets you subclass the existing View toolbox or implement your own View controls, giving you total freedom to tailor your UI to optimize the user experience.

When designing a UI, it's important to balance raw aesthetics and usability. With the power to create your own custom controls comes the temptation to rebuild all your controls from scratch. Resist that urge. The standard Views will be familiar to users from other Android applications and will update in line with new platform releases. On small screens, with users often paying limited attention, familiarity can often provide better usability than a slightly shinier control.

The best approach to use when creating a new View depends on what you want to achieve:

- **Modify or extend the appearance and/or**

behavior of an existing View when it supplies the basic functionality you want. By overriding the event handlers and/or `onDraw`, but still calling back to the superclass's methods, you can customize a View without having to re-implement its functionality. For example, you could customize a `TextView` to display numbers using a set number of decimal points.

- **Combine Views** to create atomic, reusable controls that leverage the functionality of several interconnected Views. For example, you could create a stopwatch timer by combining a `TextView` and a `Button` that resets the counter when clicked.
- **Create an entirely new control** when you need a completely different interface that you can't get by changing or combining existing controls.

Modifying Existing Views

The Android widget toolbox includes Views that provide many common UI requirements, but the controls are necessarily generic. By customizing these basic Views, you avoid re-implementing existing behavior while still tailoring the UI, and functionality, to your application's needs.

To create a new View based on an existing control, create a new class that extends it, as shown with the `TextView` derived class shown in Listing 4.11. In this example you extend the Text View to customize its appearance and behavior.

Listing 4.11: Extending Text View

```
import android.content.Context;
import android.graphics.Canvas;
import android.util.AttributeSet;
import android.view.KeyEvent;
import android.widget.TextView;

public class MyTextView extends TextView {

  public MyTextView (Context context, AttributeSet attrs, int
defStyle)
  {
    super(context, attrs, defStyle);
  }

  public MyTextView (Context context) {
    super(context);
  }

  public MyTextView (Context context, AttributeSet attrs) {
    super(context, attrs);
  }
}
```

code snippet PA4AD_Ch04_Views/src/MyTextView.java

To override the appearance or behavior of your new

View, override and extend the event handlers associated with the behavior you want to change.

In the following extension of the Listing 4.11 code, the `onDraw` method is overridden to modify the View's appearance, and the `onKeyDown` handler is overridden to allow custom key-press handling.

```
public class MyTextView extends TextView {

  public MyTextView (Context context, AttributeSet ats,
int defStyle) {
    super(context, ats, defStyle);
  }

  public MyTextView (Context context) {
    super(context);
  }

  public MyTextView (Context context, AttributeSet attrs)
{
    super(context, attrs);
  }

  @Override
  public void onDraw(Canvas canvas) {
    [ ... Draw things on the canvas under the text ... ]

    // Render the text as usual using the TextView base
class.
    super.onDraw(canvas);

    [ ... Draw things on the canvas over the text ... ]
  }

  @Override
  public boolean onKeyDown(int keyCode, KeyEvent
keyEvent) {
    [ ... Perform some special processing ... ]
    [ ... based on a particular key press ... ]

    // Use the existing functionality implemented by
    // the base class to respond to a key press event.
    return super.onKeyDown(keyCode, keyEvent);
  }
}
```

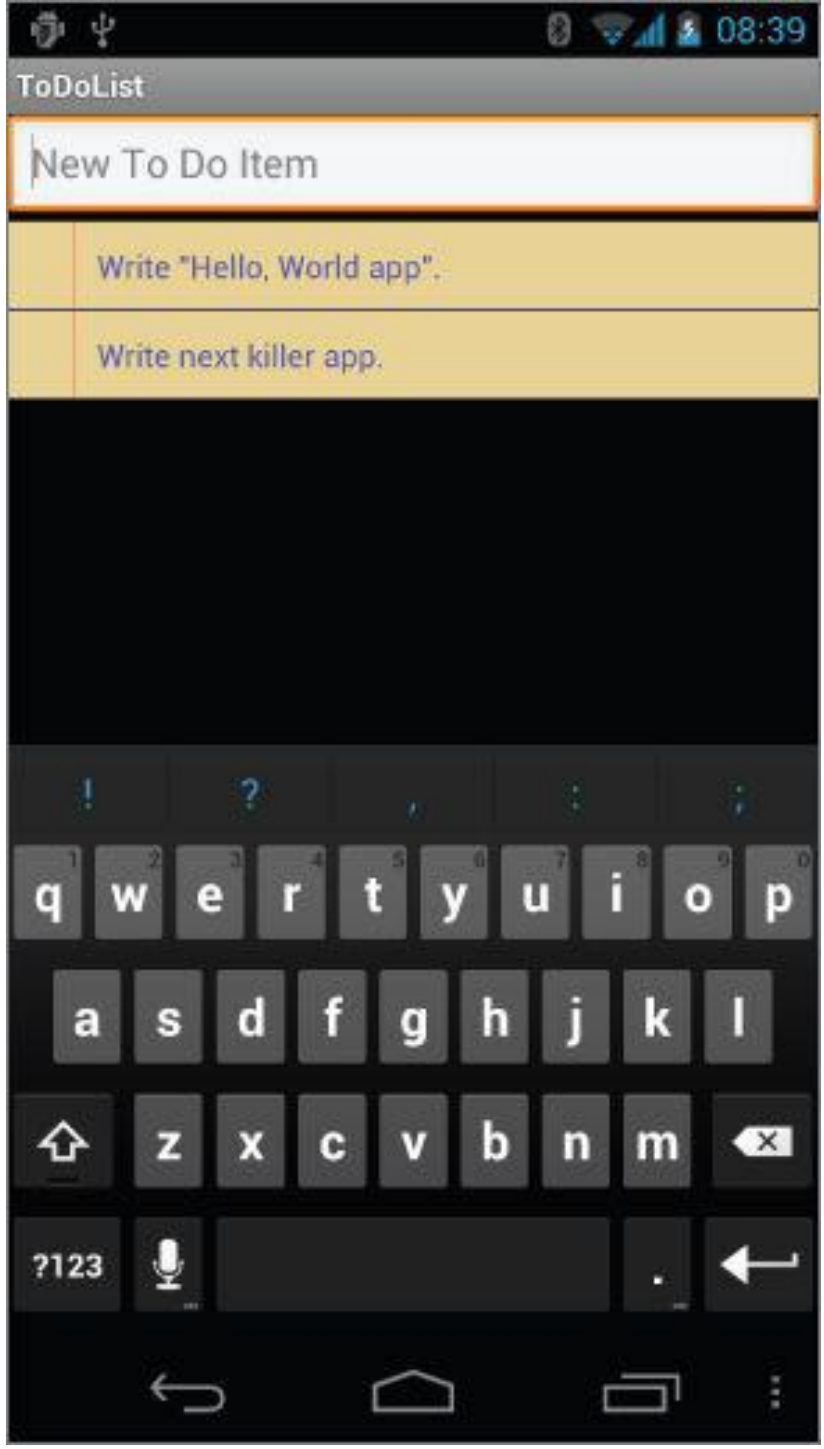
The event handlers available within Views are covered in more detail later in this chapter.

Customizing Your To-Do List

The to-do list example uses `TextView` controls to represent each row in a List View. You can customize the appearance of the list by extending Text View and overriding the `onDraw` method.

In this example you'll create a new `TodoListItemView` that will make each item appear as if on a paper pad. When complete, your customized to-do list should look like Figure 4.10.

Figure 4.10



1. Create a new `ToDoListItemView` class that `extends` `TextView`. Include a stub for overriding the `onDraw` method, and implement constructors that call a new `init` method stub.

```
package com.paad.todolist;

import android.content.Context;
import android.content.res.Resources;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.util.AttributeSet;
```

```
import android.widget.TextView;

public class ToDoListItemView extends TextView
{

  public ToDoListItemView (Context context,
AttributeSet ats, int ds) {
    super(context, ats, ds);
    init();
  }

  public ToDoListItemView (Context context) {
    super(context);
    init();
  }

  public ToDoListItemView (Context context,
AttributeSet attrs) {
    super(context, attrs);
    init();
  }

  private void init() {
  }

  @Override
  public void onDraw(Canvas canvas) {
    // Use the base TextView to render the
text.
    super.onDraw(canvas);
  }

}
```

2. Create a new `colors.xml` resource in the `res/values` folder. Create new color values for the paper, margin, line, and text colors.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color
name="notepad_paper">#EEF8E0A0</color>
  <color
name="notepad_lines">#FF0000FF</color>
  <color
name="notepad_margin">#90FF0000</color>
  <color name="notepad_text">#AA0000FF</color>
</resources>
```

3. Create a new `dimens.xml` resource file, and add a new value for the paper's margin width.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <dimen name="notepad_margin">30dp</dimen>
</resources>
```

4. With the resources defined, you're ready to customize the `ToDoListItemView` appearance. Create new private instance variables to store the `Paint` objects you'll use to draw the paper background and margin. Also create variables for the paper color and margin width values. Fill in the `init` method to get instances of the resources you created in the last two steps, and create the `Paint` objects.

```
private Paint marginPaint;
private Paint linePaint;
private int paperColor;
private float margin;

private void init() {
  // Get a reference to our resource table.
  Resources myResources = getResources();

  // Create the paint brushes we will use in
the onDraw method.
  marginPaint = new
Paint(Paint.ANTI_ALIAS_FLAG);
  marginPaint.setColor(myResources.getColor(R.color.notepad_margin));
  linePaint = new
Paint(Paint.ANTI_ALIAS_FLAG);
  linePaint.setColor(myResources.getColor(R.color.notepad_lines));

  // Get the paper background color and the
margin width.
  paperColor =
myResources.getColor(R.color.notepad_paper);
  margin =
myResources.getDimension(R.dimen.notepad_margin);
}
```

5. To draw the paper, override `onDraw` and draw the image using the `Paint` objects you created in step 4. After you've drawn the paper image, call the superclass's `onDraw` method and let it draw the text as usual.

```
@Override
public void onDraw(Canvas canvas) {
  // Color as paper
  canvas.drawColor(paperColor);

  // Draw ruled lines
  canvas.drawLine(0, 0, 0,
getMeasuredHeight(), linePaint);
  canvas.drawLine(0, getMeasuredHeight(),
```

```
                        getMeasuredWidth(),
getMeasuredHeight(),
                        linePaint);

    // Draw margin
    canvas.drawLine(margin, 0, margin,
getMeasuredHeight(), marginPaint);

    // Move the text across from the margin
    canvas.save();
    canvas.translate(margin, 0);

    // Use the TextView to render the text
    super.onDraw(canvas);
    canvas.restore();
  }
```

6. That completes the `ToDoListItemView` implementation. To use it in the To-Do List Activity, you need to include it in a new layout and pass that layout in to the Array Adapter constructor. Start by creating a new `todolist_item.xml` resource in the `res/layout` folder. It will specify how each of the to-do list items is displayed within the List View. For this example, your layout need only consist of the new `ToDoListItemView`, set to fill the entire available area.

```
<?xml version="1.0" encoding="utf-8"?>
<com.paad.todolist.ToDoListItemView
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:padding="10dp"
  android:scrollbars="vertical"
  android:textColor="@color/notepad_text"
  android:fadingEdge="vertical"
/>
```

7. The final step is to change the parameters passed in to the `ArrayAdapter` in `onCreate` of the `ToDoListActivity` class. Replace the reference to the default `android.R.layout.simple_list_item_1` with a reference to the new `R.layout.todolist_item` layout created in step 6.

```
int resID = R.layout.todolist_item;
aa = new ArrayAdapter<String>(this, resID,
todoItems);
```

Creating Compound Controls

All code snippets in this example are part of the *Chapter 4 To-do List Part 3* project, available for download at `www.wrox.com`.

Compound controls are atomic, self-contained View Groups that contain multiple child Views laid out and connected together.

When you create a compound control, you define the layout, appearance, and interaction of the Views it contains. You create compound controls by extending a `ViewGroup` (usually a layout). To create a new compound control, choose the layout class that's most suitable for positioning the child controls and extend it:

```
public class MyCompoundView extends LinearLayout {
  public MyCompoundView(Context context) {
    super(context);
  }

  public MyCompoundView(Context context, AttributeSet attrs) {
    super(context, attrs);
  }
}
```

As with Activities, the preferred way to design compound View UI layouts is by using an external resource.

Listing 4.12 shows the XML layout definition for a simple compound control consisting of an Edit Text for text entry, with a Clear Text button beneath it.

Listing 4.12: A compound View layout resource

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="wrap_content">
  <EditText
    android:id="@+id/editText"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
  />
  <Button
    android:id="@+id/clearButton"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Clear"
  />
</LinearLayout>
```

code snippet PA4AD_Ch04_Views/res/layout/clearable_edit_text.xml

To use this layout in your new compound View, override its constructor to inflate the layout resource using the `inflate` method from the `LayoutInflate` system service. The `inflate` method takes the layout resource and returns the inflated View.

For circumstances such as this, in which the returned View should be the class you're creating, you can pass in the parent View and attach the

result to it automatically.

Listing 4.13 demonstrates this using the `ClearableEditText` class. Within the constructor it inflates the layout resource from Listing 4.12 and then finds a reference to the Edit Text and Button Views it contains. It also makes a call to `hookupButton` that will later be used to hook up the plumbing that will implement the *clear text* functionality.

Listing 4.13: Constructing a compound View

```
public class ClearableEditText extends LinearLayout
{

  EditText editText;
  Button clearButton;

  public ClearableEditText(Context context) {
    super(context);

    // Inflate the view from the layout resource.
    String infService =
Context.LAYOUT_INFLATER_SERVICE;
    LayoutInflater li;
    li =
(LayoutInflater)getContext().getSystemService(infService);
    li.inflate(R.layout.clearable_edit_text, this,
true);

    // Get references to the child controls.
    editText =
(EditText)findViewById(R.id.editText);
    clearButton =
(Button)findViewById(R.id.clearButton);

    // Hook up the functionality
    hookupButton();
  }
}
```

code snippet
PA4AD_Ch04_Views/src/ClearableEditText.java

If you prefer to construct your layout in code, you can do so just as you would for an Activity:

```
public ClearableEditText(Context context) {
  super(context);

  // Set orientation of layout to vertical
  setOrientation(LinearLayout.VERTICAL);

  // Create the child controls.
  editText = new EditText(getContext());
  clearButton = new Button(getContext());
  clearButton.setText("Clear");

  // Lay them out in the compound control.
  int lHeight =
LinearLayout.LayoutParams.WRAP_CONTENT;
  int lWidth =
LinearLayout.LayoutParams.MATCH_PARENT;

  addView(editText, new
LinearLayout.LayoutParams(lWidth, lHeight));
  addView(clearButton, new
LinearLayout.LayoutParams(lWidth, lHeight));

  // Hook up the functionality
  hookupButton();
}
```

After constructing the View layout, you can hook up the event handlers for each child control to provide the functionality you need. In Listing 4.14, the `hookupButton` method is filled in to clear the Edit Text when the button is pressed.

Listing 4.14: Implementing the Clear Text Button

```
private void hookupButton() {
  clearButton.setOnClickListener(new
Button.OnClickListener() {
    public void onClick(View v) {
      editText.setText("");
    }
  });
}
```

code snippet PA4AD_Ch04_Views/src/ClearableEditText.java

Creating Simple Compound Controls Using Layouts

It's often sufficient, and more flexible, to define the layout and appearance of a set of Views without hard-wiring their interactions.

You can create a reusable layout by creating an XML resource that encapsulates the UI pattern you want to reuse. You can then import these layout patterns when creating the UI for Activities or Fragments by using the `include` tag within their layout resource definitions.

```
<include layout="@layout/clearable_edit_text"/>
```

The `include` tag also enables you to override the `id` and `layout` parameters of the root node of the included layout:

```
<include layout="@layout/clearable_edit_text"
         android:id="@+id/add_new_entry_input"
         android:layout_width="match_parent"
         android:layout_height="wrap_content"
         android:layout_gravity="top"/>
```

Creating Custom Views

Creating new Views gives you the power to fundamentally shape the way your applications look and feel. By creating your own controls, you can create UIs that are uniquely suited to your needs.

To create new controls from a blank canvas, you extend either the `View` or `SurfaceView` class. The `View` class provides a `Canvas` object with a series of draw methods and `Paint` classes. Use them to create a visual interface with bitmaps and raster graphics. You can then override user events, including screen touches or key presses to provide interactivity.

In situations in which extremely rapid repaints and 3D graphics aren't required, the `View` base class offers a powerful lightweight solution.

The `SurfaceView` class provides a `Surface` object that supports drawing from a background thread and optionally using `openGL` to implement your graphics. This is an excellent option for graphics-heavy controls that are frequently updated (such as live video) or that display complex graphical information (particularly, games and 3D visualizations).

This section focuses on building controls based on the View class. To learn more about the `SurfaceView` class and some of the more advanced Canvas paint features available in Android, see Chapter 10.

Creating a New Visual Interface

The base `View` class presents a distinctly empty 100-pixel-by-100-pixel square. To change the size of the control and display a more compelling visual interface, you need to override the `onMeasure` and `onDraw` methods.

Within `onMeasure` your View will determine the height and width it will occupy given a set of boundary conditions. The `onDraw` method is where you draw onto the Canvas.

Listing 4.15 shows the skeleton code for a new `View` class, which will be examined and developed further in the following sections.

Listing 4.15: Creating a new View

```
public class MyView extends View {

  // Constructor required for in-code creation
  public MyView(Context context) {
    super(context);
  }

  // Constructor required for inflation from resource file
  public MyView (Context context, AttributeSet ats, int
defaultStyle) {
    super(context, ats, defaultStyle );
  }

  //Constructor required for inflation from resource file
  public MyView (Context context, AttributeSet attrs) {
    super(context, attrs);
  }

  @Override
  protected void onMeasure(int wMeasureSpec, int hMeasureSpec)
{
    int measuredHeight = measureHeight(hMeasureSpec);
    int measuredWidth = measureWidth(wMeasureSpec);

    // MUST make this call to setMeasuredDimension
    // or you will cause a runtime exception when
    // the control is laid out.
    setMeasuredDimension(measuredHeight, measuredWidth);
  }

  private int measureHeight(int measureSpec) {
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

     [ ... Calculate the view height ... ]

     return specSize;
  }

  private int measureWidth(int measureSpec) {
    int specMode = MeasureSpec.getMode(measureSpec);
    int specSize = MeasureSpec.getSize(measureSpec);

     [ ... Calculate the view width ... ]

     return specSize;
  }

  @Override
  protected void onDraw(Canvas canvas) {
    [ ... Draw your visual interface ... ]
  }
}
```

code snippet PA4AD_Ch04_Views/src/MyView.java

The `onMeasure` method calls `setMeasuredDimension`. You must always call this method within your overridden `onMeasure` method; otherwise, your control will throw an exception when the parent container attempts to lay it out.

Drawing Your Control

The `onDraw` method is where the magic happens. If you're creating a new widget from scratch, it's because you want to create a completely new visual interface. The Canvas parameter in the `onDraw` method is the surface you'll use to bring your imagination to life.

The Android Canvas uses the *painter's algorithm*, meaning that each time you draw on to the canvas, it will cover anything previously drawn on the same area.

The drawing APIs provide a variety of tools to help draw your design on the Canvas using various `Paint` objects. The `Canvas` class includes helper methods for drawing primitive 2D objects, including circles, lines, rectangles, text, and Drawables (images). It also supports transformations that let you rotate, translate (move), and scale (resize) the Canvas while you draw on it.

When these tools are used in combination with `Drawables` and the `Paint` class (which offer a variety of customizable fills and pens), the complexity and detail that your control can render are limited only by the size of the screen and the power of the processor rendering it.

One of the most important techniques for writing efficient code in Android is to avoid the repetitive creation and destruction of objects. Any object created in your `onDraw` method will be created and destroyed every time the screen refreshes. Improve efficiency by making as many of these objects (particularly instances of `Paint` and `Drawable`) class-scoped and by moving their creation into the constructor.

Listing 4.16 shows how to override the `onDraw` method to display a simple text string in the center of the control.

Listing 4.16: Drawing a custom View

```
@Override
protected void onDraw(Canvas canvas) {
  // Get the size of the control based on the last call
to onMeasure.
  int height = getMeasuredHeight();
  int width = getMeasuredWidth();

  // Find the center
  int px = width/2;
  int py = height/2;

  // Create the new paint brushes.
  // NOTE: For efficiency this should be done in
```

```
  // the views's constructor
  Paint mTextPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
  mTextPaint.setColor(Color.WHITE);

  // Define the string.
  String displayText = "Hello World!";

  // Measure the width of the text string.
  float textWidth = mTextPaint.measureText(displayText);

  // Draw the text string in the center of the control.
  canvas.drawText(displayText, px-textWidth/2, py,
mTextPaint);
}
```

code snippet PA4AD_Ch04_Views/src/MyView.java

So that we don't diverge too far from the current topic, a more detailed look at the Canvas and Paint classes, and the techniques available for drawing more complex visuals is included in Chapter 10.

Android does not currently support vector graphics. As a result, changes to any element of your Canvas require that the entire Canvas be repainted; modifying the color of a brush will not change your View's display until the control is invalidated and redrawn. Alternatively, you can use OpenGL to render graphics. For more details, see the discussion on `SurfaceView` in Chapter 15, "Audio, Video, and Using the Camera."

Sizing Your Control

Unless you conveniently require a control that always occupies a space 100 pixels square, you will also need to override `onMeasure`.

The `onMeasure` method is called when the control's parent is laying out its child controls. It asks the question, "How much space will you use?" and passes in two parameters: `widthMeasureSpec` and `heightMeasureSpec`. These parameters specify the space available for the control and some metadata to describe that space.

Rather than return a result, you pass the View's height and width into the `setMeasuredDimension` method.

The following snippet shows how to override `onMeasure`. The calls to the local method stubs `measureHeight` and `measureWidth`, which are used to decode the `widthHeightSpec` and `heightMeasureSpec` values and calculate the preferred height and width values, respectively.

```
    @Override
```

```
protected void onMeasure(int widthMeasureSpec, int
heightMeasureSpec) {

  int measuredHeight =
measureHeight(heightMeasureSpec);
  int measuredWidth =
measureWidth(widthMeasureSpec);

  setMeasuredDimension(measuredHeight,
measuredWidth);
}

private int measureHeight(int measureSpec) {
  // Return measured widget height.
}

private int measureWidth(int measureSpec) {
  // Return measured widget width.
}
```

The boundary parameters, `widthMeasureSpec` and `heightMeasureSpec`, are passed in as integers for efficiency reasons. Before they can be used, they first need to be decoded using the static `getMode` and `getSize` methods from the `MeasureSpec` class.

```
int specMode =
MeasureSpec.getMode(measureSpec);
int specSize =
MeasureSpec.getSize(measureSpec);
```

Depending on the *mode* value, the *size* represents either the maximum space available for the control (in the case of `AT_MOST`), or the exact size that your control will occupy (for `EXACTLY`). In the case of `UNSPECIFIED`, your control does not have any reference for what the size represents.

By marking a measurement size as `EXACT`, the parent is insisting that the View will be placed into an area of the exact size specified. The `AT_MOST` mode says the parent is asking what size the View would like to occupy, given an upper boundary. In many cases the value you return will either be the same, or the size required to appropriately wrap the UI you want to display.

In either case, you should treat these limits as absolute. In some circumstances it may still be appropriate to return a measurement outside these limits, in which case you can let the parent choose how to deal with the oversized View, using techniques such as clipping and scrolling.

Listing 4.17 shows a typical

implementation for handling View measurements.

Listing 4.17: A typical View measurement implementation

```
@Override
protected void onMeasure(int
widthMeasureSpec, int heightMeasureSpec)
{
  int measuredHeight =
measureHeight(heightMeasureSpec);
  int measuredWidth =
measureWidth(widthMeasureSpec);

  setMeasuredDimension(measuredHeight,
measuredWidth);
}

private int measureHeight(int
measureSpec) {
  int specMode =
MeasureSpec.getMode(measureSpec);
  int specSize =
MeasureSpec.getSize(measureSpec);

  //  Default size if no limits are
specified.
  int result = 500;

  if (specMode == MeasureSpec.AT_MOST) {
    // Calculate the ideal size of your
    // control within this maximum size.
    // If your control fills the
available
    // space return the outer bound.
    result = specSize;
  } else if (specMode ==
MeasureSpec.EXACTLY) {
    // If your control can fit within
these bounds return that value.
    result = specSize;
  }
  return result;
}

private int measureWidth(int measureSpec)
{
  int specMode =
MeasureSpec.getMode(measureSpec);
  int specSize =
MeasureSpec.getSize(measureSpec);

  //  Default size if no limits are
specified.
  int result = 500;

  if (specMode == MeasureSpec.AT_MOST) {
    // Calculate the ideal size of your
control
    // within this maximum size.
    // If your control fills the
available space
    // return the outer bound.
    result = specSize;
  } else if (specMode ==
MeasureSpec.EXACTLY) {
    // If your control can fit within
these bounds return that value.
    result = specSize;
  }
```

```
  return result;
}
```

code snippet
PA4AD_Ch04_Views/src/MyView.java

Handling User Interaction Events

For your new View to be interactive, it will need to respond to user-initiated events such as key presses, screen touches, and button clicks. Android exposes several virtual event handlers that you can use to react to user input:

- `onKeyDown`—Called when any device key is pressed; includes the D-pad, keyboard, hang-up, call, back, and camera buttons
- `onKeyUp`—Called when a user releases a pressed key
- `onTrackballEvent`—Called when the device's trackball is moved
- `onTouchEvent`—Called when the touchscreen is pressed or released, or when it detects movement

Listing 4.18 shows a skeleton class that overrides each of the user interaction handlers in a View.

Listing 4.18: Input event handling for Views

```
@Override
public boolean onKeyDown(int
keyCode, KeyEvent keyEvent) {
  // Return true if the event was
handled.
  return true;
}

@Override
public boolean onKeyUp(int keyCode,
KeyEvent keyEvent) {
  // Return true if the event was
```

```
handled.
  return true;
}

@Override
public boolean
onTrackballEvent(MotionEvent event )
{
  // Get the type of action this
event represents
  int actionPerformed =
event.getAction();
  // Return true if the event was
handled.
  return true;
}

@Override
public boolean
onTouchEvent(MotionEvent event) {
  // Get the type of action this
event represents
  int actionPerformed =
event.getAction();
  // Return true if the event was
handled.
  return true;
}
```

code snippet
PA4AD_Ch04_Views/src/MyView.java

Further details on using each of these event handlers, including greater detail on the parameters received by each method and support for multitouch events, are available in Chapter 11.

Supporting Accessibility in Custom Views

Creating a custom View with a beautiful interface is only half the story. It's just as important to create accessible controls that can be used by users with disabilities that require them to interact with their devices in different ways.

Accessibility APIs were introduced in Android 1.6 (API level 4). They provide alternative interaction methods for users with visual, physical, or age-related disabilities that make it difficult to interact fully with a touchscreen.

The first step is to ensure that your custom View is accessible and navigable using the trackball and D-pad events, as described in the previous section. It's also important to use the content description attribute within your layout definition to describe the input widgets. (This is described in more detail in Chapter 11.)

To be accessible, custom Views must implement the `AccessibilityEventSource` interface and broadcast `AccessibilityEvents` using the `sendAccessibilityEvent` method.

The View class already implements the Accessibility Event Source interface, so you need to customize only the behavior to suit the functionality introduced by your custom View. Do this by passing the type of event that has occurred—usually one of clicks, long clicks, selection changes, focus changes, and text/content changes—to the `sendAccessibilityEvent` method. For custom Views that implement a completely new UI, this will typically include a broadcast whenever the displayed content changes, as shown in Listing 4.19.

Listing 4.19: Broadcasting Accessibility Events

```
public void setSeason(Season
_season) {
  season = _season;
  sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);
}
```

code snippet PA4AD_Ch04_Views/src/SeasonView.java

Clicks, long-clicks, and focus and selection changes typically will be broadcast by the underlying View implementation, although you should take care to broadcast any additional events not captured by the base View class.

The broadcast Accessibility Event includes a number of properties used by the accessibility service to augment the user experience. Several of these properties, including the View's class name and event timestamp, won't need to be altered; however, by overriding the `dispatchPopulateAccessibilityEvent` handler, you can customize details such as the textual representation of the View's contents, checked state, and selection state of your View, as shown in Listing 4.20.

Listing 4.20: Customizing Accessibility Event properties

```
@Override
public boolean
dispatchPopulateAccessibilityEvent(final
  AccessibilityEvent
event) {

  super.dispatchPopulateAccessibilityEvent(event);
  if (isShown()) {
    String seasonStr =
Season.valueOf(season);
    if
(seasonStr.length() >
AccessibilityEvent.MAX_TEXT_LENGTH)
```

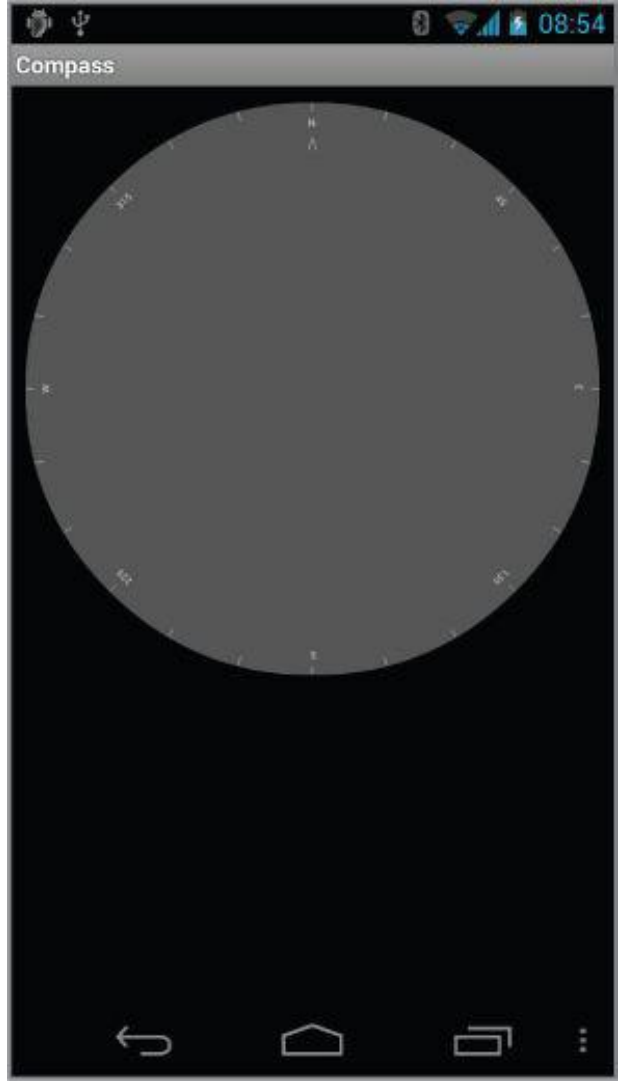
```
      seasonStr =
seasonStr.substring(0,
AccessibilityEvent.MAX_TEXT_LENGTH-
1);


 event.getText().add(seasonStr);
    return true;
  }
  else
    return false;
}
```

code snippet
PA4AD_Ch04_Views/src/SeasonView.java

Creating a Compass View Example

In the following example you'll create a new Compass View by extending the `View` class. This View will display a traditional compass rose to indicate a heading/orientation. When complete, it should appear as in Figure 4.11.

Figure 4.11



A compass is an example of a UI control that requires a radically different visual display from the Text Views and Buttons available in the SDK toolbox, making it an excellent candidate for building from scratch.

In Chapter 11 you will learn some advanced techniques for Canvas drawing that will let you dramatically improve its appearance. Then in Chapter 12, “Hardware Sensors,” you'll use this Compass View and

the device's built-in accelerometer to display the user's current orientation.

1. Create a new `Compass` project that will contain your new `CompassView`, and create a `CompassActivity` within which to display it. Within it, create a new `CompassView` class that extends `View` and add constructors that will allow the View to be instantiated, either in code or through inflation from a resource layout. Also add a new `initCompassView` method that will be used to initialize the control and call it from each constructor.

```
package
com.paad.compass;

import
android.content.Context;
import
android.content.res.Resources;
import
android.graphics.Canvas;
import
android.graphics.Paint;
import
android.util.AttributeSet;
import
android.view.View;
import
android.view.accessibility.AccessibilityEvent;

public class
CompassView
extends View {
  public
CompassView(Context
context) {

 super(context);

 initCompassView();
  }
```

```
  public
CompassView(Context
context,
AttributeSet
attrs) {

 super(context,
attrs);

 initCompassView();
  }

  public
CompassView(Context
context,


AttributeSet
ats,

      int
defaultStyle)
{

 super(context,
ats,
defaultStyle);

 initCompassView();
  }

  protected
void
initCompassView()
{

 setFocusable(true);
  }
}
```

2. The Compass View should always be a perfect circle that takes up as much of the canvas as this restriction allows. Override the `onMeasure` method to calculate the length of the shortest side, and use `setMeasuredDimension` to set the height and width using this value.

```
@Override
protected void
onMeasure(int
widthMeasureSpec,
int
heightMeasureSpec)
{
  // The
compass is a
```

```
circle that
fills as much
space as
possible.
  // Set the
measured
dimensions by
figuring out
the shortest
boundary,
  // height or
width.
  int
measuredWidth
=
measure(widthMeasureSpec);
  int
measuredHeight
=
measure(heightMeasureSpec);

  int d =
Math.min(measuredWidth,
measuredHeight);

  setMeasuredDimension(d,
d);
}

private int
measure(int
measureSpec) {
  int result =
0;

  // Decode
the
measurement
specifications.
  int specMode
=
MeasureSpec.getMode(measureSpec);
  int specSize
=
MeasureSpec.getSize(measureSpec);

  if (specMode
==
MeasureSpec.UNSPECIFIED)
{
    // Return
a default size
of 200 if no
bounds are
specified.
    result =
200;
  } else {
    // As you
want to fill
the available
space
    // always
return the
full available
bounds.
    result =
specSize;
  }
  return
result;
}
```

3. Modify the `main.xml` layout resource and replace the

TextView reference with your new CompassView:

```
<?xml
version="1.0"
encoding="utf-
8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <com.paad.compass.CompassView

  android:id="@+id/compassView"

  android:layout_width="match_parent"

  android:layout_height="match_parent"
  />
</FrameLayout>
```

4. Create two new resource files that store the colors and text strings you'll use to draw the compass.

1. Create the text string resources by modifying the res/values/strings.xml file.

```
<?xml
version="1.0"
encoding="utf-
8"?>
<resources>
  <string
name="app_name">Compass</string>
  <string
name="cardinal_north">N</string>
  <string
name="cardinal_east">E</string>
  <string
name="cardinal_south">S</string>
  <string
name="cardinal_west">W</string>
</resources>
```

2. Create the color resource res/values/colors.xml.

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

```
<resources>
  <color
name="background_color">#F555</color>
  <color
name="marker_color">#AFFF</color>
  <color
name="text_color">#AFFF</color>
</resources>
```

5. Return to the `CompassView` class. Add a new property to store the displayed bearing, and create `get` and `set` methods for it.

```
private float
bearing;

public void
setBearing(float
_bearing) {
  bearing =
_bearing;
}

public float
getBearing() {
  return
bearing;
}
```

6. Return to the `initCompassView` method and get references to each resource created in step 4. Store the string values as instance variables, and use the color values to create new class-scoped `Paint` objects. You'll use these objects in the next step to draw the compass face.

```
private Paint
markerPaint;
private Paint
textPaint;
private Paint
circlePaint;
private String
northString;
private String
```

```
eastString;
private String
southString;
private String
westString;
private int
textHeight;

protected void
initCompassView()
{
  setFocusable(true);

  Resources r
=
this.getResources();

  circlePaint
= new
Paint(Paint.ANTI_ALIAS_FLAG);
  circlePaint.setColor(r.getColor(R.color.background_color));
  circlePaint.setStrokeWidth(1);
  circlePaint.setStyle(Paint.Style.FILL_AND_STROKE);

  northString
=
r.getString(R.string.cardinal_north);
  eastString =
r.getString(R.string.cardinal_east);
  southString
=
r.getString(R.string.cardinal_south);
  westString =
r.getString(R.string.cardinal_west);

  textPaint =
new
Paint(Paint.ANTI_ALIAS_FLAG);
  textPaint.setColor(r.getColor(R.color.text_color));

  textHeight =
(int)textPaint.measureText("yY");

  markerPaint
= new
Paint(Paint.ANTI_ALIAS_FLAG);
  markerPaint.setColor(r.getColor(R.color.marker_color));
}
```

7. The next step is to draw the compass face using the `String` and `Paint` objects you created in step 6. The following code snippet is presented with only limited commentary. You can find more detail about drawing on the Canvas and using advanced

Paint effects in Chapter 11.

1. Start by overriding the `onDraw` method in the `CompassView` class.

```
@Override
protected void
onDraw(Canvas
canvas) {
```

2. Find the center of the control, and store the length of the smallest side as the compass's radius.

```
  int
mMeasuredWidth
=
getMeasuredWidth();
  int
mMeasuredHeight
=
getMeasuredHeight();

  int px
=
mMeasuredWidth
/ 2;
  int py
=
mMeasuredHeight
/ 2 ;

  int
radius =
Math.min(px,
py);
```

3. Draw the outer boundary, and color the background

of the Compass face using the `drawCircle` method. Use the `circlePaint` object you created in step 6.

```
  // Draw the background
  canvas.drawCircle(px, py, radius, circlePaint);
```

4. This Compass displays the current heading by rotating the face so that the current direction is always at the top of the device. To achieve this, rotate the

the canvas in the opposite direction to the current heading.

```
  // Rotate our perspective so
that the 'top' is
  // facing the current bearing.
  canvas.save();
  canvas.rotate(-bearing, px,
py);
```

5. All that's left is to draw the markings. Rotate the canvas through a full rotation, drawing markings every 15 degrees and the abbreviated direction string every 45 degrees.

```
  int textWidth =
(int)textPaint.measureText("W");
  int cardinalX = px-
textWidth/2;
  int cardinalY = py-
radius+textHeight;

  // Draw the marker every
15 degrees and text every
45.
  for (int i = 0; i < 24;
i++) {
    // Draw a marker.
    canvas.drawLine(px, py-
radius, px, py-radius+10,
markerPaint);

    canvas.save();
    canvas.translate(0,
textHeight);

    // Draw the cardinal
points
    if (i % 6 == 0) {
      String dirString =
"";
      switch (i) {
        case(0)  : {

dirString = northString;
                    int
arrowY = 2*textHeight;

canvas.drawLine(px, arrowY,
px-5, 3*textHeight,

          markerPaint);

canvas.drawLine(px, arrowY,
px+5, 3*textHeight,

          markerPaint);
                    break;
```

```
                    }
        case(6)  :
dirString = eastString;
break;
        case(12) :
dirString = southString;
break;
        case(18) :
dirString = westString;
break;
      }

 canvas.drawText(dirString,
cardinalX, cardinalY,
textPaint);
    }

    else if (i % 3 == 0) {
      // Draw the text
every alternate 45deg
      String angle =
String.valueOf(i*15);
      float angleTextWidth
=
textPaint.measureText(angle);

      int angleTextX =
(int)(px-angleTextWidth/2);
      int angleTextY = py-
radius+textHeight;

 canvas.drawText(angle,
angleTextX, angleTextY,
textPaint);
    }
    canvas.restore();

    canvas.rotate(15, px,
py);
  }
  canvas.restore();
}
```

8. The next step is to add accessibility support. The Compass View presents a heading visually, so to make it accessible you need to broadcast an accessibility event signifying that the “text” (in this case, content) has changed when the bearing changes. Do this by modifying the `setBearing` method.

```
public void
setBearing(float
```

```
_bearing) {
  bearing =
_bearing;
  sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_TEXT_CHANGED);
}
```

9. Override the `dispatchPopulateAccessibilityEvent` to use the current heading as the content value to be used for accessibility events.

```
@Override
public boolean
dispatchPopulateAccessibilityEvent(final
AccessibilityEvent
event) {
  super.dispatchPopulateAccessibilityEvent(event);
  if
(isShown()) {
    String
bearingStr =
String.valueOf(bearing);
    if
(bearingStr.length()
>
AccessibilityEvent.MAX_TEXT_LENGTH)

 bearingStr =
bearingStr.substring(0,
AccessibilityEvent.MAX_TEXT_LENGTH);


 event.getText().add(bearingStr);
    return
true;
  }
  else
    return
false;
}
```

All code snippets in this example are part of the Chapter 4 Compass project, available for download at `www.wrox.com`.

Run the Activity, and you should see the `CompassView` displayed. See Chapter 12 to learn how to bind the `CompassView` to the device's compass sensor.

Using Custom Controls

Having created your own custom Views, you can use them within code and layouts as you would any other View. Note that you must specify the fully qualified class name when you create a new node in the layout definition.

```
<com.paad.compass.CompassView
  android:id="@+id/compassView"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
/>
```

You can inflate the layout and get a reference to the `CompassView`, as usual, using the following code:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.main);
  CompassView cv =
(CompassView)this.findViewById(R.id.compassView);
  cv.setBearing(45);
}
```

You can also add your new view to a layout in code:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  CompassView cv = new CompassView(this);
  setContentView(cv);
  cv.setBearing(45);
}
```

Introducing Adapters

Adapters are used to bind data to View Groups that extend the `AdapterView` class (such as List View or Gallery). Adapters are responsible for creating child Views that represent the underlying data within the bound parent View.

You can create your own Adapter classes and build your own `AdapterView`-derived controls.

Introducing Some Native Adapters

In most cases you won't have to create your own Adapters from scratch. Android supplies a set of Adapters that can pump data from common data sources (including arrays and Cursors) into the native controls that extend Adapter View.

Because Adapters are responsible both for supplying the data and for creating the Views that represent each item, Adapters can radically modify the appearance and functionality of the controls they're bound to.

The following list highlights two of the most useful and versatile native Adapters:

- `ArrayAdapter`—The Array Adapter uses generics to bind an Adapter View to an array of objects of the specified class. By default, the Array Adapter uses the `toString` value of each object in the array to create and populate Text Views. Alternative constructors enable you to use more complex layouts, or you can extend the class (as shown in the next section) to bind data to more complicated layouts.
- `SimpleCursorAdapter`—The Simple Cursor Adapter enables you to bind the Views within a layout to specific columns contained within a Cursor (typically returned from a Content Provider query). You specify

an XML layout to inflate and populate to display each child, and then bind each column in the Cursor to a particular View within that layout. The adapter will create a new View for each Cursor entry and inflate the layout into it, populating each View within the layout using the Cursor's corresponding column value.

The following sections delve into these Adapter classes. The examples provided bind data to List Views, though the same logic will work just as well for other Adapter View classes, such as Spinners and Galleries.

Customizing the Array Adapter

By default, the Array Adapter uses the `toString` values of each item within an object array to populate a Text View within the layout you specify.

In most cases you will need to customize the Array Adapter to populate the layout used for each View to represent the underlying array data. To do so, extend `ArrayAdapter` with a type-specific variation, overriding the `getView` method to assign object properties to layout Views, as shown in Listing 4.21.

Listing 4.21: Customizing the Array Adapter

```
public class MyArrayAdapter extends ArrayAdapter<MyClass> {

  int resource;

  public MyArrayAdapter(Context context,
                        int _resource,
                        List<MyClass> items) {
    super(context, _resource, items);
    resource = _resource;
  }

  @Override
  public View getView(int position, View convertView,
ViewGroup parent) {
    // Create and inflate the View to display
```

```
    LinearLayout newView;

    if (convertView == null) {
      // Inflate a new view if this is not an update.
      newView = new LinearLayout(getContext());
      String inflater = Context.LAYOUT_INFLATER_SERVICE;
      LayoutInflater li;
      li =
(LayoutInflater)getContext().getSystemService(inflater);
      li.inflate(resource, newView, true);
    } else {
      // Otherwise we'll update the existing View
      newView = (LinearLayout)convertView;
    }

    MyClass classInstance = getItem(position);

    // TODO Retrieve values to display from the
    // classInstance variable.

    // TODO Get references to the Views to populate from the
layout.
    // TODO Populate the Views with object property values.

    return newView;
  }
}
```

code snippet PA4AD_Ch04_Adapters/src/MyArrayAdapter.java

The `getView` method is used to construct, inflate, and populate the View that will be added to the parent Adapter View class (e.g., List View), which is being bound to the underlying array using this Adapter.

The `getView` method receives parameters that describe the position of the item to be displayed, the View being updated (or `null`), and the View Group into which this new View will be placed. A call to `getItem` will return the value stored at the specified

index in the underlying array.

Return the newly created and populated (or updated) View instance as a result from this method.

Using Adapters to Bind Data to a View

To apply an Adapter to an `AdapterView`-derived class, call the View's `setAdapter` method, passing in an Adapter instance, as shown in Listing 4.22.

Listing 4.22: Creating and applying an Adapter

```
ArrayList<String> myStringArray = new ArrayList<String>();

int layoutID = android.R.layout.simple_list_item_1;

ArrayAdapter<String> myAdapterInstance;
myAdapterInstance =
  new ArrayAdapter<String>(this, layoutID, myStringArray);

myListView.setAdapter(myAdapterInstance);
```

code snippet PA4AD_Ch04_Adapters/src/MyActivity.java

This snippet shows the simplest case, in which the array being bound contains Strings and each List View item is represented by a single Text View.

The following example demonstrates how to bind an array of complex objects to a List View using a custom layout.

Customizing the To-Do List Array Adapter

This example extends the To-Do List project, storing each item as a `ToDoItem` object that includes the date each item was created.

You will extend `ArrayAdapter` to bind a collection of `ToDoItem` objects to the `ListView` and customize the layout used to display each to-do item within the List View.

1. Return to the To-Do List project. Create a new `ToDoItem` class that stores the task and its creation date. Override the `toString` method to return a summary of the item data.

```
package com.paad.todolist;

import java.text.SimpleDateFormat;
import java.util.Date;

public class ToDoItem {

  String task;
  Date created;

  public String getTask() {
    return task;
  }

  public Date getCreated() {
    return created;
  }

  public ToDoItem(String _task) {
    this(_task, new
Date(java.lang.System.currentTimeMillis()));
  }

  public ToDoItem(String _task, Date _created) {
    task = _task;
    created = _created;
  }

  @Override
  public String toString() {
    SimpleDateFormat sdf = new
SimpleDateFormat("dd/MM/yy");
    String dateString = sdf.format(created);
```

```
    return "(" + dateString + ") " + task;
  }
}
```

2. Open the `ToDoListActivity` and modify the `ArrayList` and `ArrayAdapter` variable types to store `ToDoItem` objects rather than Strings. You then need to modify the `onCreate` method to update the corresponding variable initialization.

```
private ArrayList<ToDoItem> todoItems;
private ArrayAdapter<ToDoItem> aa;

public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);

  // Inflate your view
  setContentView(R.layout.main);

  // Get references to the Fragments
  FragmentManager fm = getFragmentManager();
  ToDoListFragment todoListFragment =

 (ToDoListFragment)fm.findFragmentById(R.id.TodoListFragment);

  // Create the array list of to do items
  todoItems = new ArrayList<ToDoItem>();

  // Create the array adapter to bind the array to
the listview
  int resID = R.layout.todolist_item;
  aa = new ArrayAdapter<ToDoItem>(this, resID,
todoItems);

  // Bind the array adapter to the listview.
  todoListFragment.setListAdapter(aa);
}
```

3. Update the `onNewItemAdded` handler to support the `ToDoItem` objects.

```
public void onNewItemAdded(String newItem) {
  ToDoItem newTodoItem = new ToDoItem(newItem);
  todoItems.add(0, newTodoItem);
  aa.notifyDataSetChanged();
}
```

4. Now you can modify the `todolist_item.xml`

layout to display the additional information stored for each to-do item. Start by modifying the custom layout you created earlier in this chapter to include a second `TextView`. It will be used to show the creation date of each to-do item.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <TextView
    android:id="@+id/rowDate"
    android:background="@color/notepad_paper"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:padding="10dp"
    android:scrollbars="vertical"
    android:fadingEdge="vertical"
    android:textColor="#F000"
    android:layout_alignParentRight="true"
  />
  <com.paad.todolist.ToDoListItemView
    android:id="@+id/row"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="10dp"
    android:scrollbars="vertical"
    android:fadingEdge="vertical"
    android:textColor="@color/notepad_text"
    android:layout_toLeftOf="@+id/rowDate"
  />
</RelativeLayout>
```

6. To assign the ToDoItem values to each ListView Item, create a new class (`ToDoItemAdapter`) that extends an `ArrayAdapter` with a `ToDoItem`-specific variation. Override `getView` to assign the task and date properties in the `ToDoItem` object to the Views in the layout you created in step 4.

```
package com.paad.todolist;

import java.text.SimpleDateFormat;
import java.util.Date;
```

```
import java.util.List;
import android.content.Context;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ArrayAdapter;
import android.widget.LinearLayout;
import android.widget.TextView;

public class ToDoItemAdapter extends
ArrayAdapter<ToDoItem> {

  int resource;

  public ToDoItemAdapter(Context context,
                         int resource,
                         List<ToDoItem> items) {
    super(context, resource, items);
    this.resource = resource;
  }

  @Override
  public View getView(int position, View
convertView, ViewGroup parent) {
    LinearLayout todoView;

    ToDoItem item = getItem(position);

    String taskString = item.getTask();
    Date createdDate = item.getCreated();
    SimpleDateFormat sdf = new
SimpleDateFormat("dd/MM/yy");
    String dateString = sdf.format(createdDate);

    if (convertView == null) {
      todoView = new LinearLayout(getContext());
      String inflater =
Context.LAYOUT_INFLATER_SERVICE;
      LayoutInflater li;
      li =
(LayoutInflater)getContext().getSystemService(inflater);
      li.inflate(resource, todoView, true);
    } else {
      todoView = (LinearLayout) convertView;
    }

    TextView dateView =
(TextView)todoView.findViewById(R.id.rowDate);
    TextView taskView =
(TextView)todoView.findViewById(R.id.row);
```

```
        dateView.setText(dateString);
        taskView.setText(taskString);

        return todoView;
      }
    }
```

7. Return to the `ToDoListActivity` and replace the `ArrayAdapter` declaration with a `ToDoItemAdapter`:

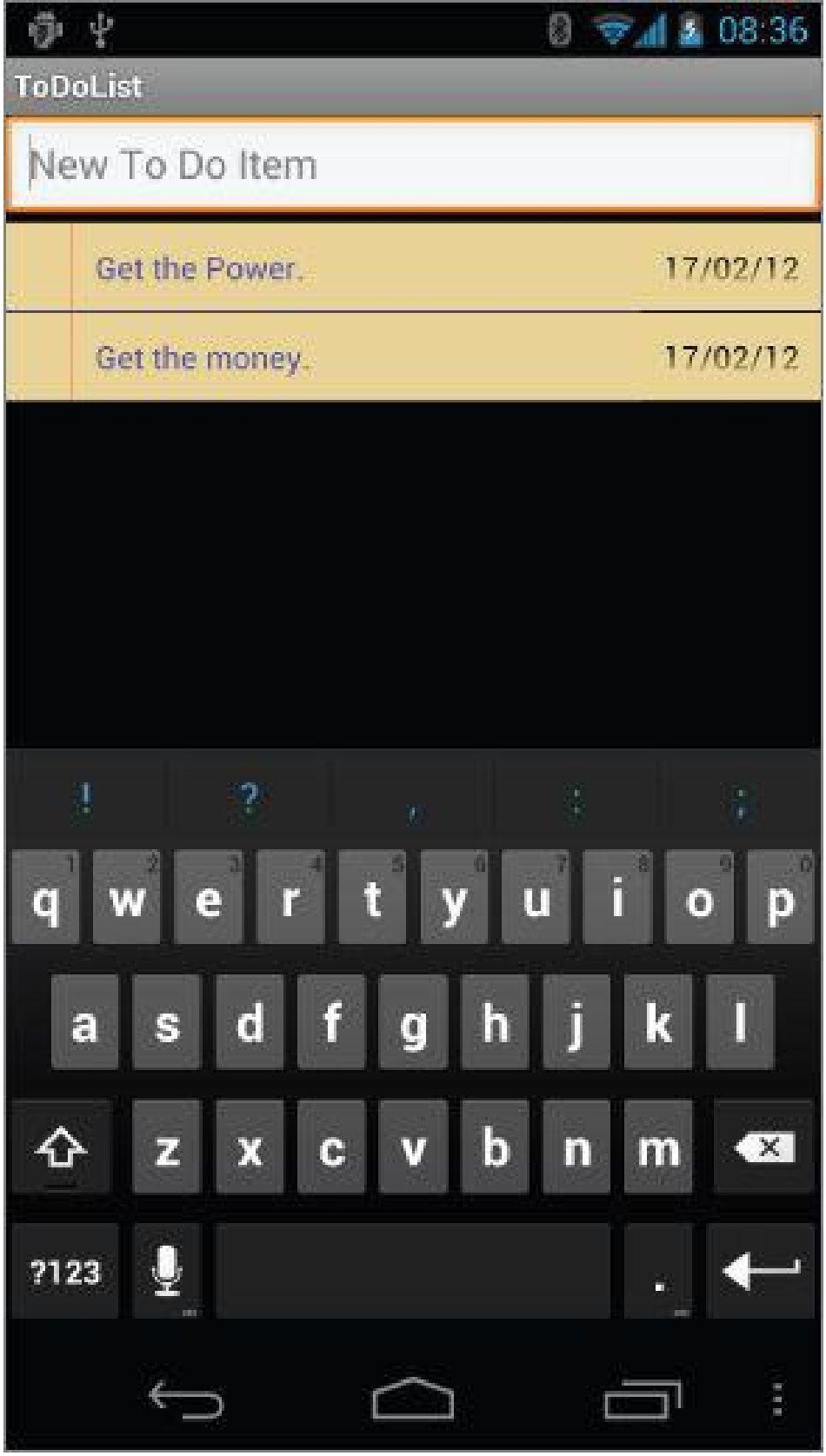
```
private ToDoItemAdapter aa;
```

8. Within `onCreate`, replace the `ArrayAdapter<ToDoItem>` instantiation with the new `ToDoItemAdapter`:

```
aa = new ToDoItemAdapter(this, resID, todoItems);
```

If you run your Activity and add some to-do items, it should appear as shown in Figure 4.12.

Figure 4.12



All code snippets in this example are part of the Chapter 4 To-do List Part 4 project, available for download at www.wrox.com.

Using the Simple Cursor Adapter

The `SimpleCursorAdapter` is used to bind a `Cursor` to an Adapter View using a layout to define the UI of each row/item. The content of each row's View is populated using the column values of the corresponding row in the underlying Cursor.

Construct a Simple Cursor Adapter by passing in the current context, a layout resource to use for each item, a Cursor that represents the data to display, and two integer arrays: one that contains the indexes of the columns from which to source the data, and a second (equally sized) array that contains resource IDs to specify which Views within the layout should be used to display the contents of the corresponding columns.

Listing 4.23 shows how to construct a Simple Cursor Adapter to display recent call information.

Listing 4.23: Creating a Simple Cursor Adapter

```
LoaderManager.LoaderCallbacks<Cursor> loaded =
  new LoaderManager.LoaderCallbacks<Cursor>() {

  public Loader<Cursor> onCreateLoader(int id, Bundle
args) {
    CursorLoader loader = new
CursorLoader(MyActivity.this,
      CallLog.CONTENT_URI, null, null, null, null);
    return loader;
  }
```

```
  public void onLoadFinished(Loader<Cursor> loader,
Cursor cursor) {

String[] fromColumns = new String[]
{CallLog.Calls.CACHED_NAME,

CallLog.Calls.NUMBER};

    int[] toLayoutIDs = new int[] { R.id.nameTextView,
R.id.numberTextView};

    SimpleCursorAdapter myAdapter;
    myAdapter = new SimpleCursorAdapter(MyActivity.this,

 R.layout.mysimplecursorlayout,
                                        cursor,
                                        fromColumns,
                                        toLayoutIDs);

    myListView.setAdapter(myAdapter);
  }

  public void onLoaderReset(Loader<Cursor> loader) {}

};

getLoaderManager().initLoader(0, null, loaded);
```

code snippet PA4AD_Ch4_Adapters/src/MyActivity.java

You'll learn more about Content Providers, Cursors, and Cursor Loaders in Chapter 8, “Databases and Content Providers,” where you'll also find more Simple Cursor Adapter examples.

Chapter 5

Intents and Broadcast Receivers

What's in this Chapter?

- Introducing Intents
- Starting Activities, sub-Activities, and Services using implicit and explicit Intents
- Using Linkify
- Broadcasting events using Broadcast Intents
- Using Pending Intents
- An introduction to Intent Filters and Broadcast Receivers
- Extending application functionality using Intent Filters
- Listening for Broadcast Intents
- Monitoring device state changes
- Managing manifest Receivers at run time

This chapter looks at Intents—probably the most unique and important concept in Android development. You'll learn

how to use Intents to broadcast data within and between applications and how to listen for them to detect changes in the system state.

You'll also learn how to define implicit and explicit Intents to start Activities or Services using late runtime binding. Using implicit Intents, you'll learn how to request that an action be performed on a piece of data, enabling Android to determine which application components can best service that request.

Broadcast Intents are used to announce events systemwide. You'll learn how to transmit these broadcasts and receive them using Broadcast Receivers.

Introducing Intents

Intents are used as a message-passing mechanism that works both within your application and between applications. You can use Intents to do the following:

- Explicitly start a particular Service or Activity using its class name
- Start an Activity or Service to perform an action with (or on) a particular piece of data
- Broadcast that an event has occurred

You can use Intents to support interaction among any of the application components installed on an Android device, no matter which application they're a part of. This turns your device from a platform containing a collection of independent components into a single, interconnected system.

One of the most common uses for Intents is to start new Activities, either *explicitly* (by specifying the class to load) or *implicitly* (by requesting that an action be performed on a piece of data). In the latter case the action does not need to be performed by an Activity within the calling application.

You can also use Intents to broadcast messages across

the system. Applications can register Broadcast Receivers to listen for, and react to, these Broadcast Intents. This enables you to create event-driven applications based on internal, system, or third-party application events.

Android broadcasts Intents to announce system events, such as changes in Internet connectivity or battery charge levels. The native Android applications, such as the Phone Dialer and SMS Manager, simply register components that listen for specific Broadcast Intents—such as “incoming phone call” or “SMS message received”—and react accordingly. As a result, you can replace many of the native applications by registering Broadcast Receivers that listen for the same Intents.

Using Intents, rather than explicitly loading classes, to propagate actions—even within the same application—is a fundamental Android design principle. It encourages the decoupling of components to allow the seamless replacement of application elements. It also provides the basis of a simple model for extending an application's functionality.

Using Intents to Launch Activities

The most common use of Intents is to bind your application components and communicate between them. Intents are used to start Activities, allowing you to create a workflow of different screens.

The instructions in this section refer to starting new Activities, but the same details also apply to Services. Details on starting (and creating) Services are available in Chapter 9, "Working in the Background."

To create and display an Activity, call `startActivity`, passing in an `Intent`, as follows:

```
startActivity(myIntent);
```

The `startActivity` method finds and starts the single Activity that best matches your Intent.

You can construct the Intent to explicitly specify the Activity class to open, or to include an action that the target Activity must be able to perform. In the latter case, the run time will choose an Activity dynamically using a process known as *intent resolution*.

When you use `startActivity`, your application won't receive any notification when the newly launched Activity finishes. To track feedback from a sub-Activity, use `startActivityForResult`, as described later in this chapter.

Explicitly Starting New Activities

You learned in Chapter 3, "Creating Applications and

Activities," that applications consist of a number of interrelated screens—Activities—that must be included in the application manifest. To transition between them, you will often need to explicitly specify which Activity to open.

To select a specific Activity class to start, create a new Intent, specifying the current Activity's Context and the class of the Activity to launch. Pass this Intent into `startActivity`, as shown in Listing 5.1.

Listing 5.1: Explicitly starting an Activity

```
Intent intent = new Intent(MyActivity.this,
MyOtherActivity.class);
startActivity(intent);
```

code snippet PA4AD_Ch05_Intents/src/MyActivity.java

After `startActivity` is called, the new Activity (in this example, `MyOtherActivity`) will be created, started, and resumed—moving to the top of the Activity stack.

Calling `finish` on the new Activity, or pressing the hardware back button, closes it and removes it from the stack. Alternatively, you can continue to navigate to other Activities by calling `startActivity`. Note that each time you call `startActivity`, a new Activity will be added to the stack; pressing back (or calling `finish`) will remove each of these Activities, in turn.

Implicit Intents and Late Runtime Binding

An implicit Intent is a mechanism that lets anonymous application components service action requests. That means you can ask the system to start an Activity to perform an action without knowing which application, or Activity, will be started.

For example, to let users make calls from your application, you could implement a new dialer, or you could use an implicit Intent that requests the action (dialing) be performed on a phone number (represented as a URI).

```
if (somethingWeird && itDontLookGood) {
  Intent intent =
    new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:555-2368"));

  startActivity(intent);
}
```

Android resolves this Intent and starts an Activity that provides the dial action on a telephone number—in this case, typically the Phone Dialer.

When constructing a new implicit Intent, you specify an action to perform and, optionally, supply the URI of the data on which to perform that action. You can send additional data to the target Activity by adding extras to the Intent.

Extras are a mechanism used to attach primitive values to an Intent. You can use the overloaded `putExtra` method on any Intent to attach a new name / value pair (NVP) that can then be retrieved using the corresponding `get[type]Extra` method in

the started Activity.

The extras are stored within the Intent as a Bundle object that can be retrieved using the `getExtras` method.

When you use an implicit Intent to start an Activity, Android will—at run time—resolve it into the Activity class best suited to performing the required action on the type of data specified. This means you can create projects that use functionality from other applications without knowing exactly which application you're borrowing functionality from ahead of time.

In circumstances where multiple Activities can potentially perform a given action, the user is presented with a choice. The process of intent resolution is determined through an analysis of the registered Broadcast Receivers, which are described in detail later in this chapter.

Various native applications provide Activities capable of performing actions against specific data. Third-party applications, including your own, can be registered to support new actions or to provide an alternative provider of native actions. You'll be introduced to some of the native actions, as well as how to register your own Activities to support them, later in this chapter.

Determining If an Intent Will Resolve

Incorporating the Activities and Services of a third-party application into your own is incredibly powerful; however, there is no guarantee that any particular application will be installed on a device, or that any application capable of handling your request is available.

As a result, it's good practice to determine if your call will resolve to an Activity *before* calling `startActivity`.

You can query the Package Manager to determine which, if any, Activity will be launched to service a specific Intent by calling `resolveActivity` on your Intent object, passing in the Package Manager, as shown in Listing 5.2.

Listing 5.2: Implicitly starting an Activity

```
if (somethingWeird && itDontLookGood) {
  // Create the impliciy Intent to use to
start a new Activity.
  Intent intent =
    new Intent(Intent.ACTION_DIAL,
Uri.parse("tel:555-2368"));

  // Check if an Activity exists to perform
this action.
  PackageManager pm = getPackageManager();
  ComponentName cn =
intent.resolveActivity(pm);
  if (cn == null) {
    // If there is no Activity available to
perform the action
    // Check to see if the Google Play Store
is available.
```

```
    Uri marketUri =
      Uri.parse("market://search?
q=pname:com.myapp.packagename");
    Intent marketIntent = new

 Intent(Intent.ACTION_VIEW).setData(marketUri);

    // If the Google Play Store is available,
use it to download an application
    // capable of performing the required
action. Otherwise log an
    // error.
    if (marketIntent.resolveActivity(pm) !=
null)
      startActivity(marketIntent);
    else
      Log.d(TAG, "Market client not
available.");
  }
  else
    startActivity(intent);
}
```

code snippet
PA4AD_Ch05_Intents/src/MyActivity.java

If no Activity is found, you can choose to either disable the related functionality (and associated user interface controls) or direct users to the appropriate application in the Google Play Store. Note that Google Play is not available on all devices, nor the emulator, so it's good practice to check for that as well.

Returning Results from Activities

An Activity started via `startActivity` is independent of its parent and will not provide any feedback when it closes.

Where feedback is required, you can start an Activity as a sub-Activity that can

pass results back to its parent. Sub-Activities are actually just Activities opened in a different way. As such, you must register them in the application manifest in the same way as any other Activity. Any manifest-registered Activity can be opened as a sub-Activity, including those provided by the system or third-party applications.

When a sub-Activity is finished, it triggers the `onActivityResult` event handler within the calling Activity. Sub-Activities are particularly useful in situations in which one Activity is providing data input for another, such as a user selecting an item from a list.

Launching Sub-Activities

The `startActivityForResult` method works much like `startActivity`, but with one important difference. In addition to passing in the explicit or implicit Intent used to determine which Activity to launch, you also pass in a *request code*. This value will later be used to uniquely identify the sub-Activity that has returned a result.

Listing 5.3 shows the skeleton code for launching a sub-Activity explicitly.

Listing 5.3: Explicitly starting a sub-Activity for a result

```
private static final int SHOW_SUBACTIVITY
= 1;

private void startSubActivity() {
  Intent intent = new Intent(this,
MyOtherActivity.class);
  startActivityForResult(intent,
SHOW_SUBACTIVITY);
}
```

code snippet
PA4AD_Ch05_Intents/src/MyActivity.java

Like regular Activities, you can start sub-Activities implicitly or explicitly. Listing 5.4 uses an implicit Intent to launch a new sub-Activity to pick a contact.

Listing 5.4: Implicitly starting a sub-Activity for a result

```
private static final int
PICK_CONTACT_SUBACTIVITY = 2;

private void
startSubActivityImplicitly() {
  Uri uri =
Uri.parse("content://contacts/people");
  Intent intent = new
Intent(Intent.ACTION_PICK, uri);
  startActivityForResult(intent,
PICK_CONTACT_SUBACTIVITY);
```

```
}
```

code snippet PA4AD_Ch05_Intents/src/MyActivity.java

Returning Results

When your sub-Activity is ready to return, call `setResult` before `finish` to return a result to the calling Activity.

The `setResult` method takes two parameters: the result code and the result data itself, represented as an Intent.

The result code is the "result" of running the sub-Activity—generally, either `Activity.RESULT_OK` or `Activity.RESULT_CANCELED`. In some circumstances, where OK and cancelled don't sufficiently or accurately describe the available return results, you'll want to use your own response codes to handle application-specific choices; `setResult` supports any integer value.

The Intent returned as a result often includes a data URI that points to a piece of content (such as the selected contact, phone number, or media file)

and a collection of extras used to return additional information.

Listing 5.5, taken from a sub-Activity's `onCreate` method, shows how an OK and Cancel button might return different results to the calling Activity.

Listing 5.5: Returning a result from a sub-Activity

```
Button okButton = (Button)
findViewById(R.id.ok_button);
okButton.setOnClickListener(new
View.OnClickListener() {
  public void onClick(View
view) {
    long selected_horse_id =
listView.getSelectedItemId();

    Uri selectedHorse =
Uri.parse("content://horses/"
+

    selected_horse_id);
    Intent result = new
Intent(Intent.ACTION_PICK,
selectedHorse);

    setResult(RESULT_OK,
result);
    finish();
  }
});

Button cancelButton = (Button)
findViewById(R.id.cancel_button);
cancelButton.setOnClickListener(new
View.OnClickListener() {
  public void onClick(View
view) {
```

```
    setResult(RESULT_CANCELED);
      finish();
    }
});
```

code snippet PA4AD_Ch05_Intents/src/SelectHorseActivity.java

If the Activity is closed by the user pressing the hardware back key, or `finish` is called without a prior call to `setResult`, the result code will be set to `RESULT_CANCELED` and the result Intent set to `null`.

Handling Sub-Activity Results

When a sub-Activity closes, the `onActivityResult` event handler is fired within the calling Activity. Override this method to handle the results returned by sub-Activities.

The `onActivityResult` handler receives a number of parameters:

- **Request code**—The request code that was used to

launch the returning sub-Activity.

- **Result code**—The result code set by the sub-Activity to indicate its result. It can be any integer value, but typically will be either `Activity.RESULT_OK` or `Activity.RESULT_CANCELED`.
- **Data**—An Intent used to package returned data. Depending on the purpose of the sub-Activity, it may include a URI that represents a selected piece of content. The sub-Activity can also return information as an extra within the returned data Intent.

If the sub-Activity closes abnormally or doesn't specify a result code before it closes, the result code is `Activity.RESULT_CANCELED`.

Listing 5.6 shows the

skeleton code for implementing the `onActivityResult` event handler within an Activity.

Listing 5.6: Implementing an On Activity Result handler

```
private static final int
SELECT_HORSE = 1;
private static final int
SELECT_GUN = 2;

Uri selectedHorse = null;
Uri selectedGun = null;

@Override
public void
onActivityResult(int
requestCode,

    int resultCode,

    Intent data) {

  super.onActivityResult(requestCode,
resultCode, data);

  switch(requestCode) {
    case (SELECT_HORSE):
      if (resultCode ==
Activity.RESULT_OK)
        selectedHorse =
data.getData();
      break;

    case (SELECT_GUN):
      if (resultCode ==
Activity.RESULT_OK)
        selectedGun =
data.getData();
      break;
```

```
    default: break;
  }
}
```

code snippet PA4AD_Ch05_Intents/src/MyActivity.java

Native Android Actions

Native Android applications also use Intents to launch Activities and sub-Activities.

The following (noncomprehensive) list shows some of the native actions available as static string constants in the `Intent` class. When creating implicit Intents, you can use these actions, known as *Activity Intents*, to start Activities and sub-Activities within your own applications.

Later you will be introduced to Intent Filters and how to

register your own Activities as handlers for these actions.

- `ACTION_ALL_APPS`— Opens an Activity that lists all the installed applications. Typically, this is handled by the launcher.
- `ACTION_ANSWER`— Opens an Activity that handles incoming calls. This is normally handled by the native in-call screen.
- `ACTION_BUG_REPORT`— Displays an Activity that can report a bug. This is normally handled by the native bug-reporting mechanism.
- `ACTION_CALL`— Brings up a phone dialer and immediately

initiates a call using the number supplied in the Intent's data URI. This action should be used only for Activities that replace the native dialer application. In most situations it is considered better form to use `ACTION_DIAL`.

- `ACTION_CALL_BUTTON`—Triggered when the user presses a hardware "call button." This typically initiates the dialer Activity.
- `ACTION_DELETE`—Starts an Activity that lets you delete the data specified at the Intent's data URI.
- `ACTION_DIAL`—Brings up a

dialer application with the number to dial prepopulated from the Intent's data URI. By default, this is handled by the native Android phone dialer. The dialer can normalize most number schemas—for example, `tel:555-1234` and `tel:(212) 555 1212` are both valid numbers.

- `ACTION_EDIT`—Requests an Activity that can edit the data at the Intent's data URI.
- `ACTION_INSERT`—Opens an Activity capable of inserting new items into the Cursor specified in the Intent's data

URI. When called as a sub-Activity, it should return a URI to the newly inserted item.

- `ACTION_PICK`—Launches a sub-Activity that lets you pick an item from the Content Provider specified by the Intent's data URI. When closed, it should return a URI to the item that was picked. The Activity launched depends on the data being picked—for example, passing `content://contacts/people` will invoke the native contacts list.
- `ACTION_SEARCH`—Typically used to launch a specific search

Activity. When it's fired without a specific Activity, the user will be prompted to select from all applications that support search. Supply the search term as a string in the Intent's extras using `SearchManager.QUERY` as the key.

- `ACTION_SEARCH_LONG_PRESS`—Enables you to intercept long presses on the hardware search key. This is typically handled by the system to provide a shortcut to a voice search.
- `ACTION_SENDTO`—Launches an Activity to send data to the contact specified by the Intent's data

URI.

- `ACTION_SEND`—Launches an Activity that sends the data specified in the Intent. The recipient contact needs to be selected by the resolved Activity. Use `setType` to set the MIME type of the transmitted data. The data itself should be stored as an extra by means of the key `EXTRA_TEXT` or `EXTRA_STREAM`, depending on the type. In the case of email, the native Android applications will also accept extras via the `EXTRA_EMAIL`, `EXTRA_CC`, `EXTRA_BCC`, and

`EXTRA_SUBJECT` keys. Use the `ACTION_SEND` action only to send data to a remote recipient (not to another application on the device).

- `ACTION_VIEW`—This is the most common generic action. View asks that the data supplied in the Intent's data URI be viewed in the most reasonable manner. Different applications will handle view requests depending on the URI schema of the data supplied. Natively `http:` addresses will open in the `browser`; `tel:` addresses will

open the dialer to call the number; `geo:` addresses will be displayed in the Google Maps application; and contact content will be displayed in the Contact Manager.

- `ACTION_WEB_SEARCH`—Opens the Browser to perform a web search based on the query supplied using the `SearchManager.QUERY` key.

In addition to these Activity actions, Android includes a large number of broadcast actions to create Intents that are broadcast to announce system events. These broadcast actions are

described later in this chapter.

Introducing Linkify

`Linkify` is a helper class that creates hyperlinks within Text View (and Text View-derived) classes through RegEx pattern matching.

Text that matches a specified RegEx pattern will be converted into a clickable hyperlink that implicitly fires `startActivity(new Intent(Intent.ACTION_VIEW, uri))`, using the matched text as the target URI.

You can specify any string pattern to be treated as a clickable link; for convenience, the `Linkify` class provides presets for common content types.

Native Linkify Link Types

The `Linkify` class has presets that can detect and linkify web URLs, email addresses, and phone numbers. To apply a preset, use the static `Linkify.addLinks` method, passing in a View to Linkify and a bitmask of one or more of the following self-describing `Linkify` class constants: `WEB_URLS`, `EMAIL_ADDRESSES`, `PHONE_NUMBERS`, and `ALL`.

```
TextView textView = (TextView)findViewById(R.id.myTextView);
Linkify.addLinks(textView,
Linkify.WEB_URLS|Linkify.EMAIL_ADDRESSES);
```

Most Android devices have at least two email applications: Gmail and Email. In situations in which multiple Activities are resolved as possible action consumers, users are asked to select their preference. In the case of the emulator, you must have the email client configured before it will respond to Linkified email addresses.

You can also linkify Views directly within a layout using the `android:autoLink` attribute. It supports one or more of the following values: `none`, `web`, `email`, `phone`, and `all`.

```
<TextView
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:text="@string/linkify_me"
  android:autoLink="phone|email"
/>
```

Creating Custom Link Strings

To linkify your own data, you need to define your own linkify strings. Do this by creating a new RegEx pattern that matches the text you want to display as hyperlinks.

As with the native types, you can linkify the target Text View by calling `Linkify.addLinks`; however, rather than passing in one of the preset constants, pass in your RegEx pattern. You can also pass in a prefix that will be prepended to the

target URI when a link is clicked.

Listing 5.7 shows a View being linkified to support earthquake data provided by an Android Content Provider (which you will create in Chapter 8, “Databases and Content Providers”). Note that rather than include the entire schema, the specified RegEx matches any text that starts with “quake” and is followed by a number, with optional whitespace. The full schema is then prepended to the URI before the Intent is fired.

Listing 5.7: Creating custom link strings in Linkify

```
// Define the base URI.
String baseUri =
"content://com.paad.earthquake/earthquakes/";

// Contruct an Intent to test if there is an
Activity capable of
// viewing the content you are Linkifying. Use the
Package Manager
// to perform the test.
PackageManager pm = getPackageManager();
Intent testIntent = new Intent(Intent.ACTION_VIEW,
Uri.parse(baseUri));
boolean activityExists =
testIntent.resolveActivity(pm) != null;

// If there is an Activity capable of viewing the
content
// Linkify the text.
```

```
if (activityExists) {
  int flags = Pattern.CASE_INSENSITIVE;
  Pattern p = Pattern.compile("\\bquake[\\s]?[0-
9]+\\b", flags);
  Linkify.addLinks(myTextView, p, baseUri);
}
```

code snippet PA4AD_Ch05_Linkify/src/MyActivity.java

Note that in this example, including whitespace between “quake” and a number will return a match, but the resulting URI won't be valid. You can implement and specify one or both of a `TransformFilter` and `MatchFilter` interface to resolve this problem. These interfaces, defined in detail in the following section, offer additional control over the target URI structure and the definition of matching strings, and are used as in the following skeleton code:

```
Linkify.addLinks(myTextView, p, baseUri,
                 new MyMatchFilter(), new
MyTransformFilter());
```

Using the Match Filter

To add additional conditions to RegEx pattern matches, implement the `acceptMatch` method in a Match Filter. When a potential match is found, `acceptMatch` is triggered, with the match

start and end index (along with the full text being searched) passed in as parameters.

Listing 5.8 shows a `MatchFilter` implementation that cancels any match immediately preceded by an exclamation mark.

Listing 5.8: Using a Linkify Match Filter

```
class MyMatchFilter implements
MatchFilter {
  public boolean acceptMatch(CharSequence
s, int start, int end) {
    return (start == 0 || s.charAt(start-
1) != '!');
  }
}
```

code snippet PA4AD_Ch05_Linkify/src/MyActivity.java

Using the Transform Filter

The Transform Filter lets you modify the implicit URI generated by

matching link text. Decoupling the link text from the target URI gives you more freedom in how you display data strings to your users.

To use the Transform Filter, implement the `transformUrl` method in your Transform Filter. When Linkify finds a successful match, it calls `transformUrl`, passing in the RegEx pattern used and the matched text string (before the base URI is prepended). You can modify the matched string and return it such that it can be appended to the base string as the data for a View Intent.

As shown in Listing 5.9, the `TransformFilter` implementation transforms the matched text into a lowercase URI, having also removed any whitespace characters.

Listing 5.9: Using a Linkify

Transform Filter

```
class MyTransformFilter implements
TransformFilter {
  public String transformUrl(Matcher
match, String url) {
    return
url.toLowerCase().replace(" ", "");
  }
}
```

code snippet PA4AD_Ch05_Linkify/src/MyActivity.java

Using Intents to Broadcast Events

So far, you've looked at using Intents to start new application components, but you can also use Intents to broadcast messages anonymously *between* components via the `sendBroadcast` method.

As a system-level message-passing mechanism, Intents are capable of sending structured messages across process boundaries. As a result, you can implement Broadcast Receivers to listen for, and respond to, these Broadcast Intents within your applications.

Broadcast Intents are used to notify applications of system or application events, extending the event-driven programming model between applications.

Broadcasting Intents helps make your application more open; by broadcasting an event using an Intent, you let yourself and third-party developers react to events without having to modify your original application. Within your applications you can listen for Broadcast Intents to to react to device state changes and third-party application events.

Android uses Broadcast Intents extensively to broadcast system events, such as changes in network connectivity, docking state, and incoming calls.

Broadcasting Events with Intents

Within your application, construct the Intent you want to broadcast and call `sendBroadcast` to send it.

Set the action, data, and category of your Intent in a way that lets Broadcast Receivers accurately determine their interest. In this scenario, the Intent *action* string is used to identify the event being broadcast, so it should be a unique string that identifies the event. By convention, action strings are constructed using the same form as Java package names:

```
public static final String NEW_LIFEFORM_DETECTED =
  "com.paad.action.NEW_LIFEFORM";
```

If you want to include data within the Intent, you can specify a URI using the Intent's `data` property. You can also include extras to add additional primitive values. Considered in terms of an event-driven paradigm, the extras equate to optional parameters passed into an event handler.

Listing 5.10 shows the basic creation of a Broadcast Intent using the action defined previously, with additional event information stored as extras.

Listing 5.10: Broadcasting an Intent

```
Intent intent = new
Intent(LifeformDetectedReceiver.NEW_LIFEFORM);
intent.putExtra(LifeformDetectedReceiver.EXTRA_LIFEFORM_NAME,
                detectedLifeform);
intent.putExtra(LifeformDetectedReceiver.EXTRA_LONGITUDE,
                currentLongitude);
intent.putExtra(LifeformDetectedReceiver.EXTRA_LATITUDE,
                currentLatitude);

sendBroadcast(intent);
```

code snippet
PA4AD_Ch05_BroadcastIntents/src/MyActivity.java

Listening for Broadcasts with Broadcast Receivers

Broadcast Receivers (commonly referred to simply as Receivers) are used to listen for Broadcast Intents. For a Receiver to receive broadcasts, it must be registered, either in code or within the application manifest—the latter case is referred to as a *manifest Receiver*. In either case, use an Intent Filter to specify which Intent actions and data your Receiver is listening for.

In the case of applications that include manifest Receivers, the applications don't have to be running when the Intent is broadcast for those

receivers to execute; they will be started automatically when a matching Intent is broadcast. This is excellent for resource management, as it lets you create event-driven applications that will still respond to broadcast events even after they've been closed or killed.

To create a new Broadcast Receiver, extend the `BroadcastReceiver` class and override the `onReceive` event handler:

```
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;

public class MyBroadcastReceiver extends
BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent
intent) {
    //TODO: React to the Intent received.
  }
}
```

The `onReceive` method will be executed on the main application thread when a Broadcast Intent is received that matches the Intent Filter used to register the Receiver. The `onReceive` handler must complete within five seconds; otherwise, the Force Close dialog will be displayed.

Typically, Broadcast Receivers will update content, launch Services, update Activity UI, or notify the user using the Notification Manager. The five-second execution limit ensures that major processing cannot, and should not, be done within the Broadcast Receiver itself.

Listing 5.11 shows how to implement a Broadcast Receiver that extracts the data and several extras from the broadcast Intent and uses them to start a new Activity. In the following sections you will learn how to register it in code or in your application manifest.

Listing 5.11: Implementing a Broadcast Receiver

```
public class LifeformDetectedReceiver
  extends BroadcastReceiver {

  public final static String
EXTRA_LIFEFORM_NAME
    = "EXTRA_LIFEFORM_NAME";
  public final static String EXTRA_LATITUDE =
"EXTRA_LATITUDE";
  public final static String EXTRA_LONGITUDE =
"EXTRA_LONGITUDE";

  public static final String
    ACTION_BURN =
"com.paad.alien.action.BURN_IT_WITH_FIRE";

  public static final String
    NEW_LIFEFORM =
"com.paad.alien.action.NEW_LIFEFORM";

  @Override
  public void onReceive(Context context,
Intent intent) {
    // Get the lifeform details from the
intent.
    Uri data = intent.getData();
    String type =
intent.getStringExtra(EXTRA_LIFEFORM_NAME);
    double lat =
intent.getDoubleExtra(EXTRA_LATITUDE, 0);
    double lng =
intent.getDoubleExtra(EXTRA_LONGITUDE, 0);
    Location loc = new Location("gps");
    loc.setLatitude(lat);
    loc.setLongitude(lng);
    if (type.equals("facehugger")) {
      Intent startIntent = new
Intent(ACTION_BURN, data);
      startIntent.putExtra(EXTRA_LATITUDE,
lat);
      startIntent.putExtra(EXTRA_LONGITUDE,
lng);

      context.startService(startIntent);
    }
  }
}
```

code snippet PA4AD_Ch05_BroadcastIntents/src/LifeformDetectedReceiver.java

Registering Broadcast Receivers in

Code

Broadcast Receivers that affect the UI of a particular Activity are typically registered in code. A Receiver registered programmatically will respond to Broadcast Intents only when the application component it is registered within is running.

This is useful when the Receiver is being used to update UI elements in an Activity. In this case, it's good practice to register the Receiver within the `onResume` handler and unregister it during `onPause`.

Listing 5.12 shows how to register and unregister a Broadcast Receiver in code using the `IntentFilter` class.

Listing 5.12: Registering and unregistering a Broadcast Receiver in code

```
private IntentFilter filter =
  new
IntentFilter(LifeformDetectedReceiver.NEW_LIFEFORM);

private LifeformDetectedReceiver receiver
=
  new LifeformDetectedReceiver();

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

  // Register the broadcast receiver.
  registerReceiver(receiver, filter);
}

@Override
public void onPause() {
  // Unregister the receiver
  unregisterReceiver(receiver);

  super.onPause();
}
```

code snippet
PA4AD_Ch05_BroadcastIntents/src/MyActivity.java

Registering Broadcast Receivers in Your Application Manifest

To include a Broadcast Receiver in the application manifest, add a `receiver` tag within the `application` node, specifying the class name of the Broadcast Receiver to register. The receiver node needs to include an `intent-filter` tag that specifies the action string being listened for.

```
<receiver
android:name=".LifeformDetectedReceiver">
  <intent-filter>
    <action
android:name="com.paad.alien.action.NEW_LIFEFORM"/>
  </intent-filter>
</receiver>
```

Broadcast Receivers registered this way are always active and will receive Broadcast Intents even when the application has been killed or hasn't been started.

Broadcasting Ordered Intents

When the order in which the Broadcast Receivers receive the Intent is important—particularly where you want to allow Receivers to affect the Broadcast Intent received by future Receivers—you can use

`sendOrderedBroadcast`, as follows:

```
String requiredPermission =
"com.paad.MY_BROADCAST_PERMISSION";
sendOrderedBroadcast(intent,
requiredPermission);
```

Using this method, your Intent will be delivered to all registered Receivers that hold the required permission (if one is specified) in the order of their specified priority. You can specify the priority of a Broadcast Receiver using the `android:priority` attribute within its Intent Filter manifest node, where higher values are considered higher priority.

```
<receiver
  android:name=".MyOrderedReceiver"
  android:permission="com.paad.MY_BROADCAST_PERMISSION">
  <intent-filter

 android:priority="100">
    <action
android:name="com.paad.action.ORDERED_BROADCAST"
/>
  </intent-filter>
</receiver>
```

It's good practice to send ordered broadcasts, and specify Receiver priorities, only for Receivers used within your application that specifically need to impose a specific order of receipt.

One common use-case for sending ordered broadcasts is to broadcast Intents

for which you want to receive result data. Using the `sendOrderedBroadcast` method, you can specify a Broadcast Receiver that will be placed at the end of the Receiver queue, ensuring that it will receive the Broadcast Intent after it has been handled (and modified) by the ordered set of registered Broadcast Receivers.

In this case, it's often useful to specify default values for the Intent result, data, and extras that may be modified by any of the Receivers that receive the broadcast before it is returned to the final result Receiver.

```
// Specify the
default result,
data, and extras.
// The may be
modified by any of
the Receivers who
handle the broadcast
// before being
received by the
final Receiver.
int initialResult =
Activity.RESULT_OK;
String initialData =
null;
String initialExtras
= null;

// A special Handler
instance on which to
receive the final
```

```
result.
// Specify null to
use the Context on
which the Intent was
broadcast.
Handler scheduler =
null;

sendOrderedBroadcast(intent,
requiredPermission,
finalResultReceiver,

scheduler,
initialResult,
initialData,
initialExtras);
```

Broadcasting Sticky Intents

Sticky Intents are useful variations of Broadcast Intents that persist the values associated with their last broadcast, returning them as an Intent when a new Receiver is registered to receive the broadcast.

When you call `registerReceiver`, specifying an Intent Filter that matches a sticky Broadcast Intent, the return value will be the last Intent broadcast, such as the

battery-level changed broadcast:

```
IntentFilter battery = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent currentBatteryCharge = registerReceiver(null, battery);
```

As shown in the preceding snippet, it is not necessary to specify a Receiver to obtain the current value of a sticky Intent. As a result, many of the system device state broadcasts (such as battery and docking state) use sticky Intents to improve efficiency. These are examined in more detail later in this chapter.

To broadcast your own

sticky Intents, your application must have the `BROADCAST_STICKY` uses-permission before calling `sendStickyBroadcast` and passing in the relevant Intent:

```
sendStickyBroadcast(intent);
```

To remove a sticky Intent, call `removeStickyBroadcast`, passing in the sticky Intent to remove:

```
removeStickyBroadcast(intent);
```

Introducing the Local Broadcast Manager

The Local Broadcast Manager was introduced to the Android Support Library to simplify the process of registering for, and sending, Broadcast Intents between components within your application.

Because of the reduced broadcast scope, using the Local Broadcast Manager is more efficient than sending a global broadcast. It also ensures that the Intent you broadcast cannot be received by any components outside your application, ensuring that there is no risk of leaking private or sensitive data, such as location information.

Similarly, other applications can't transmit broadcasts to your Receivers, negating the risk of these Receivers becoming vectors for security exploits.

To use the Local Broadcast Manager, you must first include the Android Support Library in your application, as described in Chapter 2.

Use the `LocalBroadcastManager.getInstance` method to return an instance of the Local Broadcast Manager:

```
LocalBroadcastManager lbm =
LocalBroadcastManager.getInstance(this);
```

To register a local broadcast Receiver, use the Local Broadcast Manager's `registerReceiver` method, much as you would register a global receiver,

passing in a Broadcast Receiver and an Intent Filter:

```
lbm.registerReceiver(new BroadcastReceiver() {
  @Override
  public void onReceive(Context context, Intent intent) {
    // TODO Handle the received local broadcast
  }
 }, new IntentFilter(LOCAL_ACTION));
```

Note that the Broadcast Receiver specified can also be used to handle global Intent broadcasts.

To transmit a local Broadcast Intent, use the Local Broadcast Manager's `sendBroadcast` method, passing in the Intent to broadcast:

```
lbm.sendBroadcast(new Intent(LOCAL_ACTION));
```

The Local Broadcast Manager also includes a `sendBroadcastSync` method that operates synchronously, blocking until each registered Receiver has been dispatched.

Introducing Pending Intents

The `PendingIntent` class provides a mechanism for creating Intents that can be fired on your application's behalf by another application at a later time.

A Pending Intent is commonly used to package Intents that will be fired in response to a future event, such as a Widget or Notification being clicked.

When used, Pending Intents execute the packaged Intent with the same permissions and identity as if you had executed them yourself, within your own application.

The `PendingIntent` class offers static methods to construct Pending Intents used to start an Activity, to start a Service, or to broadcast an Intent:

```
int requestCode = 0;
int flags = 0;

// Start an Activity
Intent startActivityIntent = new Intent(this,
MyOtherActivity.class);
PendingIntent.getActivity(this, requestCode,
                          startActivityIntent, flags);

// Start a Service
Intent startServiceIntent = new Intent(this, MyService.class);
PendingIntent.getService(this, requestCode,
                         startServiceIntent , flags);

// Broadcast an Intent
Intent broadcastIntent = new Intent(NEW LIFEFORM DETECTED);
```

```
PendingIntent.getBroadcast(this, requestCode,
                           broadcastIntent, flags);
```

The `PendingIntent` class includes static constants that can be used to specify flags to update or cancel any existing Pending Intent that matches your specified action, as well as to specify if this Intent is to be fired only once. The various options will be examined in more detail when Notifications and Widgets are introduced in Chapters 10 10 and 14, respectively.

Creating Intent Filters and Broadcast Receivers

Having learned to use Intents to start Activities/Services and to broadcast events, it's important to understand how to create the Broadcast Receivers and Intent Filters that listen for Broadcast Intents and allow your application to respond to them.

In the case of Activities and Services, an Intent is a request for an action to be performed on a set of data, and an Intent Filter is a declaration that a particular application component is capable of performing an action on a type of data.

Intent Filters are also used to specify the actions a Broadcast Receiver is interested in receiving.

Using Intent Filters to Service Implicit Intents

If an Intent is a request for an action to be performed on a set of data, how does Android know which application (and component) to use to service that request? Using Intent Filters, application components can declare the actions and data they support.

To register an Activity or Service as a potential Intent handler, add an `intent-filter` tag to its manifest node using the following tags (and associated attributes):

- `action`—Uses the `android:name` attribute to specify the name of the action being serviced. Each Intent Filter must have at least one action tag. Actions should be unique strings that are self-describing. Best practice is to use a naming system based on the Java package naming conventions.
- `category`—Uses the `android:name` attribute to specify under which circumstances the action should be serviced. Each Intent Filter tag can include multiple category tags. You can specify your own categories or use the following standard values provided by Android:
    - `ALTERNATIVE`—This category specifies that this action should be available as an alternative to the default action performed on an item of this data type. For example, where the default action for a contact is to view it, the alternative could be to edit it.
    - `SELECTED_ALTERNATIVE`—Similar to the `ALTERNATIVE` category, but whereas that category will always resolve to a single action using the intent resolution described next, `SELECTED_ALTERNATIVE` is used when a list of possibilities is required. As you'll see later in this chapter, one of the uses of Intent Filters is to help populate context menus dynamically using actions.
    - `BROWSABLE`—Specifies an action available from within the browser. When an Intent is fired from within the browser, it will always include the browsable category. If you want your application to respond to actions triggered within the browser (e.g., intercepting links to a particular

website), you must include the browsable category.
    - DEFAULT—Set this to make a component the default action for the data type specified in the Intent Filter. This is also necessary for Activities that are launched using an explicit Intent.
    - HOME—By setting an Intent Filter category as home without specifying an action, you are presenting it as an alternative to the native home screen.
    - LAUNCHER—Using this category makes an Activity appear in the application launcher.
- data—The data tag enables you to specify which data types your component can act on; you can include several data tags as appropriate. You can use any combination of the following attributes to specify the data your component supports:
    - android:host—Specifies a valid hostname (e.g., google.com).
    - android:mimetype—Specifies the type of data your component is capable of handling. For example, <type android:value="vnd.android.cursor.dir/*"/> would match any Android cursor.
    - android:path—Specifies valid path values for the URI (e.g., /transport/boats/).
    - android:port—Specifies valid ports for the specified host.
    - android:scheme—Requires a particular scheme (e.g., content or http).

The following snippet shows an Intent Filter for an Activity that can perform the SHOW_DAMAGE action as either a primary or an alternative action based on its mime type.

```
<intent-filter>
  <action

 android:name="com.paad.earthquake.intent.action.SHOW_DAMAGE"
  />
  <category android:name="android.intent.category.DEFAULT"/>
  <category

 android:name="android.intent.category.SELECTED_ALTERNATIVE"/>
  <data android:mimeType="vnd.earthquake.cursor.item/*"/>
</intent-filter>
```

You may have noticed that clicking a link to a YouTube video or Google Maps location on an Android device prompts you to use YouTube or Google Maps, respectively, rather than the browser. This is achieved by specifying the scheme, host, and

path attributes within the data tag of an Intent Filter, as shown in Listing 5.13. In this example, any link of the form that begins `http://blog.radioactiveyak.com` can be serviced by this Activity.

Listing 5.13: Registering an Activity as an Intent Receiver for viewing content from a specific website using an Intent Filter

```
<activity android:name=".MyBlogViewerActivity">
  <intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category
android:name="android.intent.category.DEFAULT" />
    <category
android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http"
          android:host="blog.radioactiveyak.com"/>
  </intent-filter>
</activity>
```

code snippet PA4AD_Ch05_Intents/AndroidManifest.xml

Note that you must include the *browsable* category in order for links clicked within the browser to trigger this behavior.

How Android Resolves Intent Filters

The process of deciding which Activity to start when an implicit Intent is passed in to `startActivity` is called *intent resolution*. The aim of intent resolution is to find the best Intent Filter match possible by means of the following process:

1. Android puts together a list of all the Intent Filters available from the installed packages.

2. Intent Filters that do not match the action *or* category associated with the Intent being resolved are removed from the list.

- Action matches are made only if the Intent Filter includes the specified action. An Intent Filter will fail the action match check if *none* of its actions matches the one specified by the Intent.
- For category matching, Intent Filters

must include *all* the categories defined in the resolving Intent, but can include additional categories not included in the Intent. An Intent Filter with no categories specified matches only Intents with no categories.

3. Each part of the Intent's data URI is compared to the Intent Filter's `data` tag. If the Intent Filter specifies a scheme, host/authority, path, or MIME type, these values are compared to the Intent's URI. Any mismatch will remove the Intent Filter from the list. Specifying no data values in an Intent Filter will result in a match with all Intent data values.

- The MIME type is the data type of the data being matched. When matching data types, you can use wildcards to match subtypes (e.g., `earthquakes/*`). If the Intent Filter specifies a data type, it must match the Intent; specifying no data types results in a match with all of them.
- The scheme is the “protocol” part of the URI (e.g., `http:`, `mailto:`, or `tel:`).
- The hostname or *data authority* is the section of the URI between the scheme and the path (e.g., `developer.android.com`). For a hostname to match, the Intent Filter's scheme must also pass.
- The data path is what comes after the authority (e.g., `/training`). A path can match only if the scheme and hostname parts of the data tag also match.

4. When you implicitly start an Activity, if more than one component is resolved from this process, all the matching possibilities are offered to the user. For Broadcast Receivers, each matching Receiver will receive the broadcast Intent.

Native Android application components are part of the intent-resolution process in exactly the

same way as third-party applications. They do not have a higher priority and can be completely replaced with new Activities that declare Intent Filters that service the same actions.

Finding and Using Intents Received Within an Activity

When an application component is started through an implicit Intent, it needs to find the action it's to perform and the data to perform it on.

To find the Intent used to start the Activity, call `getIntent`, as shown in Listing 5.14.

Listing 5.14: Finding the launch Intent in an Activity

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

  Intent intent = getIntent();
  String action = intent.getAction();
  Uri data = intent.getData();
}
```

code snippet
PA4AD_Ch05_Intents/src/MyOtherActivity.java

Use the `getData` and `getAction` methods to find the data and action, respectively, associated with the Intent. Use the type-safe `get<type>Extra` methods to extract additional information stored in its extras Bundle.

The `getIntent` method will always return the initial Intent used to create the Activity. In some circumstances your Activity may continue to receive Intents after it has been launched. You can use widgets and Notifications to provide shortcuts to displaying data within your Activity that may still be running, though not visible.

Override the `onNewIntent` handler within your Activity to receive and handle new Intents after the Activity has been created.

```
@Override
public void onNewIntent(Intent newIntent) {
  // TODO React to the new Intent
  super.onNewIntent(newIntent);
}
```

Passing on Responsibility

To pass responsibility for action handling to the next best Activity, use `startNextMatchingActivity`.

```
Intent intent = getIntent();
if (isDuringBreak)
  startNextMatchingActivity(intent);
```

This lets you add additional conditions to your components that restrict their use beyond the ability of the Intent Filter-based intent-resolution process.

Selecting a Contact Example

In this example you'll create a new Activity that services `ACTION_PICK` for contact data. It displays each of the contacts in the contacts database and lets the user select one, before closing and returning the selected contact's URI to the calling Activity.

This example is somewhat contrived. Android already supplies an Intent Filter for picking a contact from a list that can be invoked by means of the `content://contacts/people/` URI in an implicit Intent. The purpose of this exercise is to demonstrate the form, even if this particular implementation isn't particularly useful.

1. Create a new ContactPicker project that includes a `ContactPicker` Activity:

```
package
com.paad.contactpicker;

import android.app.Activity;
import android.content.Intent;
import
```

```
android.database.Cursor;
import android.net.Uri;
import android.os.Bundle;
import
android.provider.ContactsContract.Contacts;
import android.view.View;
import
android.widget.AdapterView;
import
android.widget.ListView;
import
android.widget.SimpleCursorAdapter;

public class ContactPicker
extends Activity {
  @Override
  public void onCreate(Bundle
savedInstanceState) {

 super.onCreate(savedInstanceState);

 setContentView(R.layout.main);
  }
}
```

2. Modify the `main.xml` layout resource to include a single `ListView` control. This control will be used to display the contacts.

```
<?xml version="1.0"
encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <ListView
android:id="@+id/contactListView"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"
  />
</LinearLayout>
```

3. Create a new `listitemlayout.xml` layout resource that includes a single `TextView` control. This control will be used to display each contact in the List View.

```
<?xml version="1.0"
encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView

 android:id="@+id/itemTextView"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"
    android:padding="10dp"
    android:textSize="16dp"
```

```
    android:textColor="#FFF"
  />
</LinearLayout>
```

4. Return to the `ContactPicker` Activity. Override the `onCreate` method.

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

4.1 Create a new Cursor to retrieve the people stored in the contact list, and bind it to the List View using a `SimpleCursorArrayAdapter`. Note that in this example the query is executed on the main UI thread. A better approach would be to use a Cursor Loader, as shown in Chapter 8.

```
  final Cursor c =
getContentResolver().query(

 ContactsContract.Contacts.CONTENT_URI,
null, null, null, null);

  String[] from = new
String[] {
Contacts.DISPLAY_NAME_PRIMARY
};
  int[] to = new int[] {
R.id.itemTextView };

  SimpleCursorAdapter
adapter = new
SimpleCursorAdapter(this,



 R.layout.listitemlayout,


     c,


     from,


     to);
  ListView lv =
(ListView)findViewById(R.id.contactListView);
  lv.setAdapter(adapter);
```

4.2 Add an `onItemClickListener` to the List View.

Selecting a contact from the list should return a path to the item to the calling Activity.

```
  lv.setOnItemClickListener(new
ListView.OnItemClickListener()
{
    public void
onItemClick(AdapterView<?
> parent, View view,
int pos,

       long id) {
      // Move the
cursor to the
selected item

 c.moveToPosition(pos);
      // Extract the
row id.
      int rowId =
c.getInt(c.getColumnIndexOrThrow("_id"));
      // Construct
the result URI.
      Uri outURI =
  ContentUris.withAppendedId(ContactsContract.Contacts.CONTENT_URI,
rowId);
      Intent outData
= new Intent();

 outData.setData(outURI);

 setResult(Activity.RESULT_OK,
outData);
      finish();
    }
  });
```

c. Close off the `onCreate` method:

```
}
```

5. Modify the application manifest and replace the `intent-filter` tag of the Activity to add support for the `ACTION_PICK` action on contact data:

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

 package="com.paad.contactpicker">
    <application
android:icon="@drawable/ic_launcher">
        <activity
android:name=".ContactPicker"
android:label="@string/app_name">
            <intent-filter>
              <action
android:name="android.intent.action.PICK"></action>
```

```
                <category
android:name="android.intent.category.DEFAULT"></category>
                <data
android:path="contacts"
android:scheme="content"></data>
              </intent-filter>
          </activity>
      </application>
</manifest>
```

6. This completes the sub-Activity. To test it, create a new test harness `ContactPickerTester` Activity. Create a new layout resource —`contactpickertester.xml`— that includes a `TextView` to display the selected contact and a `Button` to start the sub-Activity:

```
<?xml version="1.0"
encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <TextView

 android:id="@+id/selected_contact_textview"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"
  />
  <Button

 android:id="@+id/pick_contact_button"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"
    android:text="Pick
Contact"
  />
</LinearLayout>
```

7. Override the `onCreate` method of the `ContactPickerTester` to add a click listener to the Button so that it implicitly starts a new sub-Activity by specifying the `ACTION_PICK` and the contact database URI (`content://contacts/`):

```
package
com.paad.contactpicker;

import android.app.Activity;
import android.content.Intent;
import
android.database.Cursor;
```

```
import android.net.Uri;
import android.os.Bundle;
import
android.provider.ContactsContract;
import android.view.View;
import
android.view.View.OnClickListener;
import android.widget.Button;
import
android.widget.TextView;

public class
ContactPickerTester extends
Activity {

  public static final int
PICK_CONTACT = 1;

  @Override
  public void onCreate(Bundle
savedInstanceState) {

 super.onCreate(savedInstanceState);

 setContentView(R.layout.contactpickertester);

    Button button =
(Button)findViewById(R.id.pick_contact_button);


 button.setOnClickListener(new
OnClickListener() {
     @Override
     public void onClick(View
_view) {
        Intent intent = new
Intent(Intent.ACTION_PICK,

 Uri.parse("content://contacts/"));

 startActivityForResult(intent,
PICK_CONTACT);
      }
    });
  }
}
```

8. When the sub-Activity returns, use the result to populate the Text View with the selected contact's name:

```
@Override
public void
onActivityResult(int reqCode,
int resCode, Intent data) {
  super.onActivityResult(reqCode,
resCode, data);

  switch(reqCode) {
    case (PICK_CONTACT) : {
      if (resCode ==
Activity.RESULT_OK) {
        Uri contactData =
data.getData();
        Cursor c =
getContentResolver().query(contactData,
null, null, null, null);
        c.moveToFirst();
        String name =
c.getString(c.getColumnIndexOrThrow(

 ContactsContract.Contacts.DISPLAY_NAME_PRIMARY));
```

```
            c.close();
            TextView tv =
      (TextView)findViewById(R.id.selected_contact_textview);
            tv.setText(name);
          }
          break;
        }
        default: break;
      }
    }
```

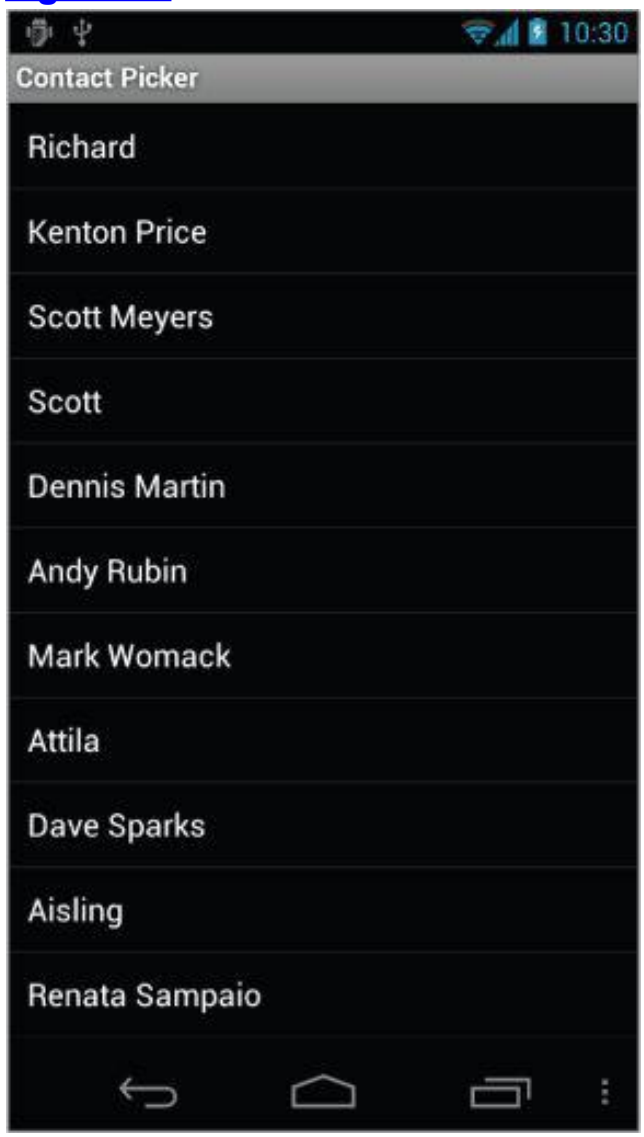
9. With your test harness complete, simply add it to your application manifest. You'll also need to add a `READ_CONTACTS` permission within a `uses-permission` tag to allow the application to access the contacts database.

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

 package="com.paad.contactpicker">
    <uses-permission
android:name="android.permission.READ_CONTACTS"/>
    <application
android:icon="@drawable/ic_launcher">
      <activity
android:name=".ContactPicker"
android:label="@string/app_name">
        <intent-filter>
          <action
android:name="android.intent.action.PICK"></action>
          <category
android:name="android.intent.category.DEFAULT"></category>
          <data
android:path="contacts"
android:scheme="content"></data>
        </intent-filter>
      </activity>
      <activity
android:name=".ContactPickerTester"

 android:label="Contact Picker
Test">
        <intent-filter>
          <action
android:name="android.intent.action.MAIN"
/>
          <category
android:name="android.intent.category.LAUNCHER"
/>
        </intent-filter>
      </activity>
    </application>

</manifest>
```

All code snippets in this example are part of the *Chapter* 5 *Contact Picker* project, available for download at Wrox.com.

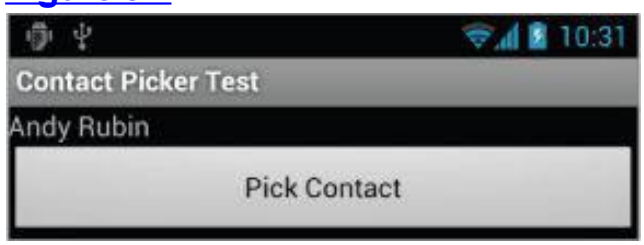
When your Activity is running, press the “pick contact” button. The contact picker Activity should appear, as shown in Figure 5.1.

Figure 5.1



After you select a contact, the parent Activity should return to the foreground with the selected contact name displayed (see Figure 5.2).

Figure 5.2



Using Intent Filters for Plug-Ins and Extensibility

Having used Intent Filters to declare the actions your Activities can perform on different types of data, it stands to reason that applications can also query to find which actions are available to be performed on a particular piece of data.

Android provides a plug-in model that lets your applications take advantage of functionality, provided anonymously from your own or third-party application components you haven't yet conceived of, without your having to modify or recompile your projects.

Supplying Anonymous Actions to Applications

To use this mechanism to make your Activity's actions available anonymously for existing applications, publish them using `intent-filter` tags within their manifest nodes, as described earlier.

The Intent Filter describes the `action` it performs and the `data` upon which it can be performed. The latter will be used during the intent-resolution process to determine when this action should be available. The `category` tag must be either `ALTERNATIVE` or `SELECTED_ALTERNATIVE`, or both. The `android:label` attribute should be a human-readable label that describes the action.

Listing 5.15 shows an example of an Intent Filter used to advertise an Activity's capability to nuke Moon bases from orbit.

Listing 5.15: Advertising supported Activity actions

```
<activity android:name=".NostromoController">
  <intent-filter
    android:label="@string/Nuke_From_Orbit">
    <action android:name="com.pad.nostromo.NUKE_FROM_ORBIT"/>
    <data android:mimeType="vnd.moonbase.cursor.item/*"/>
    <category
android:name="android.intent.category.ALTERNATIVE"/>
    <category

 android:name="android.intent.category.SELECTED_ALTERNATIVE"
    />
  </intent-filter>
</activity>
```

code snippet PA4AD_Ch05_Intents/AndroidManifest.xml

Discovering New Actions from Third-Party Intent Receivers

Using the Package Manager, you can create an Intent that specifies a type of data and a category of action, and have the system return a list of Activities capable of performing an action on that data.

The elegance of this concept is best explained by an example. If the data your Activity displays is a list of places, you might include functionality to View them on a map or “Show directions to” each. Jump a few years ahead and you've created an application that interfaces with your car, allowing your phone to handle driving. Thanks to the runtime menu generation, when a new Intent Filter—with a `DRIVE_CAR` action—is included within the new Activity's node, Android will

resolve this new action and make it available to your earlier application.

This provides you with the ability to retrofit functionality to your application when you create new components capable of performing actions on a given type of data. Many of Android's native applications use this functionality, enabling you to provide additional actions to native Activities.

The Intent you create will be used to resolve components with Intent Filters that supply actions for the data you specify. The Intent is being used to find actions, so don't assign it one; it should specify only the data to perform actions on. You should also specify the category of the action, either `CATEGORY_ALTERNATIVE` or `CATEGORY_SELECTED_ALTERNATIVE`.

The skeleton code for creating an Intent for menu-action resolution is shown here:

```
Intent intent = new Intent();
intent.setData(MyProvider.CONTENT_URI);
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

Pass this Intent into the Package Manager method `queryIntentActivityOptions`, specifying any options flags.

Listing 5.16 shows how to generate a list of actions to make available within your application.

Listing 5.16: Generating a list of possible actions to be performed on specific data

```
PackageManager packageManager = getPackageManager();

// Create the intent used to resolve which actions
// should appear in the menu.
```

```
Intent intent = new Intent();
intent.setData(MoonBaseProvider.CONTENT_URI);
intent.addCategory(Intent.CATEGORY_SELECTED_ALTERNATIVE);

// Specify flags. In this case, to return only
filters
// with the default category.
int flags = PackageManager.MATCH_DEFAULT_ONLY;

// Generate the list
List<ResolveInfo> actions;
actions =
packageManager.queryIntentActivities(intent, flags);

// Extract the list of action names
ArrayList<String> labels = new ArrayList<String>();
Resources r = getResources();
for (ResolveInfo action : actions )
  labels.add(r.getString(action.labelRes));
```

code snippet PA4AD_Ch05_Intents/src/MyActivity.java

Incorporating Anonymous Actions as Menu Items

The most common way to incorporate actions from third-party applications is to include them within your Menu Items or Action Bar Actions.

The `addIntentOptions` method, available from the `Menu` class, lets you specify an Intent that describes the data acted upon within your Activity, as described previously; however, rather than simply returning a list of possible Receivers, a new Menu Item will be created for each, with the text populated from the matching Intent Filters' labels.

To add Menu Items to your Menus dynamically at run time, use the `addIntentOptions` method on the `Menu` object in question: Pass in an Intent that specifies

the data for which you want to provide actions. Generally, this will be handled within your Activities' `onCreateOptionsMenu` or `onCreateContextMenu` handlers.

As in the previous section, the Intent you create will be used to resolve components with Intent Filters that supply actions for the data you specify. The Intent is being used to find actions, so don't assign it one; it should specify only the data to perform actions on. You should also specify the category of the action, either `CATEGORY_ALTERNATIVE` or `CATEGORY_SELECTED_ALTERNATIVE`.

The skeleton code for creating an Intent for menu-action resolution is shown here:

```
Intent intent = new Intent();
intent.setData(MyProvider.CONTENT_URI);
intent.addCategory(Intent.CATEGORY_ALTERNATIVE);
```

Pass this Intent in to `addIntentOptions` on the Menu you want to populate, as well as any options flags, the name of the calling class, the Menu group to use, and the Menu ID values. You can also specify an array of Intents you'd like to use to create additional Menu Items.

Listing 5.17 gives an idea of how to dynamically populate an Activity Menu.

Listing 5.17: Dynamic Menu population from advertised

actions

```
@Override
public boolean onCreateOptionsMenu(Menu
menu) {
  super.onCreateOptionsMenu(menu);

  // Create the intent used to resolve
which actions
  // should appear in the menu.
  Intent intent = new Intent();
  intent.setData(MoonBaseProvider.CONTENT_URI);
  intent.addCategory(Intent.CATEGORY_SELECTED_ALTERNATIVE);

  // Normal menu options to let you set a
group and ID
  // values for the menu items you're
adding.
  int menuGroup = 0;
  int menuItemId = 0;
  int menuItemOrder = Menu.NONE;

  // Provide the name of the component
that's calling
  // the action -- generally the current
Activity.
  ComponentName caller =
getComponentName();

  // Define intents that should be added
first.
  Intent[] specificIntents = null;
  // The menu items created from the
previous Intents
  // will populate this array.
  MenuItem[] outSpecificItems = null;

  // Set any optional flags.
  int flags = Menu.FLAG_APPEND_TO_GROUP;

  // Populate the menu
  menu.addIntentOptions(menuGroup,
                        menuItemId,
                        menuItemOrder,
                        caller,
                        specificIntents,
                        intent,
                        flags,

 outSpecificItems);

  return true;
}
```

code snippet
PA4AD_Ch05_Intents/src/MyActivity.java

Listening for Native Broadcast Intents

Many of the system Services broadcast Intents to signal changes. You can use these messages to add functionality to your own projects based on system events, such as time-zone changes, data-connection status, incoming SMS messages, or phone calls.

The following list introduces some of the native actions exposed as constants in the Intent class; these actions are used primarily to track device status changes:

- `ACTION_BOOT_COMPLETED`—Fired once when the device has completed its startup sequence. An application requires the `RECEIVE_BOOT_COMPLETED` permission to receive this broadcast.
- `ACTION_CAMERA_BUTTON`—Fired when the camera button is clicked.
- `ACTION_DATE_CHANGED` and `ACTION_TIME_CHANGED`—These actions are broadcast if the date or time on the device is manually changed (as opposed to changing through the inexorable progression of time).
- `ACTION_MEDIA_EJECT`—If the user chooses to eject the external storage media, this event is fired first. If your application is reading or writing to the external media storage, you should listen for this event to save and close any open file handles.

- `ACTION_MEDIA_MOUNTED` and `ACTION_MEDIA_UNMOUNTED`—These two events are broadcast whenever new external storage media are successfully added to or removed from the device, respectively.
- `ACTION_NEW_OUTGOING_CALL`—Broadcast when a new outgoing call is about to be placed. Listen for this broadcast to intercept outgoing calls. The number being dialed is stored in the `EXTRA_PHONE_NUMBER` extra, whereas the `resultData` in the returned Intent will be the number actually dialed. To register a Broadcast Receiver for this action, your application must declare the `PROCESS_OUTGOING_CALLS` uses-permission.
- `ACTION_SCREEN_OFF` and `ACTION_SCREEN_ON`—Broadcast when the screen turns off or on, respectively.
- `ACTION_TIMEZONE_CHANGED`—This action is broadcast whenever the phone's current time zone changes. The Intent includes a `time-zone` extra that returns the ID of the new `java.util.TimeZone`.

A comprehensive list of the broadcast actions used and transmitted natively by Android to notify applications of system state changes is available at http://developer.android.com/reference/android/content/Intent.html.

Android also uses Broadcast Intents to announce application-specific events, such as incoming SMS messages, changes in dock state, and battery level. The actions and Intents associated with these events will be discussed in more detail in later chapters when you learn more about the associated Services.

Monitoring Device State Changes Using Broadcast Intents

Monitoring the device state is an important part of creating efficient and dynamic applications whose behavior can change based on connectivity, battery charge state, and docking status.

Android broadcasts Intents for changes in each of these device states. The following sections examine how to create Intent Filters to register Broadcast Receivers that can react to such changes, and how to extract the device state information accordingly.

Listening for Battery Changes

To monitor changes in the battery level or charging status within an Activity, you can register a Receiver using an Intent Filter that listens for the `Intent.ACTION_BATTERY_CHANGED` broadcast by the Battery Manager.

The Broadcast Intent containing the current battery charge and charging status is a sticky Intent, so you can retrieve the current battery status at any time without needing to implement a Broadcast Receiver, as shown in Listing 5.18.

Listing 5.18: Determining battery and charge state information

```
IntentFilter batIntentFilter = new
IntentFilter(Intent.ACTION_BATTERY_CHANGED);
Intent battery = context.registerReceiver(null,
batIntentFilter);
int status = battery.getIntExtra(BatteryManager.EXTRA_STATUS,
-1);
boolean isCharging =
  status == BatteryManager.BATTERY_STATUS_CHARGING ||
  status == BatteryManager.BATTERY_STATUS_FULL;
```

code snippet PA4AD_Ch05_Intents/src/DeviceStateActivity.java

Note that you can't register the battery changed action within a manifest Receiver; however, you can monitor connection and disconnection from a power source and a low battery level using the following action strings, each prefixed with `android.intent.action`:

- `ACTION_BATTERY_LOW`
- `ACTION_BATTERY_OKAY`
- `ACTION_POWER_CONNECTED`
- `ACTION_POWER_DISCONNECTED`

Listening for Connectivity Changes

Changes in connectivity, including the bandwidth, latency, and availability of an Internet connection, can be significant signals for your application. In particular, you might choose to suspend recurring updates when you lose connectivity or to delay downloads of significant size until you have a Wi-Fi connection.

To monitor changes in connectivity, register a Broadcast Receiver (either within your application or within the manifest) to listen for the `android.net.conn.CONNECTIVITY_CHANGE` (`ConnectivityManager.CONNECTIVITY_ACTION`) action.

The connectivity change broadcast isn't sticky and doesn't contain any additional information regarding the change. To extract details on the current connectivity status, you need to use the Connectivity Manager, as shown in Listing 5.19.

Listing 5.19: Determining connectivity state information

```
String svcName = Context.CONNECTIVITY_SERVICE;
ConnectivityManager cm =
(ConnectivityManager)context.getSystemService(svcName);

NetworkInfo activeNetwork = cm.getActiveNetworkInfo();
boolean isConnected =
activeNetwork.isConnectedOrConnecting();
boolean isMobile = activeNetwork.getType() ==
                   ConnectivityManager.TYPE_MOBILE;
```

code snippet

The Connectivity Manager is examined in more detail in Chapter 16, "Bluetooth, NFC, Networks, and Wi-Fi."

Listening for Docking Changes

Android devices can be docked in either a car dock or desk dock. These, in term, can be either analog or digital docks. By registering a Receiver to listen for the `Intent.ACTION_DOCK_EVENT` (`android.intent.action.ACTION_DOCK_EVENT`), you can determine the docking status and type of dock.

Like the battery status, the dock event Broadcast Intent is sticky. Listing 5.20 shows how to extract the current docking status from the Intent returned when registering a Receiver for docking events.

Listing 5.20: Determining docking state

information

```
IntentFilter dockIntentFilter =
  new IntentFilter(Intent.ACTION_DOCK_EVENT);
Intent dock = registerReceiver(null,
dockIntentFilter);

int dockState =
dock.getIntExtra(Intent.EXTRA_DOCK_STATE,
                 Intent.EXTRA_DOCK_STATE_UNDOCKED);
boolean isDocked = dockState !=
Intent.EXTRA_DOCK_STATE_UNDOCKED;
```

code snippet
PA4AD_Ch05_Intents/src/DeviceStateActivity.java

Managing Manifest Receivers at Run Time

Using the Package Manager, you can enable and disable any of your application's manifest Receivers at run time using the `setComponentEnabledSetting` method. You can use this technique to enable or disable any application component (including Activities and Services), but it is particularly useful for manifest Receivers.

To minimize the footprint of your application, it's good practice to disable manifest Receivers that listen for common system events (such as connectivity changes) when your application doesn't need to respond to those events. This technique also enables you to schedule an action based on a system event—such as downloading a large file when the device is connected to Wi-Fi—without gaining the overhead of having the application launch every time a connectivity change is broadcast.

Listing 5.21 shows how to enable and disable a manifest Receiver at run time.

Listing 5.21: Dynamically toggling manifest Receivers

```
ComponentName myReceiverName = new ComponentName(this,
MyReceiver.class);
PackageManager pm = getPackageManager();

// Enable a manifest receiver
pm.setComponentEnabledSetting(myReceiverName,
```

```
  PackageManager.COMPONENT_ENABLED_STATE_ENABLED,
  PackageManager.DONT_KILL_APP);

// Disable a manifest receiver
pm.setComponentEnabledSetting(myReceiverName,
  PackageManager.COMPONENT_ENABLED_STATE_DISABLED,
  PackageManager.DONT_KILL_APP);
```

code snippet PA4AD_Ch05_Intents/src/DeviceStateActivity.java

Chapter 6

Using Internet Resources

What's in this Chapter?

- Connecting to Internet resources
- Parsing XML resources
- Using the Download Manager to download files
- Querying the Download Manager
- Using the Account Manager to authenticate with Google App Engine

This chapter introduces Android's Internet connectivity model and some of the Java techniques for parsing Internet data feeds. You'll learn how to connect to an Internet resource and how to use the SAX Parser and the XML Pull Parser to parse XML resources.

An earthquake-monitoring example will demonstrate how to tie together all these features, and forms the basis of an ongoing example that you'll improve and extend in later chapters.

This chapter introduces the Download Manager, and you

learn how to use it to schedule and manage long-running downloads. You'll also learn how to customize its notifications and query the Downloads Content Provider to determine the status of your downloads.

Finally, this chapter introduces how to use the Account Manager to make authenticated requests from Google App Engine backends.

Downloading and Parsing Internet Resources

Android offers several ways to leverage Internet resources. At one extreme you can use a WebView to include a WebKit-based browser within an Activity. At the other extreme you can use client-side APIs, such as the Google APIs, to interact directly with server processes. Somewhere in between, you can process remote XML feeds to extract and process data using a Java-based XML parser, such as SAX or the XML Pull Parser.

With Internet connectivity and a WebKit browser, you might ask if there's any reason to create native Internet-based applications when you could make a web-based version instead.

There are a number of benefits to creating thick- and thin-client native applications rather than relying on entirely web-based solutions:

- **Bandwidth**—Static resources such as images, layouts, and sounds can be expensive on devices with bandwidth restraints. By creating a native application, you can limit the bandwidth requirements to changed data only.

- **Caching**—With a browser-based solution, a patchy Internet connection can result in intermittent application availability. A native application can cache data and user actions to provide as much functionality as possible without a live connection and synchronize with the cloud when a connection is reestablished.
- **Reducing battery drain**—Each time your application opens a connection to a server, the wireless radio will be turned on (or kept on). A native application can bundle its connections, minimizing the number of connections initiated. The longer the period between network requests, the longer the wireless radio can be left off.
- **Native features**—Android devices are more than simple platforms for running a browser. They include location-based services, Notifications, widgets, camera hardware, background Services, and hardware sensors. By creating a native application, you can combine the data available online with the hardware features available on the device to provide a richer user experience.

Modern mobile devices offer a number of alternatives for accessing the Internet. Broadly speaking, Android provides two connection techniques for Internet connectivity. Each is offered transparently to the application layer.

- **Mobile Internet**—GPRS, EDGE, 3G, 4G, and LTE Internet access is available through carriers that offer mobile data.
- **Wi-Fi**—Wi-Fi receivers and mobile hotspots are becoming increasingly common.

If you use Internet resources in your application, remember that your users' data connections are dependent on the communications technology available to them. EDGE and GSM connections are notoriously low-bandwidth, whereas a Wi-Fi connection may be unreliable in a mobile setting.

Optimize the user experience by limiting the quantity of data transmitted and ensure that your application is robust enough to handle network outages and bandwidth limitations.

Connecting to an Internet Resource

Before you can access Internet resources, you need to add a n `INTERNET` uses-permission node to your application manifest, as shown in the following XML snippet:

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

Listing 6.1 shows the basic pattern for opening an Internet data stream.

Listing 6.1: Opening an Internet data stream

```
String myFeed = getString(R.string.my_feed);
try {
  URL url = new URL(myFeed);

  // Create a new HTTP URL connection
  URLConnection connection = url.openConnection();
  HttpURLConnection httpConnection =
(HttpURLConnection)connection;

  int responseCode = httpConnection.getResponseCode();
  if (responseCode == HttpURLConnection.HTTP_OK) {
    InputStream in = httpConnection.getInputStream();
    processStream(in);
  }
}
catch (MalformedURLException e) {
  Log.d(TAG, "Malformed URL Exception.");
}
catch (IOException e) {
  Log.d(TAG, "IO Exception.");
```

```
}
```

code snippet PA4AD_Ch06_Internet/src/MyActivity.java

Attempting to perform network operations on the main UI thread will cause a `NetworkOnMainThreadException` on the latest Android platform releases. Be sure to execute code, such as that shown in Listing 6.1, in a background thread, as described in Chapter 9, "Working in the Background."

Android includes several classes to help you handle network communications. They are available in the `java.net.*` and `android.net.*` packages.

Later in this chapter is a working example that shows how to obtain and process an Internet feed to get a list of earthquakes felt in the last 24 hours. Chapter 16, "Bluetooth, NFC, Networks, and Wi-Fi," features more information on managing specific Internet connections, including monitoring connection status and configuring Wi-Fi access point connections.

Parsing XML Using the XML Pull Parser

Although detailed instructions for parsing XML and interacting with specific web services are outside the scope of this book, it's important to understand the available technologies.

This section provides a brief overview of the XML Pull Parser, whereas the next section demonstrates the use of the DOM parser to retrieve earthquake details from the United States Geological Survey (USGS).

The XML Pull Parser API is available from the following libraries:

```
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserException;
import org.xmlpull.v1.XmlPullParserFactory;
```

It enables you to parse an XML document in a single pass. Unlike the DOM parser, the Pull Parser presents the elements of your document in a sequential series of events and tags.

Your location within the document is represented by the current event. You can determine the current event by calling `getEventType`. Each document begins at the `START_DOCUMENT` event and ends at `END_DOCUMENT`.

To proceed through the tags, simply call `next`, which causes you to progress through a series of matched (and often nested) `START_TAG` and `END_TAG` events. You

can extract the name of each tag by calling `getName` and extract the text between each set of tags using `getNextText`.

Listing 6.2 demonstrates how to use the XML Pull Parser to extract details from the points of interest list returned by the Google Places API.

Listing 6.2: Parsing XML using the XML Pull Parser

```
private void processStream(InputStream inputStream) {
  // Create a new XML Pull Parser.
  XmlPullParserFactory factory;
  try {
    factory = XmlPullParserFactory.newInstance();
    factory.setNamespaceAware(true);
    XmlPullParser xpp = factory.newPullParser();

    // Assign a new input stream.
    xpp.setInput(inputStream, null);
    int eventType = xpp.getEventType();

    // Continue until the end of the document is reached.
    while (eventType != XmlPullParser.END_DOCUMENT) {
      // Check for a start tag of the results tag.
      if (eventType == XmlPullParser.START_TAG &&
          xpp.getName().equals("result")) {
        eventType = xpp.next();
        String name = "";
        // Process each result within the result tag.
        while (!(eventType == XmlPullParser.END_TAG &&
               xpp.getName().equals("result"))) {
          // Check for the name tag within the results
tag.
          if (eventType == XmlPullParser.START_TAG &&
```

```
                xpp.getName().equals("name"))
              // Extract the POI name.
              name = xpp.nextText();
            // Move on to the next tag.
            eventType = xpp.next();
          }
          // Do something with each POI name.
        }
        // Move on to the next result tag.
        eventType = xpp.next();
      }
    } catch (XmlPullParserException e) {
      Log.d("PULLPARSER", "XML Pull Parser Exception", e);
    } catch (IOException e) {
      Log.d("PULLPARSER", "IO Exception", e);
    }
  }
```

code snippet PA4AD_Ch6_Internet/src/MyActivity.java

Creating an Earthquake Viewer

In the following example you'll create a tool that uses a USGS earthquake feed to display a list of recent earthquakes. You will return to this earthquake application several times in the following chapters, gradually adding more features and functionality.

The earthquake feed XML is parsed here by the DOM parser. Several alternatives exist, including the XML Pull Parser described in the previous section. As noted, a detailed analysis of the alternative XML parsing techniques is beyond the scope of this book.

In this example you'll create a list-based Activity that connects to an earthquake feed and displays the location, magnitude, and time of the earthquakes it contains.

To simplify readability, each of these examples excludes the import statements. If you are using Eclipse, you can press Ctrl+Shift+o (or Cmd+Shift+o on Mac) to automatically populate the import statements required to support the classes used in your code.

1. Start by creating an Earthquake project featuring an `Earthquake` Activity.

2. Create a new `EarthquakeListFragment` that extends `ListFragment`. This Fragment displays your list of earthquakes.

```
public class EarthquakeListFragment extends ListFragment
```

```
{
}
```

3. Modify the `main.xml` layout resource to include the Fragment you created in Step 2. Be sure to name it so that you can reference it from the Activity code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">
  <fragment
android:name="com.paad.earthquake.EarthquakeListFragment"
    android:id="@+id/EarthquakeListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
  />
</LinearLayout>
```

4. Create a new public `Quake` class. This class will be used to store the details (date, details, location, magnitude, and link) of each earthquake. Override the `toString` method to provide the string that will be used to represent each quake in the List View.

```
package com.paad.earthquake;

import java.util.Date;
import java.text.SimpleDateFormat;
import android.location.Location;

public class Quake {
  private Date date;
  private String details;
  private Location location;
  private double magnitude;
  private String link;

  public Date getDate() { return date; }
  public String getDetails() { return details; }
  public Location getLocation() { return location; }
```

```
  public double getMagnitude() { return magnitude; }
  public String getLink() { return link; }

  public Quake(Date _d, String _det, Location _loc,
double _mag, String _link) {
    date = _d;
    details = _det;
    location = _loc;
    magnitude = _mag;
    link = _link;
  }

  @Override
  public String toString() {
    SimpleDateFormat sdf = new SimpleDateFormat("HH.mm");
    String dateString = sdf.format(date);
    return dateString + ": " + magnitude + " " + details;
  }

}
```

5. In the `EarthquakeListFragment`, override the `onActivityCreated` method to store an `ArrayList` of `Quake` objects, and bind that to the underlying `ListView` using an `ArrayAdapter`:

```
public class EarthquakeListFragment extends ListFragment
{

  ArrayAdapter<Quake> aa;
  ArrayList<Quake> earthquakes = new ArrayList<Quake>();

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

    int layoutID = android.R.layout.simple_list_item_1;
    aa = new ArrayAdapter<Quake>(getActivity(), layoutID
, earthquakes);
    setListAdapter(aa);
  }
}
```

6. Start processing the earthquake feed. For this example, the feed used is the one-day USGS feed for earthquakes with a magnitude greater than 2.5. Add the location of your feed as an external string resource. This lets you potentially specify a different feed based on a user's location.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Earthquake</string>
  <string name="quake_feed">
    http://earthquake.usgs.gov/eqcenter/catalogs/1day-
M2.5.xml
  </string>
</resources>
```

7. Before your application can access the Internet, it needs to be granted permission for Internet access. Add the Internet `uses-permission` to the manifest:

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

8. Returning to the Earthquake List Fragment, create a new `refreshEarthquakes` method that connects to and parses the earthquake feed. Extract each earthquake and parse the details to obtain the date, magnitude, link, and location. As you finish parsing each earthquake, pass it in to a new `addNewQuake` method. Note that the `addNewQuake` method is executed within a `Runnable` posted from a `Handler` object. This allows you to execute the `refreshEarthquakes` method on a background thread before updating the UI within `addNewQuake`. This will be explored in more detail in Chapter 9.

```
private static final String TAG = "EARTHQUAKE";
```

```
private Handler handler = new Handler();

public void refreshEarthquakes() {
  // Get the XML
  URL url;
  try {
    String quakeFeed = getString(R.string.quake_feed);
    url = new URL(quakeFeed);

    URLConnection connection;
    connection = url.openConnection();

    HttpURLConnection httpConnection =
(HttpURLConnection)connection;
    int responseCode = httpConnection.getResponseCode();

    if (responseCode == HttpURLConnection.HTTP_OK) {
      InputStream in = httpConnection.getInputStream();

      DocumentBuilderFactory dbf =
DocumentBuilderFactory.newInstance();
      DocumentBuilder db = dbf.newDocumentBuilder();

      // Parse the earthquake feed.
      Document dom = db.parse(in);
      Element docEle = dom.getDocumentElement();

      // Clear the old earthquakes
      earthquakes.clear();

      // Get a list of each earthquake entry.
      NodeList nl = docEle.getElementsByTagName("entry");
      if (nl != null && nl.getLength() > 0) {
        for (int i = 0 ; i < nl.getLength(); i++) {
          Element entry = (Element)nl.item(i);
          Element title =
(Element)entry.getElementsByTagName("title").item(0);
          Element g =
(Element)entry.getElementsByTagName("georss:point").item(0);
          Element when =
(Element)entry.getElementsByTagName("updated").item(0);
          Element link =
(Element)entry.getElementsByTagName("link").item(0);

          String details =
title.getFirstChild().getNodeValue();
```

```
          String hostname = "http://earthquake.usgs.gov";
          String linkString = hostname +
link.getAttribute("href");

          String point =
g.getFirstChild().getNodeValue();
          String dt =
when.getFirstChild().getNodeValue();
          SimpleDateFormat sdf = new
SimpleDateFormat("yyyy-MM-dd'T'hh:mm:ss'Z'");
          Date qdate = new
GregorianCalendar(0,0,0).getTime();
          try {
            qdate = sdf.parse(dt);
          } catch (ParseException e) {
            Log.d(TAG, "Date parsing exception.", e);
          }

          String[] location = point.split(" ");
          Location l = new Location("dummyGPS");
          l.setLatitude(Double.parseDouble(location[0]));

 l.setLongitude(Double.parseDouble(location[1]));

          String magnitudeString = details.split(" ")[1];
          int end =  magnitudeString.length()-1;
          double magnitude =
Double.parseDouble(magnitudeString.substring(0, end));

          details = details.split(",")[1].trim();

          final Quake quake = new Quake(qdate, details,
l, magnitude, linkString);

          // Process a newly found earthquake
          handler.post(new Runnable() {
            public void run() {
              addNewQuake(quake);
            }
          });
        }
      }
    }
  } catch (MalformedURLException e) {
    Log.d(TAG, "MalformedURLException");
  } catch (IOException e) {
```

```
      Log.d(TAG, "IOException");
    } catch (ParserConfigurationException e) {
      Log.d(TAG, "Parser Configuration Exception");
    } catch (SAXException e) {
      Log.d(TAG, "SAX Exception");
    }
    finally {
    }
  }

  private void addNewQuake(Quake _quake) {
    // TODO Add the earthquakes to the array list.
  }
```

9. Update the `addNewQuake` method so that it takes each newly processed quake and adds it to the earthquake Array List. It should also notify the Array Adapter that the underlying data has changed.

```
  private void addNewQuake(Quake _quake) {
    // Add the new quake to our list of earthquakes.
    earthquakes.add(_quake);

    // Notify the array adapter of a change.
    aa.notifyDataSetChanged();
  }
```

10. Modify your `onActivityCreated` method to call `refreshEarthquakes` on startup. Network operations should always be performed in a background thread—a requirement that is enforced in API level 11 onwards.

```
  @Override
  public void onActivityCreated(Bundle savedInstanceState)
  {
    super.onActivityCreated(savedInstanceState);

    int layoutID = android.R.layout.simple_list_item_1;
    aa = new ArrayAdapter<Quake>(getActivity(), layoutID ,
  earthquakes);
    setListAdapter(aa);

    Thread t = new Thread(new Runnable() {
      public void run() {
```

```
            refreshEarthquakes();
          }
        });
        t.start();
    }
```

If your application is targeting API level 11 or above, attempting to perform network operations on the main UI thread will cause a `NetworkOnMainThreadException`. In this example a simple Thread is used to post the `refreshEarthquakes` method on a background thread.

11. When you run your project, you should see a List View that features the earthquakes from the last 24 hours with a magnitude greater than 2.5 (Figure 6.1).

Figure 6.1


21:07
Earthquake
04.30: 5.2 Samar
02.43: 4.7 Region Metropolitana
02.36: 3.6 Virgin Islands region
00.38: 2.6 Puerto Rico
21.26: 2.8 Kodiak Island region
19.58: 5.1 Vanuatu
19.35: 4.8 near the east coast of Honshu

All code snippets in this example are part of the *Chapter* 6 *Earthquake* project, available for download at www.wrox.com.

Using the Download Manager

The Download Manager was introduced in Android 2.3 (API level 9) as a Service to optimize the handling of long-running downloads. The Download Manager handles the HTTP connection and monitors connectivity changes and system reboots to ensure each download completes successfully.

It's good practice to use the Download Manager in most situations, particularly where a download is likely to continue in the background between user sessions, or when successful completion is important.

To access the Download Manager, request the `DOWNLOAD_SERVICE` using the `getSystemService` method, as follows:

```
String serviceString = Context.DOWNLOAD_SERVICE;
DownloadManager downloadManager;
downloadManager =
(DownloadManager)getSystemService(serviceString);
```

Downloading Files

To request a download, create a new `DownloadManager.Request`, specifying the URI of the file to download and passing it in to the Download Manager's `enqueue` method, as shown in Listing 6.3.

Listing 6.3: Listing 6 3: Downloading files using the Download Manager

```
String serviceString = Context.DOWNLOAD_SERVICE;
DownloadManager downloadManager;
downloadManager =
(DownloadManager)getSystemService(serviceString);

Uri uri =
Uri.parse("http://developer.android.com/shareables/icon_templates-
v4.0.zip");
DownloadManager.Request request = new Request(uri);
long reference = downloadManager.enqueue(request);
```

code snippet PA4AD_Ch6_DownloadManager/src/MyActivity.java

You can use the returned reference value to perform future actions or queries on the download, including checking its status or canceling it.

You can add an HTTP header to your request, or override the mime type returned by the server, by calling `addRequestHeader` and `setMimeType`, respectively, on your `Request` object.

You can also specify the connectivity conditions under which to execute the download. The

`setAllowedNetworkTypes` method enables you to restrict downloads to either Wi-Fi or mobile networks, whereas the `setAllowedOverRoaming` method predictably enables you to prevent downloads while the phone is roaming.

The following snippet shows how to ensure a large file is downloaded only when connected to Wi-Fi:

```
request.setAllowedNetworkTypes(Request.NETWORK_WIFI);
```

Android API level 11 introduced the `getRecommendedMaxBytesOverMobile` convenience method, which is useful to determine if you should restrict a download to Wi-Fi by returning a recommended maximum number of bytes to transfer over a mobile data connection.

After calling `enqueue`, the download begins as soon as connectivity is available and the Download Manager is free.

To receive a notification when the download is completed, register a Receiver to receive an `ACTION_DOWNLOAD_COMPLETE` broadcast. It will include an `EXTRA_DOWNLOAD_ID` extra that contains the reference ID of the download that has completed, as shown in Listing 6.4.

Listing 6.4: Monitoring downloads for completion

```
IntentFilter filter = new
IntentFilter(DownloadManager.ACTION_DOWNLOAD_COMPLETE);
```

```
BroadcastReceiver receiver = new BroadcastReceiver()
{
  @Override
  public void onReceive(Context context, Intent
intent) {
    long reference =
intent.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID,
-1);
    if (myDownloadReference == reference) {
      // Do something with downloaded file.
    }
  }
};

registerReceiver(receiver, filter);
```

code snippet PA4AD_Ch6_DownloadManager/src/MyActivity.java

You can use Download Manager's `openDownloadedFile` method to receive a Parcel File Descriptor to your file, to query the Download Manager to obtain its location, or to manipulate it directly if you've specified a filename and location yourself.

It's also good practice to register a Receiver for the `ACTION_NOTIFICATION_CLICKED` action, as shown in Listing 6.5. This Intent will be broadcast whenever a user selects a download from the Notification tray or the Downloads app.

Listing 6.5: Responding to download notification clicks

```
IntentFilter filter = new
IntentFilter(DownloadManager.ACTION_NOTIFICATION_CLICKED);

BroadcastReceiver receiver = new
BroadcastReceiver() {
  @Override
  public void onReceive(Context context,
Intent intent) {
```

```
    String extraID =
DownloadManager.EXTRA_NOTIFICATION_CLICK_DOWNLOAD_IDS;
    long[] references =
intent.getLongArrayExtra(extraID);
    for (long reference : references)
      if (reference == myDownloadReference) {
        // Do something with downloading file.
      }
  }
};

registerReceiver(receiver, filter);
```

code snippet PA4AD_Ch6_DownloadManager/src/MyActivity.java

Customizing Download Manager Notifications

By default, ongoing Notifications will be displayed for each download managed by the Download Manager. Each Notification will show the current download progress and the filename (Figure 6.2).

Figure 6.2

The Download Manager enables you to customize the Notification displayed for each download request, including hiding it completely. The following snippet shows how to use the `setTitle` and `setDescription` methods to customize the text displayed in the file download Notification. Figure 6.3 shows the result.

```
request.setTitle("Earthquakes");
request.setDescription("Earthquake XML");
```

Figure 6.3

The `setNotificationVisibility` method lets you control when, and if, a Notification should be displayed for your request using one of the following flags:

- `Request.VISIBILITY_VISIBLE`—An ongoing Notification will be visible for the duration that the download is in progress. It will be removed when the download is complete. This is the default option.
- `Request.VISIBILITY_VISIBLE_NOTIFY_COMPLETED`—An ongoing Notification will be displayed during the download and will continue to be displayed (until selected or dismissed) once the download has completed.
- `Request.VISIBILITY_VISIBLE_NOTIFY_ONLY_COMPLETION`—The notification will be displayed only after the download is complete.
- `Request.VISIBILITY_HIDDEN`—No Notification will be displayed for this download. In order to set this flag, your application must have the `DOWNLOAD_WITHOUT_NOTIFICATION` uses-permission specified in its manifest.

You will learn more about creating your own custom Notifications in Chapter 9.

Specifying a Download Location

By default, all Download Manager downloads are saved to the shared download cache using system-generated filenames. Each Request object can specify a download location, though all downloads must be stored somewhere on external storage and the calling application must have the `WRITE_EXTERNAL_STORAGE` uses-permission in its manifest:

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

The following code snippet shows how to specify an arbitrary path on external storage:

```
request.setDestinationUri(Uri.fromFile(f));
```

If the downloaded file is to your application, you may want to place it in your application's external storage folder. Note that access control is not applied to this folder, and other applications will be able to access it. If your application is uninstalled, files stored in these folders will also be removed.

The following snippet specifies storing a file in your application's external downloads folder:

```
request.setDestinationInExternalFilesDir(this,
  Environment.DIRECTORY_DOWNLOADS, "Bugdroid.png");
```

For files that can or should be shared with other applications— particularly those you want to scan with the Media Scanner—you can specify a folder within the public folder on the external storage. The following snippet requests a file be stored in the public music folder:

```
request.setDestinationInExternalPublicDir(Environment.DIRECTORY_MUSIC,
  "Android_Rock.mp3");
```

> See Chapter 7, “Files, Saving State, and Preferences,” for more details about external storage and the Environment static variables you can use to specify folders within it.

It's important to note that by default files downloaded by the Download Manager are not scanned by Media Scanner, so they might not appear in apps such as Gallery and Music Player.

To make downloaded files scannable, call `allowScaningByMediaScanner` on the Request object.

If you want your files to be visible and manageable by the system's Downloads app, you need to call `setVisibleInDownloadsUi`, passing in `true`.

Cancelling and Removing Downloads

The Download Manager's `remove` method lets you cancel a pending download, abort a download in progress, or delete a completed download.

As shown in the following code snippet, the `remove` method accepts download IDs as optional arguments, enabling you to specify one or many downloads to cancel:

```
downloadManager.remove(REFERENCE_1, REFERENCE_2, REFERENCE_3);
```

It returns the number of downloads successfully canceled. If a download is canceled, all associated files—both partial and complete—are removed.

Querying the Download Manager

You can query the Download Manager to find the status, progress, and details of your download requests by using the `query` method that returns a Cursor of downloads.

Cursors are a data construct used by Android to return data stored in a Content Provider or SQLite database. You will learn more about Content Providers, Cursors, and how to find data stored in them in Chapter 8, "Databases and Content Providers."

The `query` method takes a `DownloadManager.Query` object as a parameter. Use the `setFilterById` method on a Query object to specify a sequence of download reference IDs, or use the `setFilterByStatus` method to filter on a download status using one of the `DownloadManager.STATUS_*` constants to specify running, paused, failed, or successful downloads.

The Download Manager includes a number of `COLUMN_*` static String constants that you can use to query the result Cursor. You can find details for each download, including the status, files size, bytes downloaded so far, title, description, URI, local filename and URI, media type, and Media Provider download URI.

Listing 6.6 expands on Listing 6.4 to demonstrate how to find the local filename and URI of a completed downloads from within a Broadcast Receiver registered to listen for download completions.

Listing 6.6: Finding details of completed downloads

```
@Override
public void onReceive(Context context, Intent intent) {
  long reference =
intent.getLongExtra(DownloadManager.EXTRA_DOWNLOAD_ID, -1);

  if (reference == myDownloadReference) {
    Query myDownloadQuery = new Query();
    myDownloadQuery.setFilterById(reference);

    Cursor myDownload =
downloadManager.query(myDownloadQuery);
    if (myDownload.moveToFirst()) {
      int fileNameIdx =

 myDownload.getColumnIndex(DownloadManager.COLUMN_LOCAL_FILENAME);
      int fileUriIdx =

 myDownload.getColumnIndex(DownloadManager.COLUMN_LOCAL_URI);

      String fileName = myDownload.getString(fileNameIdx);
      String fileUri = myDownload.getString(fileUriIdx);

      // TODO Do something with the file.
    }
    myDownload.close();
  }
}
```

code snippet PA4AD_Ch6_DownloadManager/src/MyActivity.java

For downloads that are either paused or have failed, you can query the `COLUMN_REASON` column to find the cause represented as an integer.

In the case of `STATUS_PAUSED` downloads, you can interpret the reason code by using one of the `DownloadManager.PAUSED_*` static constants to determine if the download has been paused while waiting for network connectivity, a Wi-Fi connection, or pending a retry.

For `STATUS_FAILED` downloads, you can determine the cause of failure using the `DownloadManager.ERROR_*` codes. Possible error codes include lack of a storage device, insufficient free space, duplicate filenames, or HTTP errors.

Listing 6.7 shows how to find a list of the currently paused downloads, extracting the reason the download was paused, the filename, its title, and the current progress.

Listing 6.7: Finding details of paused downloads

```
// Obtain the Download Manager Service.
String serviceString = Context.DOWNLOAD_SERVICE;
DownloadManager downloadManager;
downloadManager =
(DownloadManager)getSystemService(serviceString);

// Create a query for paused downloads.
Query pausedDownloadQuery = new Query();
pausedDownloadQuery.setFilterByStatus(DownloadManager.STATUS_PAUSED);

// Query the Download Manager for paused downloads.
Cursor pausedDownloads =
downloadManager.query(pausedDownloadQuery);

// Find the column indexes for the data we require.
int reasonIdx =
pausedDownloads.getColumnIndex(DownloadManager.COLUMN_REASON);
int titleIdx =
pausedDownloads.getColumnIndex(DownloadManager.COLUMN_TITLE);
int fileSizeIdx =
  pausedDownloads.getColumnIndex(DownloadManager.COLUMN_TOTAL_SIZE_BYTES);

int bytesDLIdx =
  pausedDownloads.getColumnIndex(DownloadManager.COLUMN_BYTES_DOWNLOADED_SO_FAR);

// Iterate over the result Cursor.
while (pausedDownloads.moveToNext()) {
  // Extract the data we require from the Cursor.
  String title = pausedDownloads.getString(titleIdx);
  int fileSize = pausedDownloads.getInt(fileSizeIdx);
  int bytesDL = pausedDownloads.getInt(bytesDLIdx);

  // Translate the pause reason to friendly text.
  int reason = pausedDownloads.getInt(reasonIdx);
  String reasonString = "Unknown";
  switch (reason) {
    case DownloadManager.PAUSED_QUEUED_FOR_WIFI :
      reasonString = "Waiting for WiFi"; break;
    case DownloadManager.PAUSED_WAITING_FOR_NETWORK :
      reasonString = "Waiting for connectivity"; break;
    case DownloadManager.PAUSED_WAITING_TO_RETRY :
      reasonString = "Waiting to retry"; break;
    default : break;
  }

  // Construct a status summary
  StringBuilder sb = new StringBuilder();
  sb.append(title).append("\n");
  sb.append(reasonString).append("\n");
```

```
  sb.append("Downloaded ").append(bytesDL).append(" / "
).append(fileSize);

  // Display the status
  Log.d("DOWNLOAD", sb.toString());
}

// Close the result Cursor.
pausedDownloads.close();
```

code snippet PA4AD_Ch6_DownloadManager/src/MyActivity.java

Using Internet Services

Software as a service (SaaS) and *cloud computing* are becoming increasingly popular as companies try to reduce the cost overheads associated with installing, upgrading, and maintaining deployed software. The result is a range of rich Internet services with which you can build thin mobile applications that enrich online services with the personalization available from your smartphone or tablet.

The idea of using a middle tier to reduce client-side load is not a novel one, and happily there are many Internet-based options to supply your applications with the level of service you need.

The sheer volume of Internet services available makes it impossible to list them all here (let alone look at them in any detail), but the following list shows some of the more mature and interesting Internet services currently available.

- **Google Services APIs**—In addition to the native Google applications, Google offers web APIs for access to their Calendar, Docs, Blogger, and Picasa Web Albums platforms. These APIs collectively make use of a form of XML for data communication.
- **Yahoo! Pipes**—Yahoo! Pipes offers a graphical web-based approach to XML feed manipulation.

Using pipes, you can filter, aggregate, analyze, and otherwise manipulate XML feeds and output them in a variety of formats to be consumed by your applications.

- **Google App Engine**—Using the Google App Engine, you can create cloud-hosted web services that shift complex processing away from your mobile client. Doing so reduces the load on your system resources but comes at the price of Internet-connection dependency. Google also offers Cloud Storage and Prediction API services.
- **Amazon Web Services**—Amazon offers a range of cloud-based services, including a rich API for accessing its media database of books, CDs, and DVDs. Amazon also offers a distributed storage solution (S3) and Elastic Compute Cloud (EC2).

Connecting to Google App Engine

To use the Google Play Store, users must be signed in to a Google account on their phones; therefore, if your application connects to a Google App Engine backend to store and retrieve data related to a particular user, you can use the Account Manager to handle the authentication.

The Account Manager enables you to ask users for permission to retrieve an authentication token, which, in turn, can be used to obtain a cookie from your server that can then be used to make future authenticated requests.

To retrieve accounts and authentication tokens from the Account Manager, your application requires the `GET_ACCOUNTS` uses-permission:

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

Making authenticated Google App Engine requests is a three-part process:

1. Request an auth token.
2. Use the auth token to request an auth cookie.
3. Use the auth cookie to make authenticated requests.

Listing 6.8 demonstrates how to request an auth token for Google accounts using the Account Manager.

Listing 6.8: Requesting an auth token

```
String acctSvc = Context.ACCOUNT_SERVICE;
AccountManager accountManager =
(AccountManager)getSystemService(acctSvc);

Account[] accounts =
accountManager.getAccountsByType("com.google");
```

```
if (accounts.length > 0)
  accountManager.getAuthToken(accounts[0], "ah", false,
                              myAccountManagerCallback,
null);
```

code snippet PA4AD_Ch6_AppEngine/src/MyActivity.java

The Account Manager then checks to see if the user has approved your request for an auth token. The result is returned to your application via the Account Manager Callback you specified when making the request.

In the following extension to Listing 6.8, the returned bundle is inspected for an Intent stored against the `AccountManager.KEY_INTENT` key. If this key's value is null, the user has approved your application's request, and you can retrieve the auth token from the bundle.

```
private static int ASK_PERMISSION = 1;

private class GetAuthTokenCB implements
AccountManagerCallback<Bundle> {
  public void run(AccountManagerFuture<Bundle>
result) {
    try {
      Bundle bundle = result.getResult();
      Intent launch =
(Intent)bundle.get(AccountManager.KEY_INTENT);
      if (launch != null)
        startActivityForResult(launch,
ASK_PERMISSION);
      else {
        // Extract the auth token and request an
auth cookie.
      }
    }
    catch (Exception ex) {}
  }
};
```

If the key's value is not null, you must start a new Activity using the bundled Intent to request the user's permission. The user will be prompted to approve or deny your request. After control has been passed back to your application, you should request the auth token again.

The auth token is stored within the Bundle parameter against the `AccountManager.KEY_AUTHTOKEN`, as follows:

```
String auth_token =
bundle.getString(AccountManager.KEY_AUTHTOKEN);
```

You can use this token to request an auth cookie from Google App Engine by configuring an `httpClient` and using it to transmit an `HttpGet` request, as follows:

```
DefaultHttpClient http_client = new
DefaultHttpClient();
http_client.getParams().setBooleanParameter(ClientPNames.HANDLE_REDIRECTS,
false);

String getString =
"https://[yourappsubdomain].appspot.com/_ah/login?"
+

"continue=http://localhost/&auth=" +
                    auth_token;
HttpGet get = new HttpGet(getString);

HttpResponse response =
http_client.execute(get);
```

If the request was successful, simply iterate over the Cookies stored in the HTTP Client's Cookie Store to confirm the auth cookie has been set. The HTTP Client used to make the request has the authenticated cookie, and all future requests to Google App Engine using it will be properly authenticated.

```
if
(response.getStatusLine().getStatusCode()
!= 302)
  return false;
else {
  for (Cookie cookie :
http_client.getCookieStore().getCookies())
    if
(cookie.getName().equals("ACSID")) {
      // Make authenticated requests
to your Google App Engine server.
    }
}
```

Best Practices for Downloading Data Without Draining the Battery

The timing and techniques you use to download data can have a significant effect on battery life. The wireless radio on mobile devices draws significant power when active, so it's important to consider how your application's connectivity model may impact the operation of the underlying radio hardware.

Every time you create a new connection to download additional data, you risk waking the wireless radio from standby mode to active mode. In general, it's good practice to bundle your connections and associated downloads to perform them concurrently and infrequently.

To use a converse example, creating frequent, short-lived connections that download small amounts of data can have the most dramatic impact on the battery.

You can use the following techniques to minimize your application's battery cost.

- **Aggressively prefetch**—The more data you

download in a single connection, the less frequently the radio will need to be powered up to download more data. This will need to be balanced with downloading too much data that won't be used.

- **Bundle your connections and downloads**—Rather than sending time-insensitive data such as analytics as they're received, bundle them together and schedule them to transmit concurrently with other connections, such as when refreshing content or prefetching data. Remember, each new connection has the potential of powering up the radio.
- **Reuse existing connections rather than creating new ones**—Using existing connections rather than initiating new ones for each transfer can dramatically improve network performance, reduce latency, and allow the network to intelligently react to congestion and related issues
- **Schedule repeated downloads as infrequently as possible**—It's good practice to set the default refresh frequency to as low as usability will allow, rather than as fast as possible. For users who require their updates to be more frequent, provide preferences that allow them to sacrifice battery life in exchange for freshness.

Chapter 7

Files, Saving State,and Preferences

What's in this Chapter?

- Persisting simple application data using Shared Preferences
- Saving Activity instance data between sessions
- Managing application preferences and building Preference Screens
- Saving and loading files and managing the local filesystem
- Including static files as external resources

This chapter introduces some of the simplest and most versatile data-persistence techniques in Android: Shared Preferences, instance-state Bundles, and local files.

Saving and loading data is essential for most applications. At a minimum, an Activity should save its user interface (UI) state before it becomes inactive to ensure the same UI is presented when it restarts. It's also likely that

you'll need to save user preferences and UI selections.

Android's nondeterministic Activity and application lifetimes make persisting UI state and application data between sessions particularly important, as your application process may have been killed and restarted before it returns to the foreground. Android offers several alternatives for saving application data, each optimized to fulfill a particular need.

Shared Preferences are a simple, lightweight name/value pair (NVP) mechanism for saving primitive application data, most commonly a user's application preferences. Android also offers a mechanism for recording application state within the Activity lifecycle handlers, as well as for providing access to the local filesystem, through both specialized methods and the `java.io` classes.

Android also offers a rich framework for user preferences, allowing you to create settings screens consistent with the system settings.

Saving Simple Application Data

The data-persistence techniques in Android provide options for balancing speed, efficiency, and robustness.

- **Shared Preferences**—When storing UI state, user preferences, or application settings, you want a lightweight mechanism to store a known set of values. Shared Preferences let you save groups of name/value pairs of primitive data as named preferences.
- **Saved application UI state**—Activities and Fragments include specialized event handlers to record the current UI state when your application is moved to the background.
- **Files**—It's not pretty, but sometimes writing to and reading from files is the only way to go. Android lets you create and load files on the device's internal or external media, providing support for temporary caches and storing files in publicly accessible folders.

There are two lightweight techniques for saving simple application data for Android applications: Shared

Preferences and a set of event handlers used for saving Activity instance state. Both mechanisms use an NVP mechanism to store simple primitive values. Both techniques support primitive types Boolean, string, float, long, and integer, making them ideal means of quickly storing default values, class instance variables, the current UI state, and user preferences.

Creating and Saving Shared Preferences

Using the `SharedPreferences` class, you can create named maps of name/value pairs that can be persisted across sessions and shared among application components running within the same application sandbox.

To create or modify a Shared Preference, call `getSharedPreferences` on the current Context, passing in the name of the Shared Preference to change.

```
SharedPreferences mySharedPreferences =
getSharedPreferences(MY_PREFS,

 Activity.MODE_PRIVATE);
```

Shared Preferences are stored within the application's sandbox, so they can be shared between an application's components but aren't available to other applications.

To modify a Shared Preference, use the `SharedPreferences.Editor` class. Get the Editor object by calling `edit` on the Shared Preferences object you want to change.

```
SharedPreferences.Editor editor =
mySharedPreferences.edit();
```

Use the `put<type>` methods to insert or update

the values associated with the specified name:

```
// Store new primitive types in the shared
preferences object.
editor.putBoolean("isTrue", true);
editor.putFloat("lastFloat", 1f);
editor.putInt("wholeNumber", 2);
editor.putLong("aNumber", 3l);
editor.putString("textEntryValue", "Not Empty");
```

To save edits, call `apply` or `commit` on the Editor object to save the changes asynchronously or synchronously, respectively.

```
// Commit the changes.
editor.apply();
```

The `apply` method was introduced in Android API level 9 (Android 2.3). Calling it causes a safe asynchronous write of the Shared Preference Editor object to be performed. Because it is asynchronous, it is the preferred technique for saving Shared Preferences.

If you require confi rmation of success or want to support earlier Android releases, you can call the commit method, which blocks the calling thread and returns true once a successful write has completed, or false otherwise.

Retrieving Shared Preferences

Accessing Shared Preferences, like editing and saving them, is done using the `getSharedPreferences` method.

Use the type-safe `get<type>` methods to extract saved values. Each getter takes a key and a default value (used when no value has yet been saved for that key.)

```
// Retrieve the saved values.
boolean isTrue = mySharedPreferences.getBoolean("isTrue",
false);
float lastFloat = mySharedPreferences.getFloat("lastFloat",
0f);
int wholeNumber = mySharedPreferences.getInt("wholeNumber",
1);
long aNumber = mySharedPreferences.getLong("aNumber", 0);
String stringPreference =
  mySharedPreferences.getString("textEntryValue", "");
```

You can return a map of all the available Shared Preferences keys values by calling `getAll`, and check for the existence of a particular key by calling the `contains` method.

```
Map<String, ?> allPreferences =
mySharedPreferences.getAll();
boolean containsLastFloat =
mySharedPreferences.contains("lastFloat");
```

Creating a Settings Activity for the Earthquake Viewer

In the following example you build an Activity to set application preferences for the earthquake viewer last seen in the previous chapter. The Activity lets users configure settings for a more personalized experience. You'll provide the option to toggle automatic updates, control the frequency of updates, and filter the minimum earthquake magnitude displayed.

Creating your own Activity to control user preferences is considered bad practice. Later in this chapter you'll replace this Activity with a standard settings screen using the Preferences Screen classes.

1. Open the Earthquake project you created in Chapter 6, "Using Internet Resources." Add new string resources to the `res/values/strings.xml` file for the labels to be displayed in the Preference Screen. Also, add a string for the new Menu Item that will let users open the Preference Screen:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Earthquake</string>
  <string name="quake_feed">
    http://earthquake.usgs.gov/eqcenter/catalogs/1day-
M2.5.xml
  </string>
  <string name="menu_update">Refresh Earthquakes</string>
  <string name="auto_update_prompt">Auto Update?</string>
  <string name="update_freq_prompt">Update
Frequency</string>
```

```
  <string name="min_quake_mag_prompt">Minimum Quake
Magnitude</string>
  <string name="menu_preferences">Preferences</string>
</resources>
```

2. Create a new `preferences.xml` layout resource in the `res/layout` folder for the `Preferences` Activity. Include a check box for indicating the "automatic update" toggle, and spinners to select the update rate and magnitude filter:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/auto_update_prompt"
  />
  <CheckBox android:id="@+id/checkbox_auto_update"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/update_freq_prompt"
  />
  <Spinner android:id="@+id/spinner_update_freq"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/min_quake_mag_prompt"
  />
  <Spinner android:id="@+id/spinner_quake_mag"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:drawSelectorOnTop="true"
  />
  <LinearLayout
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <Button android:id="@+id/okButton"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
```

```
      android:text="@android:string/ok"
    />
    <Button android:id="@+id/cancelButton"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@android:string/cancel"
    />
  </LinearLayout>
</LinearLayout>
```

3. Create four array resources in a new `res/values/arrays.xml` file. They will provide the values to use for the update frequency and minimum magnitude spinners:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string-array name="update_freq_options">
    <item>Every Minute</item>
    <item>5 minutes</item>
    <item>10 minutes</item>
    <item>15 minutes</item>
    <item>Every Hour</item>
  </string-array>
  <string-array name="magnitude">
    <item>3</item>
    <item>5</item>
    <item>6</item>
    <item>7</item>
    <item>8</item>
  </string-array>
  <string-array name="magnitude_options">
    <item>3</item>
    <item>5</item>
    <item>6</item>
    <item>7</item>
    <item>8</item>
  </string-array>

  <string-array name="update_freq_values">
    <item>1</item>
    <item>5</item>
    <item>10</item>
    <item>15</item>
    <item>60</item>
  </string-array>
</resources>
```

4. Create a `PreferencesActivity` Activity. Override `onCreate` to inflate the layout you created in step 2, and get references to the check box and both the

spinner controls. Then make a call to the `populateSpinners` stub:

```
package com.paad.earthquake;

import android.app.Activity;
import android.content.Context;
import android.content.SharedPreferences;
import android.content.SharedPreferences.Editor;
import android.os.Bundle;
import android.preference.PreferenceManager;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.Spinner;

public class PreferencesActivity extends Activity {

  CheckBox autoUpdate;
  Spinner updateFreqSpinner;
  Spinner magnitudeSpinner;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.preferences);

    updateFreqSpinner =
(Spinner)findViewById(R.id.spinner_update_freq);
    magnitudeSpinner =
(Spinner)findViewById(R.id.spinner_quake_mag);
    autoUpdate =
(CheckBox)findViewById(R.id.checkbox_auto_update);

    populateSpinners();
  }

  private void populateSpinners() {
  }
}
```

5. Fill in the `populateSpinners` method, using Array Adapters to bind each spinner to its corresponding array:

```
private void populateSpinners() {
  // Populate the update frequency spinner
  ArrayAdapter<CharSequence> fAdapter;
  fAdapter = ArrayAdapter.createFromResource(this,
R.array.update_freq_options,
  android.R.layout.simple_spinner_item);
  int spinner_dd_item =
android.R.layout.simple_spinner_dropdown_item;
```

```
    fAdapter.setDropDownViewResource(spinner_dd_item);
    updateFreqSpinner.setAdapter(fAdapter);
    // Populate the minimum magnitude spinner
    ArrayAdapter<CharSequence> mAdapter;
    mAdapter = ArrayAdapter.createFromResource(this,
      R.array.magnitude_options,
      android.R.layout.simple_spinner_item);
    mAdapter.setDropDownViewResource(spinner_dd_item);
    magnitudeSpinner.setAdapter(mAdapter);
  }
```

6. Add public static string values that you'll use to identify the Shared Preference keys you'll use to store each preference value. Update the `onCreate` method to retrieve the named preference and call `updateUIFromPreferences`. The `updateUIFromPreferences` method uses the `get<type>` methods on the Shared Preference object to retrieve each preference value and apply it to the current UI.

Use the default application Shared Preference object to save your settings values:

```
public static final String USER_PREFERENCE =
"USER_PREFERENCE";
public static final String PREF_AUTO_UPDATE =
"PREF_AUTO_UPDATE";
public static final String PREF_MIN_MAG_INDEX =
"PREF_MIN_MAG_INDEX";
public static final String PREF_UPDATE_FREQ_INDEX =
"PREF_UPDATE_FREQ_INDEX";

SharedPreferences prefs;

@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.preferences);

  updateFreqSpinner =
(Spinner)findViewById(R.id.spinner_update_freq);
  magnitudeSpinner =
(Spinner)findViewById(R.id.spinner_quake_mag);
  autoUpdate =
(CheckBox)findViewById(R.id.checkbox_auto_update);
  populateSpinners();
  Context context = getApplicationContext();
  prefs =
```

```
PreferenceManager.getDefaultSharedPreferences(context);
  updateUIFromPreferences();
}

private void updateUIFromPreferences() {
  boolean autoUpChecked =
prefs.getBoolean(PREF_AUTO_UPDATE, false);
  int updateFreqIndex =
prefs.getInt(PREF_UPDATE_FREQ_INDEX, 2);
  int minMagIndex = prefs.getInt(PREF_MIN_MAG_INDEX, 0);
  updateFreqSpinner.setSelection(updateFreqIndex);
  magnitudeSpinner.setSelection(minMagIndex);
  autoUpdate.setChecked(autoUpChecked);
}
```

7. Still in the `onCreate` method, add event handlers for the OK and Cancel buttons. The Cancel button should close the Activity, whereas the OK button should call `savePreferences` first:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.preferences);
  updateFreqSpinner =
(Spinner)findViewById(R.id.spinner_update_freq);
  magnitudeSpinner =
(Spinner)findViewById(R.id.spinner_quake_mag);
  autoUpdate =
(CheckBox)findViewById(R.id.checkbox_auto_update);

  populateSpinners();

  Context context = getApplicationContext();
  prefs =
PreferenceManager.getDefaultSharedPreferences(context);
  updateUIFromPreferences();

  Button okButton = (Button) findViewById(R.id.okButton);
  okButton.setOnClickListener(new View.OnClickListener()
{

    public void onClick(View view) {
      savePreferences();
      PreferencesActivity.this.setResult(RESULT_OK);
      finish();
    }
  });

  Button cancelButton = (Button)
findViewById(R.id.cancelButton);
  cancelButton.setOnClickListener(new
```

```
View.OnClickListener() {

    public void onClick(View view) {

 PreferencesActivity.this.setResult(RESULT_CANCELED);
      finish();
    }
  });
}

private void savePreferences() {
}
```

8. Fill in the `savePreferences` method to record the current preferences, based on the UI selections, to the Shared Preference object:

```
private void savePreferences() {
  int updateIndex =
updateFreqSpinner.getSelectedItemPosition();
  int minMagIndex =
magnitudeSpinner.getSelectedItemPosition();
  boolean autoUpdateChecked = autoUpdate.isChecked();

  Editor editor = prefs.edit();
  editor.putBoolean(PREF_AUTO_UPDATE, autoUpdateChecked);
  editor.putInt(PREF_UPDATE_FREQ_INDEX, updateIndex);
  editor.putInt(PREF_MIN_MAG_INDEX, minMagIndex);
  editor.commit();
}
```

9. That completes the `Preferences` Activity. Make it accessible in the application by adding it to the manifest:

```
<activity android:name=".PreferencesActivity"
          android:label="Earthquake Preferences">
</activity>
```

10. Return to the `Earthquake` Activity, and add support for the new Shared Preferences file and a Menu Item to display the Preferences Activity. Start by adding the new Menu Item. Override the `onCreateOptionsMenu` method to include a new item that opens the Preferences Activity and another to refresh the earthquake list:

```
static final private int MENU_PREFERENCES = Menu.FIRST+1;
static final private int MENU_UPDATE = Menu.FIRST+2;
```

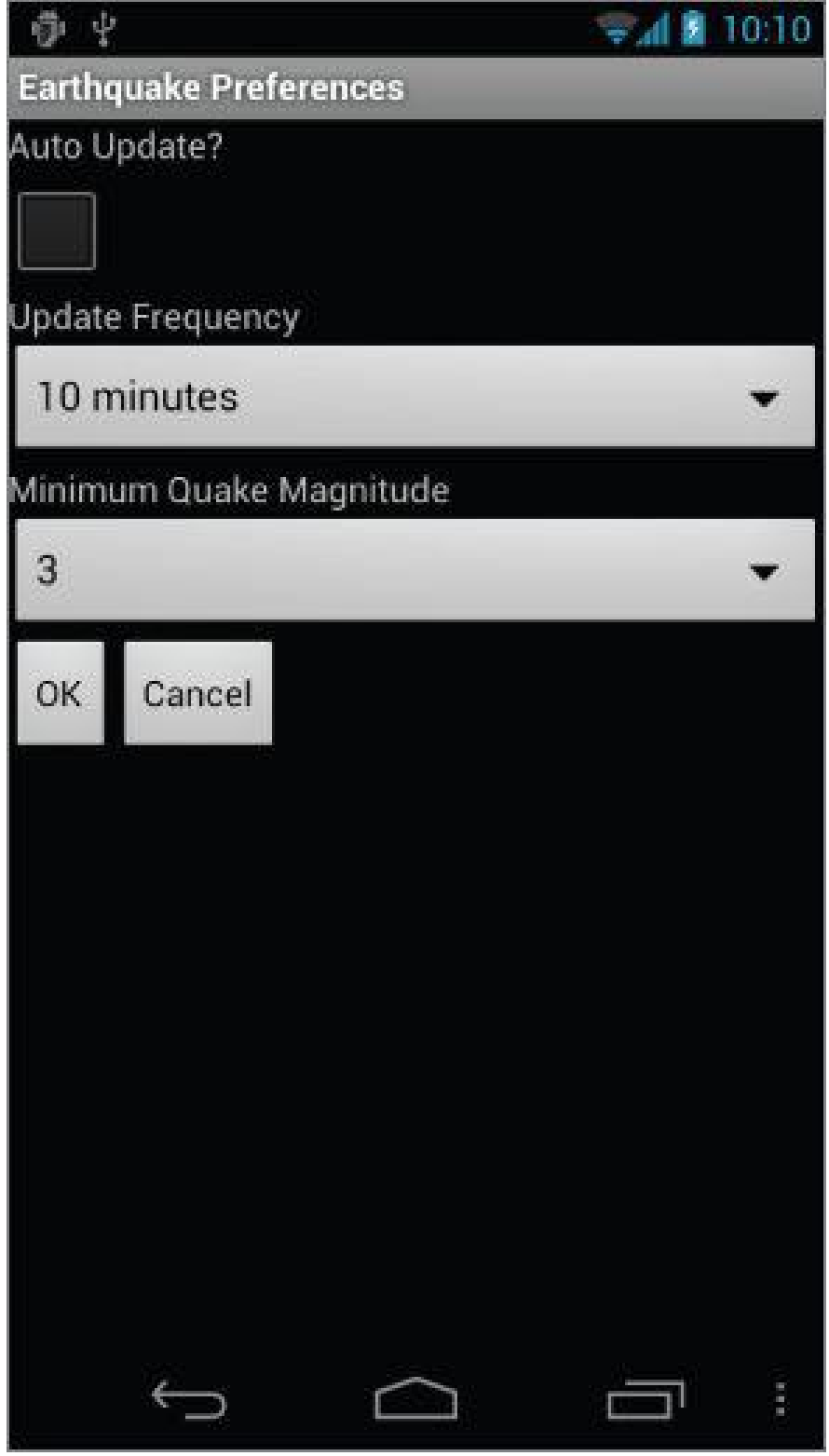
```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
  super.onCreateOptionsMenu(menu);

  menu.add(0, MENU_PREFERENCES, Menu.NONE,
R.string.menu_preferences);

  return true;
}
```

11. Override the `onOptionsItemSelected` method to display the `PreferencesActivity` Activity when the new Menu Item is selected. To launch the Preferences Activity, create an explicit Intent, and pass it in to the `startActivityForResult` method. This will launch the Activity and alert the Earthquake class when the preferences are saved through the `onActivityResult` handler:

```
private static final int SHOW_PREFERENCES = 1;

public boolean onOptionsItemSelected(MenuItem item){
  super.onOptionsItemSelected(item);
  switch (item.getItemId()) {
    case (MENU_PREFERENCES): {
      Intent i = new Intent(this,
  PreferencesActivity.class);
      startActivityForResult(i, SHOW_PREFERENCES);
      return true;
    }
  }
  return false;
}
```

12. Launch your application and select Preferences from the Activity menu. The Preferences Activity should be displayed, as shown in Figure 7.1.

Figure 7.1



13. All that's left is to apply the preferences to the earthquake functionality. Implementing the automatic updates will be left until Chapter 9, “Working in the Background,” where you'll learn to use Services and background threads. For now you can put the framework in place and apply the magnitude filter. Start by creating a new `updateFromPreferences`

method in the Earthquake Activity that reads the Shared Preference values and creates instance variables for each of them:

```
public int minimumMagnitude = 0;
public boolean autoUpdateChecked = false;
public int updateFreq = 0;

private void updateFromPreferences() {
  Context context = getApplicationContext();
  SharedPreferences prefs =

 PreferenceManager.getDefaultSharedPreferences(context);

  int minMagIndex =
prefs.getInt(PreferencesActivity.PREF_MIN_MAG_INDEX, 0);
  if (minMagIndex < 0)
    minMagIndex = 0;

  int freqIndex =
prefs.getInt(PreferencesActivity.PREF_UPDATE_FREQ_INDEX,
0);
  if (freqIndex < 0)
    freqIndex = 0;

  autoUpdateChecked =
prefs.getBoolean(PreferencesActivity.PREF_AUTO_UPDATE,
false);

  Resources r = getResources();
  // Get the option values from the arrays.
  String[] minMagValues =
r.getStringArray(R.array.magnitude);
  String[] freqValues =
r.getStringArray(R.array.update_freq_values);

  // Convert the values to ints.
  minimumMagnitude =
Integer.valueOf(minMagValues[minMagIndex]);
  updateFreq = Integer.valueOf(freqValues[freqIndex]);
}
```

14. Apply the magnitude filter by updating the `addNewQuake` method from the `EarthquakeListFragment` to check a new earthquake's magnitude before adding it to the list:

```
private void addNewQuake(Quake _quake) {
  Earthquake earthquakeActivity =
(Earthquake)getActivity();
  if (_quake.getMagnitude() >
earthquakeActivity.minimumMagnitude) {
    // Add the new quake to our list of earthquakes.
```

```
      earthquakes.add(_quake);

    }

    // Notify the array adapter of a change.
    aa.notifyDataSetChanged();
  }
```

15. Return to the Earthquake Activity and override the `onActivityResult` handler to call `updateFromPreferences` and refresh the earthquakes whenever the `Preferences` Activity saves changes. Note that once again you are creating a new Thread on which to execute the earthquake refresh code.

```
  @Override
  public void onActivityResult(int requestCode, int
  resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    if (requestCode == SHOW_PREFERENCES)
      if (resultCode == Activity.RESULT_OK) {
        updateFromPreferences();
        FragmentManager fm = getFragmentManager();
        final EarthquakeListFragment earthquakeList =

  (EarthquakeListFragment)fm.findFragmentById(R.id.EarthquakeListFragment);

        Thread t = new Thread(new Runnable() {
          public void run() {
            earthquakeList.refreshEarthquakes();
          }
        });
        t.start();
      }
  }
```

16. Finally, call `updateFromPreferences` in `onCreate` of the Earthquake Activity to ensure the preferences are applied when the Activity starts:

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

    updateFromPreferences();
  }
```

All code snippets in this example are part of the *Chapter 7 Earthquake Part 1* project, available for download at www.wrox.com.

Introducing the Preference Framework and the Preference Activity

Android offers an XML-driven framework to create system-style Preference Screens for your applications. By using this framework you can create Preference Activities that are consistent with those used in both native and other third-party applications.

This has two distinct advantages:

- Users will be familiar with the layout and use of your settings screens.
- You can integrate settings screens from other applications (including system settings such as location settings) into your application's preferences.

The preference framework consists of four parts:

- **Preference Screen layout**—An XML file that defines the hierarchy of items displayed in your Preference screens. It specifies the text and associated controls to display, the allowed values,

and the Shared Preference keys to use for each control.

- **Preference Activity** and **Preference Fragment**—Extensions of `PreferenceActivity` and `PreferenceFragment` respectively, that are used to host the Preference Screens. Prior to Android 3.0, Preference Activities hosted the Preference Screen directly; since then, Preference Screens are hosted by Preference Fragments, which, in turn, are hosted by Preference Activities.
- **Preference Header definition**—An XML file that defines the Preference Fragments for your application and the hierarchy that should be used to display them.
- **Shared Preference Change Listener**—An implementation of the `OnSharedPreferenceChangeListener` class used to listen for changes to Shared Preferences.

Android API level 11 (Android 3.0) introduced significant changes to the preference framework by introducing the concept of Preference Fragments and Preference Headers. This is now the preferred technique for creating Activity Preference screens.

As of the time of writing, Preference Fragments are not included in the support library, restricting their use to devices Android 3.0 and above.

The following sections describe the best practice techniques for creating Activity screens for Android 3.0+ devices, making note of

how to achieve similar functionality for older devices.

Defining a Preference Screen Layout in XML

Unlike in the standard UI layout, preference definitions are stored in the `res/xml` resources folder.

Although conceptually they are similar to the UI layout resources described in Chapter 4, “Building User Interfaces,” Preference Screen layouts use a specialized set of controls designed specifically for preferences. These native preference controls are described in the next section.

Each preference layout is defined as a hierarchy, beginning with a single `PreferenceScreen` element:

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

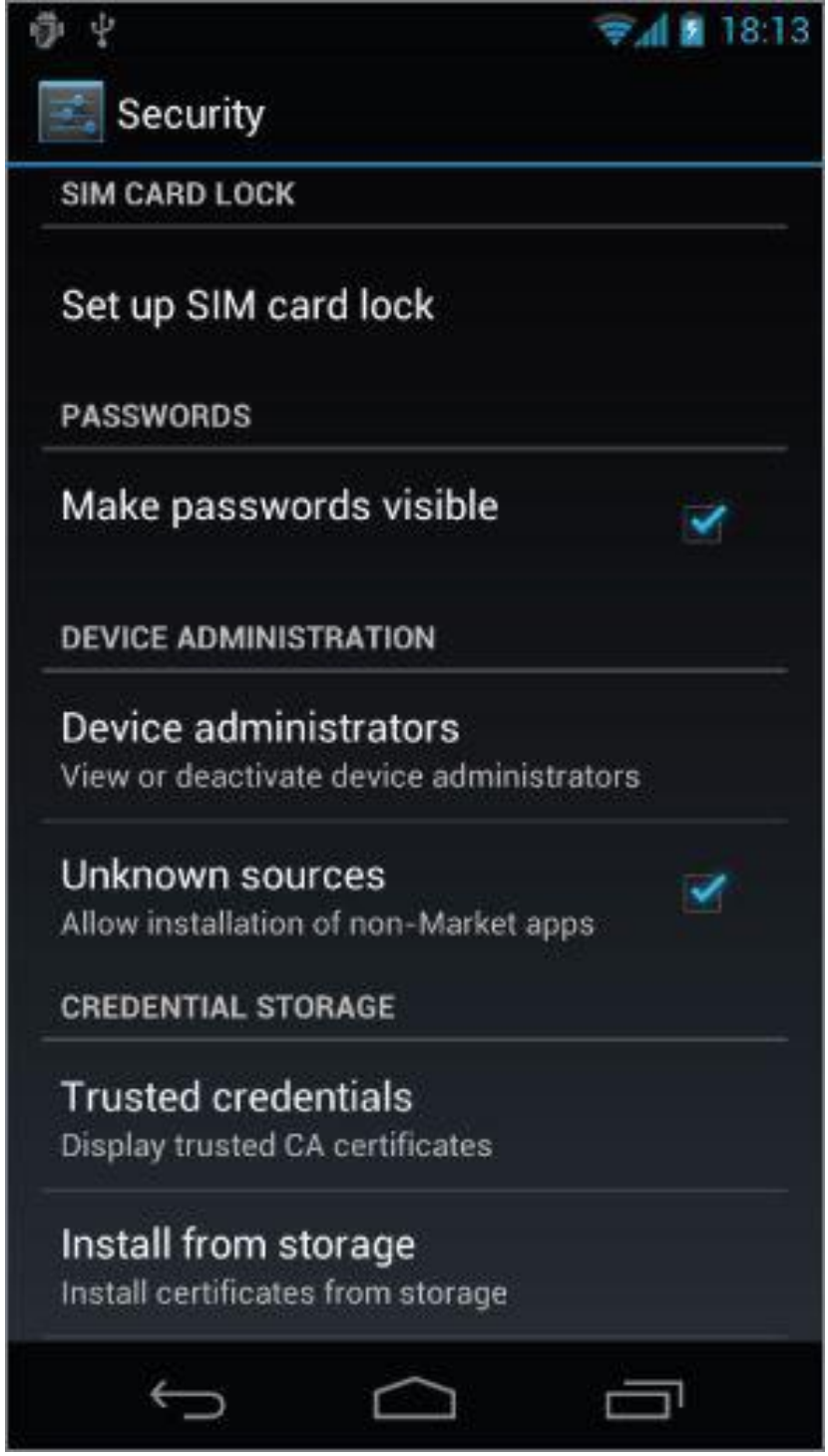
You can include additional Preference Screen elements, each of which will be represented as a selectable element that will display a new screen when clicked.

Within each Preference Screen you can include any combination of `PreferenceCategory` and `Preference<control>` elements. Preference Category elements, as shown in the following snippet, are used to break each Preference Screen into subcategories using a title bar separator:

```
<PreferenceCategory
  android:title="My Preference Category"/>
```

Figure 7.2 shows the SIM card lock, device administration, and credential storage Preference Categories used on the Security Preference Screen.

Figure 7.2



All that remains is to add the preference controls that will be used to set the preferences. Although the specific attributes available for each preference control vary, each of them includes at least the following four:

- `android:key`—The Shared Preference key against which the selected value will be recorded.

- `android:title`—The text displayed to represent the preference.
- `android:summary`—The longer text description displayed in a smaller font below the title text.
- `android:defaultValue`—The default value that will be displayed (and selected) if no preference value has been assigned to the associated preference key.

Listing 7.1 shows a sample Preference Screen that includes a Preference Category and CheckBox Preference.

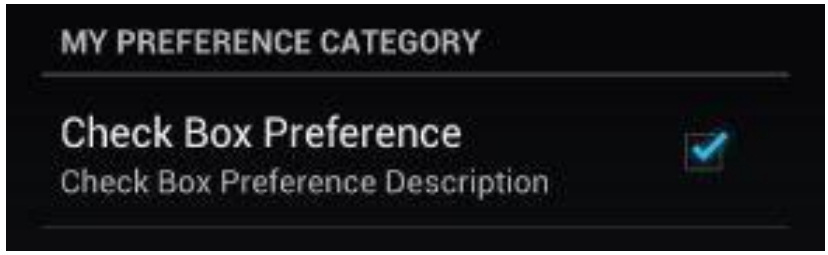
Listing 7.1: A simple Shared Preferences screen

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
  xmlns:android="http://schemas.android.com/apk/res/android">
  <PreferenceCategory
    android:title="My Preference Category">
    <CheckBoxPreference
      android:key="PREF_CHECK_BOX"
      android:title="Check Box Preference"
      android:summary="Check Box Preference
Description"
      android:defaultValue="true"
    />
  </PreferenceCategory>
</PreferenceScreen>
```

code snippet
PA4AD_Ch07_Preferences/res/xml/userpreferences.xml

When displayed, this Preference Screen will appear as shown in Figure 7.3. You'll learn how to display a Preference Screen later in this chapter.

Figure 7.3



Native Preference Controls

Android includes several preference controls to build your Preference Screens:

- `CheckBoxPreference`—A standard preference check box control used to set preferences to true or false.
- `EditTextPreference`—Allows users to enter a string value as a preference. Selecting the preference text at run time will display a text-entry dialog.
- `ListPreference`—The preference equivalent of a spinner. Selecting this preference will display a dialog box containing a list of values from which to select. You can specify different arrays to contain the display text and selection values.
- `MultiSelectListPreference`—Introduced in Android 3.0 (API level 11), this is the preference equivalent of a check box list.
- `RingtonePreference`—A specialized List Preference that presents the list of available ringtones for user selection. This is particularly useful when you're constructing a screen to configure notification settings.

You can use each preference control to construct your Preference Screen hierarchy. Alternatively, you can create your own specialized preference controls by extending the `Preference` class (or any of the subclasses listed above).

You can find further details about preference controls at http://developer.android.com/reference/android/preference/Preference.html.

Using Intents to Import System Preferences into Preference Screens

In addition to including your own Preference Screens, preference hierarchies can include Preference Screens from other applications—including system preferences.

You can invoke any Activity within your Preference Screen using an Intent. If you add an Intent node within a Preference Screen element, the system will interpret this as a request to call `startActivity` using the specified action. The following XML snippet adds a link to the system display settings:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android"
  android:title="Intent preference"
  android:summary="System preference imported
using an intent">
  <intent
android:action="android.settings.DISPLAY_SETTINGS
"/>
</PreferenceScreen>
```

The `android.provider.Settings`

class includes a number of `android.settings.*` constants that can be used to invoke the system settings screens. To make your own Preference Screens available for invocation using this technique, simply add an Intent Filter to the manifest entry for the host Preference Activity (described in detail in the following section):

```
<activity android:name=".UserPreferences"
android:label="My User Preferences">
  <intent-filter>
    <action
android:name="com.paad.myapp.ACTION_USER_PREFERENCE"
/>
  </intent-filter>
</activity>
```

Introducing the Preference Fragment

Since Android 3.0, the `PreferenceFragment` class has been used to host the preference screens defined by Preferences Screen resources. To create a new Preference Fragment, extend the `PreferenceFragment` class, as follows:

```
public class MyPreferenceFragment extends PreferenceFragment
```

To inflate the preferences, override the `onCreate` handler and call `addPreferencesFromResource`, as shown here:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  addPreferencesFromResource(R.xml.userpreferences);
}
```

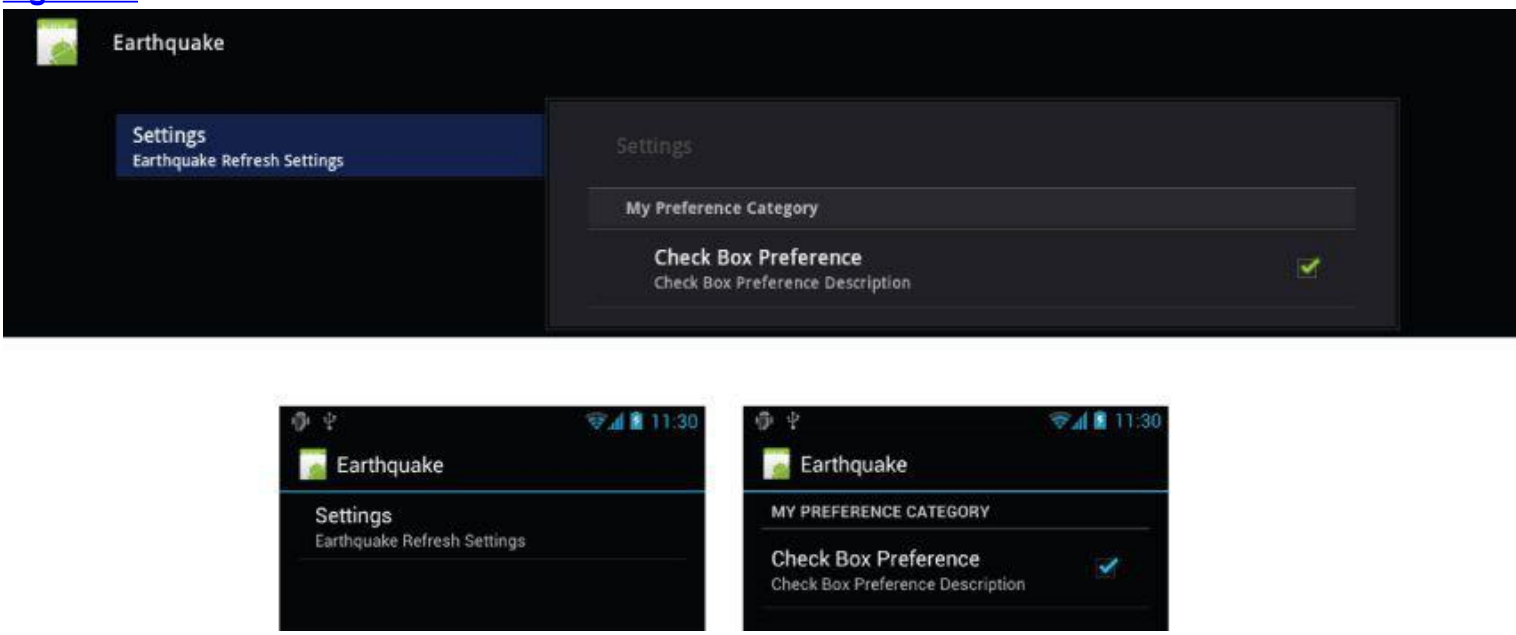
Your application can include several different Preference Fragments, which will be grouped according to the Preference Header hierarchy and displayed within a Preference Activity, as described in the following sections.

Defining the Preference Fragment Hierarchy Using Preference Headers

Preference headers are XML resources that describe how your Preference Fragments should be grouped and displayed within a Preference Activity. Each header identifies and allows you to select a particular Preference Fragment.

The layout used to display the headers and their associated Fragments can vary depending on the screen size and OS version. Figure 7.4 shows examples of how the same Preference Header definition is displayed on a phone and tablet.

Figure 7.4



Preference Headers are XML resources stored in the `res/xml` folder of your project hierarchy. The resource ID for each header is the filename (without extension).

Each header must be associated with a particular Preference Fragment that will be displayed when its header is selected. You must also specify a title and, optionally, a summary and icon resource to represent each Fragment and the Preference Screen it contains, as shown in Listing 7.2.

Listing 7.2: Defining a Preference Headers resource

```
<preference-headers
xmlns:android="http://schemas.android.com/apk/res/android">
  <header
android:fragment="com.paad.preferences.MyPreferenceFragment"
          android:icon="@drawable/preference_icon"
          android:title="My Preferences"
          android:summary="Description of these preferences"
/>
</preference-headers>
```

code snippet
PA4AD_Ch07_Preferences/res/xml/preferenceheaders.xml

Like Preference Screens, you can invoke any Activity within your Preference Headers using an Intent. If you add an Intent node within a header element, as shown in the following snippet, the system will interpret this as a request to call `startActivity` using the specified action:

```
<header android:icon="@drawable/ic_settings_display"
        android:title="Intent"
        android:summary="Launches an Intent.">
        <intent
android:action="android.settings.DISPLAY_SETTINGS "/>
</header>
```

Introducing the Preference Activity

The `PreferenceActivity` class is used to host the Preference Fragment hierarchy defined by a preference headers resource. Prior to Android 3.0, the Preference Activity was used to host Preference Screens directly. For applications that target devices prior to Android 3.0, you may still need to use the Preference Activity in this way.

To create a new Preference Activity, extend the `PreferenceActivity` class as follows:

```
public class MyFragmentPreferenceActivity extends
PreferenceActivity
```

When using Preference Fragments and headers, override the `onBuildHeaders` handler, calling `loadHeadersFromResource` and specifying your preference headers resource file:

```
public void onBuildHeaders(List<Header> target) {
  loadHeadersFromResource(R.xml.userpreferenceheaders,
target);
}
```

For legacy applications, you can inflate the Preference Screen directly in the same way as you would from a Preference Fragment—by overriding the `onCreate` handler and calling `addPreferencesFromResource`, specifying the Preference Screen layout XML resource to display within that Activity:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  addPreferencesFromResource(R.xml.userpreferences);
}
```

Like all Activities, the Preference Activity must be included in the application manifest:

```
<activity android:name=".MyPreferenceActivity"
          android:label="My Preferences">
</activity>
```

To display the application settings hosted in this Activity, open it by calling `startActivity` or `startActivityForResult`:

```
Intent i = new Intent(this,
MyPreferenceActivity.class);
startActivityForResult(i,
SHOW_PREFERENCES);
```

Backward Compatibility and Preference Screens

As noted earlier, the Preference Fragment and associated Preference Headers are not supported on Android platforms prior to Android 3.0 (API level 11). As a result, if you want to create applications that support devices running on both pre- and post-Honeycomb devices, you need to implement separate Preference Activities to support both, and launch the appropriate Activity at run time, as shown in Listing 7.3.

Listing 7.3: Runtime selection of pre- or post-Honeycomb Preference Activities

```
Class c = Build.VERSION.SDK_INT <
Build.VERSION_CODES.HONEYCOMB ?
  MyPreferenceActivity.class :
MyFragmentPreferenceActivity.class;

Intent i = new Intent(this, c);
startActivityForResult(i, SHOW_PREFERENCES);
```

code snippet PA4AD Ch07_Preferences/src/MyActivity.java

Finding and Using the Shared Preferences Set by Preference Screens

The Shared Preference values recorded for the options presented in a Preference Activity are stored within the application's sandbox. This lets any application component, including Activities, Services, and Broadcast Receivers, access the values, as shown in the following snippet:

```
Context context = getApplicationContext();
SharedPreferences prefs =
PreferenceManager.getDefaultSharedPreferences(context);
// TODO Retrieve values using get<type> methods.
```

Introducing On Shared Preference Change Listeners

The `onSharedPreferenceChangeListener` can be implemented to invoke a callback whenever a particular Shared Preference value is added, removed, or modified.

This is particularly useful for Activities and Services that use the Shared Preference framework to set application preferences. Using this handler, your application components can listen for changes to user preferences and update their UIs or behavior, as required.

Register your On Shared Preference Change Listeners using the Shared Preference you want to monitor:

```
public class MyActivity extends Activity implements
  OnSharedPreferenceChangeListener {

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

    // Register this OnSharedPreferenceChangeListener
    SharedPreferences prefs =
      PreferenceManager.getDefaultSharedPreferences(this);
    prefs.registerOnSharedPreferenceChangeListener(this);
  }

  public void onSharedPreferenceChanged(SharedPreferences
prefs,
                                        String key) {
    // TODO Check the shared preference and key parameters
    // and change UI or behavior as appropriate.
  }
}
```

Creating a Standard Preference Activity for the Earthquake Viewer

Previously in this chapter you created a custom Activity to let users modify the application settings for the earthquake viewer. In this example you replace this custom Activity with the standard application settings framework described in the previous section.

This example describes two ways of creating a Preference Activity—first, using the legacy `PreferencesActivity`, and then a backward-compatible alternative using the newer `PreferenceFragment` techniques.

1. Start by creating a new XML resource folder at `res/xml`. Within it create a new `userpreferences.xml` file. This file will define the settings UI for your earthquake application settings. Use the same controls and data sources as in the previous Activity, but this time create them using the standard application settings framework. Note that in this example difference key names are selected. This is because where you were previously recording integers, you're now recording strings. To avoid type mismatches when the application attempts to read the saved preferences, use a different key name.

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

```
  <CheckBoxPreference
    android:key="PREF_AUTO_UPDATE"
    android:title="Auto refresh"
    android:summary="Select to turn on automatic
updating"
    android:defaultValue="true"
  />
  <ListPreference
   android:key="PREF_UPDATE_FREQ"
   android:title="Refresh frequency"
   android:summary="Frequency at which to refresh
earthquake list"
   android:entries="@array/update_freq_options"
   android:entryValues="@array/update_freq_values"
   android:dialogTitle="Refresh frequency"
   android:defaultValue="60"
  />
  <ListPreference
    android:key="PREF_MIN_MAG"
    android:title="Minimum magnitude"
    android:summary="Select the minimum magnitude
earthquake to report"
    android:entries="@array/magnitude_options"
    android:entryValues="@array/magnitude"
    android:dialogTitle="Magnitude"
    android:defaultValue="3"
  />
</PreferenceScreen>
```

2. Open the `PreferencesActivity` Activity and modify its inheritance to extend `PreferenceActivity`:

```
public class PreferencesActivity extends
PreferenceActivity {
```

3. The Preference Activity will handle the controls used in the UI, so you can remove the variables used to store the check box and spinner objects. You can also remove the `populateSpinners`, `updateUIFromPreferences`, and `savePreferences` methods. Update the preference name strings to match those used in the user preferences definition in step 1.

```
public static final String PREF_MIN_MAG = "PREF_MIN_MAG";
public static final String PREF_UPDATE_FREQ =
"PREF_UPDATE_FREQ";
```

4. Update `onCreate` by removing all the references to

the UI controls and the OK and Cancel buttons. Instead of using these, inflate the `userpreferences.xml` file you created in step 1:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  addPreferencesFromResource(R.xml.userpreferences);
}
```

5. Open the Earthquake Activity and update the `updateFromPreferencesMethod`. Using this technique, the selected value itself is stored in the preferences, so there's no need to perform the array lookup steps.

```
private void updateFromPreferences() {
  Context context = getApplicationContext();
  SharedPreferences prefs =

   PreferenceManager.getDefaultSharedPreferences(context);

   minimumMagnitude =

   Integer.parseInt(prefs.getString(PreferencesActivity.PREF_MIN_MAG,
"3"));
   updateFreq =

   Integer.parseInt(prefs.getString(PreferencesActivity.PREF_UPDATE_FREQ,
"60"));

   autoUpdateChecked =
prefs.getBoolean(PreferencesActivity.PREF_AUTO_UPDATE,
false);
}
```

6. Update the `onActivityResult` handler to remove the check for the return value. Using this mechanism, all changes to user preferences are applied as soon as they are made.

```
public void onActivityResult(int requestCode, int
resultCode, Intent data) {
  super.onActivityResult(requestCode, resultCode,
  data);

  if (requestCode == SHOW_PREFERENCES)
    updateFromPreferences();

    FragmentManager fm = getFragmentManager();
    EarthquakeListFragment earthquakeList =
```

```
    (EarthquakeListFragment)
    fm.findFragmentById(R.id.EarthquakeListFragment);

      Thread t = new Thread(new Runnable() {
        public void run() {
          earthquakeList.refreshEarthquakes();
        }
      });
      t.start();
  }
```

7. If you run your application and select the Preferences Menu Item, your new "native" settings screen should be visible, as shown in Figure 7.5.

Figure 7.5

Now create an backward-compatible alternative implementation using the newer Preference Fragments and Preference Headers.

1. Start by creating a new `UserPreferenceFragment` class that extends the Preference Fragment:

```
public class UserPreferenceFragment extends
```

```
PreferenceFragment
```

2. Override its `onCreate` handler to populate the Fragment with the Preference screen, as you did in step 4 above to populate the legacy Preference Activity:

```
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  addPreferencesFromResource(R.xml.userpreferences);
}
```

3. Add your Preference Fragment to a new `preference_headers.xml` file in the `res/xml` folder.

```
<preference-headers
xmlns:android="http://schemas.android.com/apk/res/android">
  <header
android:fragment="com.paad.earthquake.UserPreferenceFragment"
          android:title="Settings"
          android:summary="Earthquake Refresh Settings"
/>
</preference-headers>
```

4. Make a copy of the `PreferencesActivity` class, naming the copy `FragmentPreferences`:

```
public class FragmentPreferences extends
PreferenceActivity
```

5. Add the new User Fragment Preferences Activity to the application manifest.

```
<activity android:name=".FragmentPreferences"/>
```

6. Open the User Fragment Preferences Activity and remove the `onCreate` handler completely. Instead, override the `onBuildHeaders` method, inflating the headers you defined in step 3:

```
@Override
public void onBuildHeaders(List<Header> target) {
  loadHeadersFromResource(R.xml.preference_headers,
target);
}
```

7. Finally, open the Earthquake Activity and modify the `onOptionsItemSelected` method to select the

appropriate Preference Activity. Create an explicit Intent based on the host platform version and pass it in to the `startActivityForResult` method:

```
private static final int SHOW_PREFERENCES = 1;
public boolean onOptionsItemSelected(MenuItem item){
  super.onOptionsItemSelected(item);
  switch (item.getItemId()) {

    case (MENU_PREFERENCES): {
      Class c = Build.VERSION.SDK_INT <
Build.VERSION_CODES.HONEYCOMB ?
        PreferencesActivity.class :
FragmentPreferences.class;
      Intent i = new Intent(this, c);

      startActivityForResult(i, SHOW_PREFERENCES);
      return true;
    }
  }
  return false;
}
```

All the code snippets in this example are part of the *Chapter 7 Earthquake Part 2* project, available for download at `www.wrox.com`.

Persisting the Application Instance State

To save Activity instance variables, Android offers two specialized variations of Shared Preferences. The first uses a Shared Preference named specifically for your Activity, whereas the other relies on a series of lifecycle event handlers.

Saving Activity State Using Shared Preferences

If you want to save Activity information that doesn't need to be shared with other components (e.g., class instance variables), you can call `Activity.getPreferences()` without specifying a Shared Preferences name. This returns a Shared Preference using the calling Activity's class name as the Shared Preference name.

```
// Create or retrieve the activity preference object.
SharedPreferences activityPreferences =
  getPreferences(Activity.MODE_PRIVATE);

// Retrieve an editor to modify the shared preferences.
SharedPreferences.Editor editor = activityPreferences.edit();

// Retrieve the View
TextView myTextView = (TextView)findViewById(R.id.myTextView);

// Store new primitive types in the shared preferences object.
editor.putString("currentTextValue",
                 myTextView.getText().toString());

// Commit changes.
editor.apply();
```

Saving and Restoring Activity Instance State Using the Lifecycle Handlers

Activities offer the `onSaveInstanceState` handler to persist data associated with UI state across sessions. It's designed specifically to persist UI state should an Activity be terminated by the run time, either in an effort to free resources for foreground applications or to accommodate restarts caused by hardware configuration changes.

If an Activity is closed by the user (by pressing the Back button), or programmatically with a call to `finish`, the instance state bundle will not be passed in to `onCreate` or `onRestoreInstanceState` when the Activity is next created. Data that should be persisted across user sessions should be stored using Shared Preferences, as described in the previous sections.

By overriding an Activity's `onSaveInstanceState` event handler, you can use its `Bundle` parameter to save UI instance values. Store values using the same `put` methods as shown for Shared Preferences, before passing the modified Bundle into the superclass's handler:

```
private static final String TEXTVIEW_STATE_KEY =
"TEXTVIEW_STATE_KEY";

@Override
public void onSaveInstanceState(Bundle saveInstanceState) {
  // Retrieve the View
  TextView myTextView =
```

```
(TextView)findViewById(R.id.myTextView);

  // Save its state
  saveInstanceState.putString(TEXTVIEW_STATE_KEY,
    myTextView.getText().toString());

  super.onSaveInstanceState(saveInstanceState);
}
```

This handler will be triggered whenever an Activity completes its active lifecycle, but only when it's not being explicitly finished (with a call to `finish`). As a result, it's used to ensure a consistent Activity state between active lifecycles of a single user session.

The saved Bundle is passed in to the `onRestoreInstanceState` and `onCreate` methods if the application is forced to restart during a session.

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

  TextView myTextView =
(TextView)findViewById(R.id.myTextView);

  String text = "";
  if (savedInstanceState != null &&
      savedInstanceState.containsKey(TEXTVIEW_STATE_KEY))
    text =
savedInstanceState.getString(TEXTVIEW_STATE_KEY);

  myTextView.setText(text);
}
```

Saving and Restoring Fragment Instance State Using the Lifecycle Handlers

The UI for most applications will be encapsulated within Fragments. Accordingly, Fragments also include an `onSaveInstanceState` handler that works in much the same way as its Activity counterpart.

The instance state persisted in the bundle is passed as a parameter to the Fragment's `onCreate`, `onCreateView`, and `onActivityCreated` handlers.

If an Activity is destroyed and restarted to handle a hardware configuration change, such as the screen orientation changing, you can request that your Fragment instance be retained. By calling `setRetainInstance` within a Fragment's `onCreate` handler, you specify that Fragment's instance should not be killed and restarted when its associated Activity is re-created.

As a result, the `onDestroy` and `onCreate` handlers for a retained Fragment will not be called when the device configuration changes and the attached Activity is destroyed and re-created. This can provide a significant efficiency improvement if you move the majority of your object creation into `onCreate`, while using `onCreateView` to update the UI with the values stored within those persisted

instance values.

Note that the rest of the Fragment's lifecycle handlers, including `onAttach`, `onCreateView`, `onActivityCreated`, `onStart`, `onResume`, and their corresponding tear-down handlers, will still be called.

Listing 7.4 shows how to use the lifecycle handlers to record the current UI state while taking advantage of the efficiency gains associated with retaining the Fragment instance.

Listing 7.4: Persisting UI state by using lifecycle handlers and retaining Fragment instances

```
public class MyFragment extends Fragment {

  private static String USER_SELECTION = "USER_SELECTION";
  private int userSelection = 0;
  private TextView tv;

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setRetainInstance(true);
    if (savedInstanceState != null)
      userSelection =
savedInstanceState.getInt(USER_SELECTION);
  }

  @Override
  public View onCreateView(LayoutInflater inflater,
                           ViewGroup container,
                           Bundle savedInstanceState) {
```

```
    View v = inflater.inflate(R.layout.mainfragment,
container, false);

    tv = (TextView)v.findViewById(R.id.text);
    setSelection(userSelection);

    Button b1 = (Button)v.findViewById(R.id.button1);
    Button b2 = (Button)v.findViewById(R.id.button2);
    Button b3 = (Button)v.findViewById(R.id.button3);

    b1.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
        setSelection(1);
      }
    });

    b2.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
        setSelection(2);
      }
    });

    b3.setOnClickListener(new OnClickListener() {
      public void onClick(View arg0) {
        setSelection(3);
      }
    });

    return v;
  }

  private void setSelection(int selection) {
    userSelection = selection;
    tv.setText("Selected: " + selection);
  }

  @Override
  public void onSaveInstanceState(Bundle outState) {
    outState.putInt(USER_SELECTION, userSelection);
    super.onSaveInstanceState(outState);
  }
}
```

code snippet PA4AD_Ch07_Preferences/src/MyFragment.java

Including Static Files as Resources

If your application requires external file resources, you can include them in your distribution package by placing them in the `res/raw` folder of your project hierarchy.

To access these read-only file resources, call the `openRawResource` method from your application's `Resource` object to receive an `InputStream` based on the specified file. Pass in the filename (without the extension) as the variable name from the `R.raw` class, as shown in the following skeleton code:

```
Resources myResources = getResources();
InputStream myFile =
myResources.openRawResource(R.raw.myfilename);
```

Adding raw files to your resources hierarchy is an excellent alternative for large, preexisting data sources (such as dictionaries) for which it's not desirable (or even possible) to convert them into Android databases.

Android's resource mechanism lets you specify alternative resource files for different languages, locations, and hardware configurations. For example, you could create an application that loads a different

dictionary resource based on the user's language settings.

Working with the File System

It's good practice to use Shared Preferences or a database to store your application data, but there may still be times when you'll want to use files directly rather than rely on Android's managed mechanisms—particularly when working with multimedia files.

File-Management Tools

Android supplies some basic file-management tools to help you deal with the file system. Many of these utilities are located within the `java.io.File` package.

Complete coverage of Java file-management utilities is beyond the scope of this book, but Android does supply some specialized utilities for file management that are available from the application Context.

- `deleteFile`—Enables you to remove files created by the current application
- `fileList`—Returns a string array that includes all the files created by the current application

These methods are particularly useful for cleaning up temporary files left behind if your application crashes or is killed unexpectedly.

Using Application-Specific Folders to Store Files

Many applications will create or download files that are specific to the application. There are two options for storing these application-specific files: internally or externally.

When referring to the *external* storage, we refer to the shared/media storage that is accessible by all applications and can typically be mounted to a computer file system when the device is connected via USB. Although it is typically located on the SD Card, some devices implement this as a separate partition on the internal storage.
The most important thing to remember when storing files on external storage is that no security is enforced on files stored here. Any application can access, overwrite, or delete files stored on the external storage.
It's also important to remember that files stored on external storage may not always be available. If the SD Card is ejected, or the device is mounted for access via a computer, your application will be unable to read (or create) files on the external storage.

Android offers two corresponding methods via the application Context, `getDir` and `getExternalFilesDir`, both of which return a File object that contains the path to the internal and external application file storage directory, respectively.

All files stored in these directories or the subfolders will be erased when your application is uninstalled.

The `getExternalFilesDir` method was introduced in Android API level 8 (Android 2.2). To support earlier platform releases, you can call `Environment.getExternalStorageDirectory` to return a path to the root of the external storage.

It's good practice to store your application-specific data in its own subdirectory using the same style as `getExternalFilesDir`—that is, `/Android/data/[Your Package Name]/files`.

Note that this work-around will not automatically delete your application files when it is uninstalled.

Both of these methods accept a string parameter that can be used to specify the subdirectory into which you want to place your files. In Android 2.2 (API level 8) the `Environment` class introduced a number of `DIRECTORY_[Category]` string constants that represent standard directory names, including downloads, images, movies, music, and camera files.

Files stored in the application folders should be specific to the parent application and are typically not detected by the media-scanner, and therefore won't be added to the Media Library automatically. If your application downloads or creates files that should be added to the Media Library or otherwise made available to other applications, consider putting them in the public external storage directory, as described later in this chapter.

Creating Private Application Files

Android offers the `openFileInput` and `openFileOutput` methods to simplify reading and writing streams from and to files stored in the application's sandbox.

```
String FILE_NAME = "tempfile.tmp";

// Create a new output file stream that's private to this
application.
FileOutputStream fos = openFileOutput(FILE_NAME,
Context.MODE_PRIVATE);
// Create a new file input stream.
FileInputStream fis = openFileInput(FILE_NAME);
```

These methods support only those files in the current application folder; specifying path separators will cause an exception to be thrown.

If the filename you specify when creating a `FileOutputStream` does not exist, Android will create it for you. The default behavior for existing files is to overwrite them; to append an existing file, specify the mode as `Context.MODE_APPEND`.

By default, files created using the `openFileOutput` method are private to the calling application—a different application will be denied access. The standard way to share a file between applications is to use a Content Provider. Alternatively, you can specify either `Context.MODE_WORLD_READABLE` or `Context.MODE_WORLD_WRITEABLE` when creating the

output file, to make it available in other applications, as shown in the following snippet:

```
String OUTPUT_FILE = "publicCopy.txt";
FileOutputStream fos = openFileOutput(OUTPUT_FILE,
Context.MODE_WORLD_WRITEABLE);
```

You can find the location of files stored in your sandbox by calling `getFilesDir`. This method will return the absolute path to the files created using `openFileOutput`:

```
File file = getFilesDir();
Log.d("OUTPUT_PATH_", file.getAbsolutePath());
```

Using the Application File Cache

Should your application need to cache temporary files, Android offers both a managed internal cache, and (since Android API level 8) an unmanaged external cache. You can access them by calling the `getCacheDir` and `getExternalCacheDir` methods, respectively, from the current Context.

Files stored in either cache location will be erased when the application is uninstalled. Files stored in the internal cache will potentially be erased by the system when it is running low on available storage; files stored on the external cache will not be erased, as the system does not track available storage on external media.

In either case it's good form to monitor and manage the size and age of your cache, deleting files when a reasonable maximum cache size is exceeded.

Storing Publicly Readable Files

Android 2.2 (API level 8) also includes a convenience method, `Environment.getExternalStoragePublicDirectory`, that can be used to find a path in which to store your application files. The returned location is where users will typically place and manage their own files of each type.

This is particularly useful for applications that provide functionality that replaces or augments system applications, such as the camera, that store files in standard locations.

The `getExternalStoragePublicDirectory` method accepts a String parameter that determines which subdirectory you want to access using a series of Environment static constants:

- `DIRECTORY_ALARMS`—Audio files that should be available as user-selectable alarm sounds
- `DIRECTORY_DCIM`—Pictures and videos taken by the device
- `DIRECTORY_DOWNLOADS`—Files downloaded by the user
- `DIRECTORY_MOVIES`—Movies
- `DIRECTORY_MUSIC`—Audio files that represent music
- `DIRECTORY_NOTIFICATIONS`—Audio files that should be available as user-selectable notification sounds

- DIRECTORY_PICTURES—Pictures
- DIRECTORY_PODCASTS—Audio files that represent podcasts
- DIRECTORY_RINGTONES—Audio files that should be available as user-selectable ringtones

Note that if the returned directory doesn't exit, you must create it before writing files to the directory, as shown in the following snippet:

```
String FILE_NAME = "MyMusic.mp3";

File path = Environment.getExternalStoragePublicDirectory(
            Environment.DIRECTORY_MUSIC);

File file = new File(path, FILE_NAME);

try {
  path.mkdirs();
  [... Write Files ...]
} catch (IOException e) {
  Log.d(TAG, "Error writing " + FILE_NAME, e);
}
```

Chapter 8

Databases and Content Providers

What's in this Chapter?

- Creating databases and using SQLite
- Using Content Providers, Cursors, and Content Values to store, share, and consume application data
- Asynchronously querying Content Providers using Cursor Loaders
- Adding search capabilities to your applications
- Using the native Media Store, Contacts, and Calendar Content Providers

This chapter introduces persistent data storage in Android, starting with the SQLite database library. SQLite offers a powerful SQL database library that provides a robust persistence layer over which you have total control.

You'll also learn how to build and use Content Providers to store, share, and consume structured data within and between applications. Content Providers offer a generic

interface to any data source by decoupling the data storage layer from the application layer. You'll see how to query Content Providers asynchronously to ensure your application remains responsive.

Although access to a database is restricted to the application that created it, Content Providers offer a standard interface your applications can use to share data with and consume data from other applications—including many of the native data stores.

Having created an application with data to store, you'll learn how to add search functionality to your application and how to build Content Providers that can provide real-time search suggestions.

Because Content Providers can be used across application boundaries, you have the opportunity to integrate your own application with several native Content Providers, including contacts, calendar, and the Media Store. You'll learn how to store and retrieve data from these core Android applications to provide your users with a richer, more consistent, and fully integrated user experience.

Introducing Android Databases

Android provides structured data persistence through a combination of SQLite databases and Content Providers.

SQLite databases can be used to store application data using a managed, structured approach. Android offers a full SQLite relational database library. Every application can create its own databases over which it has complete control.

Having created your underlying data store, Content Providers offer a generic, well-defined interface for using and sharing data that provides a consistent abstraction from the underlying data source.

SQLite Databases

Using SQLite you can create fully encapsulated relational databases for your applications. Use them to store and manage complex, structured application data.

Android databases are stored in the `/data/data/<package_name>/databases` folder on your device (or emulator). All databases are private, accessible only by the application that created them.

Database design is a big topic that deserves more thorough coverage than is possible within this book. It is worth highlighting that standard database best practices still apply in Android. In particular, when you're creating databases for resource-constrained devices (such as mobile phones), it's important to normalize your data to minimize redundancy.

Content Providers

Content Providers provide an interface for publishing and consuming data, based around a simple URI addressing model using the `content://` schema. They enable you to decouple your application layers from the underlying data layers, making your applications data-source agnostic by abstracting the underlying data source.

Content Providers can be shared between applications, queried for results, have their existing records updated or deleted, and have new records added. Any application—with the appropriate permissions—can add, remove, or update data from any other application, including the native Android Content Providers.

Several native Content Providers have been made accessible for access by third-party applications, including the contact manager, media store, and calendar, as described later in this chapter.

By publishing your own Content Providers, you make it possible for you (and other developers) to incorporate and extend your data in new applications.

Introducing SQLite

SQLite is a well-regarded relational database management system (RDBMS). It is:

- Open-source
- Standards-compliant
- Lightweight
- Single-tier

It has been implemented as a compact C library that's included as part of the Android software stack.

By being implemented as a library, rather than running as a separate ongoing process, each SQLite database is an integrated part of the application that created it. This reduces external dependencies, minimizes latency, and simplifies transaction locking and synchronization.

SQLite has a reputation for being extremely reliable and is the database system of choice for many consumer electronic devices, including many MP3 players and smartphones.

Lightweight and powerful, SQLite differs from many conventional database engines by loosely typing each column, meaning that column values are not required to

conform to a single type; instead, each value is typed individually in each row. As a result, type checking isn't necessary when assigning or extracting values from each column within a row.

For more comprehensive coverage of SQLite, including its particular strengths and limitations, check out the official site, at www.sqlite.org.

Content Values and Cursors

Content Values are used to insert new rows into tables. Each `ContentValues` object represents a single table row as a map of column names to values.

Database queries are returned as `Cursor` objects. Rather than extracting and returning a copy of the result values, Cursors are pointers to the result set within the underlying data. Cursors provide a managed way of controlling your position (row) in the result set of a database query.

The `Cursor` class includes a number of navigation functions, including, but not limited to, the following:

- `moveToFirst`—Moves the cursor to the first row in the query result
- `moveToNext`—Moves the cursor to the next row
- `moveToPrevious`—Moves the cursor to the previous row
- `getCount`—Returns the number of rows in the result set
- `getColumnIndexOrThrow`—Returns the zero-based

index for the column with the specified name (throwing an exception if no column exists with that name)

- `getColumnName`—Returns the name of the specified column index
- `getColumnNames`—Returns a string array of all the column names in the current Cursor
- `moveToPosition`—Moves the cursor to the specified row
- `getPosition`—Returns the current cursor position

Android provides a convenient mechanism to ensure queries are performed asynchronously. The `CursorLoader` class and associated Loader Manager (described later in this chapter) were introduced in Android 3.0 (API level 11) and are now also available as part of the support library, allowing you to leverage them while still supporting earlier Android releases.

Later in this chapter you'll learn how to query a database and how to extract specific row/column values from the resulting Cursors.

Working with SQLite Databases

This section shows you how to create and interact with SQLite databases within your applications.

When working with databases, it's good form to encapsulate the underlying database and expose only the public methods and constants required to interact with that database, generally using what's often referred to as a contract or helper class. This class should expose database constants, particularly column names, which will be required for populating and querying the database. Later in this chapter you'll be introduced to Content Providers, which can also be used to expose these interaction constants.

Listing 8.1 shows a sample of the type of database constants that should be made public within a helper class.

Listing 8.1: Skeleton code for contract class constants

```
// The index (key) column name for use in where clauses.
public static final String KEY_ID = "_id";

// The name and column index of each column in your database.
// These should be descriptive.
public static final String KEY_GOLD_HOARD_NAME_COLUMN =
  "GOLD_HOARD_NAME_COLUMN";
public static final String KEY_GOLD_HOARD_ACCESSIBLE_COLUMN =
  "OLD_HOARD_ACCESSIBLE_COLUMN";
public static final String KEY_GOLD_HOARDED_COLUMN =
  "GOLD_HOARDED_COLUMN";
// TODO: Create public field for each column in your table.
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyHoardDatabase.java

Introducing the SQLiteOpenHelper

`SQLiteOpenHelper` is an abstract class used to implement the best practice pattern for creating, opening, and upgrading databases.

By implementing an SQLite Open Helper, you hide the logic used to decide if a database needs to be created or upgraded before it's opened, as well as ensure that each operation is completed efficiently.

It's good practice to defer creating and opening databases until they're needed. The SQLite Open Helper caches database instances after they've been successfully opened, so you can make requests to open the database immediately prior to performing a query or transaction. For the same reason, there is no need to close the database manually unless you no longer need to use it again.

Database operations, especially opening or creating databases, can be time-consuming. To ensure this doesn't impact the user experience, make all database transactions asynchronous.

Listing 8.2 shows how to extend the `SQLiteOpenHelper` class by overriding the constructor, `onCreate`, and `onUpgrade` methods to handle the creation of a new database and upgrading to a new version, respectively.

Listing 8.2: Implementing an SQLite Open Helper

```
private static class HoardDBOpenHelper extends
SQLiteOpenHelper {

  private static final String DATABASE_NAME = "myDatabase.db";
  private static final String DATABASE_TABLE = "GoldHoards";
  private static final int DATABASE_VERSION = 1;

  // SQL Statement to create a new database.
  private static final String DATABASE_CREATE = "create table
" +
    DATABASE_TABLE + " (" + KEY_ID +
    " integer primary key autoincrement, " +
    KEY_GOLD_HOARD_NAME_COLUMN + " text not null, " +
    KEY_GOLD_HOARDED_COLUMN + " float, " +
    KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + " integer);";

  public HoardDBOpenHelper(Context context, String name,
                   CursorFactory factory, int version) {
    super(context, name, factory, version);
  }

  // Called when no database exists in disk and the helper
class needs
  // to create a new one.
  @Override
  public void onCreate(SQLiteDatabase db) {
    db.execSQL(DATABASE_CREATE);
  }

  // Called when there is a database version mismatch meaning
that
  // the version of the database on disk needs to be upgraded
to
  // the current version.
  @Override
  public void onUpgrade(SQLiteDatabase db, int oldVersion,
```

```
                          int newVersion) {
    // Log the version upgrade.
    Log.w("TaskDBAdapter", "Upgrading from version " +
      oldVersion + " to " +
      newVersion + ", which will destroy all old data");

    // Upgrade the existing database to conform to the new
    // version. Multiple previous versions can be handled by
    // comparing oldVersion and newVersion values.

    // The simplest case is to drop the old table and create a
new one.
    db.execSQL("DROP TABLE IF IT EXISTS " + DATABASE_TABLE);
    // Create a new one.
    onCreate(db);
  }
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyHoardDatabase.java

In this example *onUpgrade* simply drops the existing table and replaces it with the new definition. This is often the simplest and most practical solution; however, for important data that is not synchronized with an online service or is hard to recapture, a better approach may be to migrate existing data into the new table.

To access a database using the SQLite Open Helper, call `getWritableDatabase` or `getReadableDatabase` to open and obtain a writable or read-only instance of the underlying database, respectively.

Behind the scenes, if the database doesn't exist, the helper executes its `onCreate` handler. If the database

version has changed, the `onUpgrade` handler will fire. In either case, the `get<read/writ>ableDatabase` call will return the cached, newly opened, newly created, or upgraded database, as appropriate.

When a database has been successfully opened, the SQLite Open Helper will cache it, so you can (and should) use these methods each time you query or perform a transaction on the database, rather than caching the open database within your application.

A call to `getWritableDatabase` can fail due to disk space or permission issues, so it's good practice to fall back to the `getReadableDatabase` method for database queries if necessary. In most cases this method will provide the same, cached writeable database instance as `getWritableDatabase` unless it does not yet exist or the same permission or disk space issues occur, in which case a read-only copy will be returned.

To create or upgrade the database, it must be opened in a writeable form; therefore, it's generally good practice to attempt to open a writeable database first, falling back to a read-only alternative if it fails.

Opening and Creating Databases Without the SQLite Open Helper

If you would prefer to manage the creation, opening, and version control of your databases directly, rather than using the SQLite Open Helper, you can use the application Context's `openOrCreateDatabase` method to create the database itself:

```
SQLiteDatabase db =
context.openOrCreateDatabase(DATABASE_NAME,

Context.MODE_PRIVATE,
                                                    null);
```

After you have created the database, you must handle the creation and upgrade logic handled within the `onCreate` and `onUpgrade` handlers of the SQLite Open Helper—typically using the database's `execSQL` method to create and drop tables, as required.

It's good practice to defer creating and opening databases until they're needed, and to cache database instances after they're successfully opened to limit the associated efficiency costs.

At a minimum, any such operations must be handled asynchronously to avoid impacting the main application thread.

Android Database Design Considerations

You should keep the following Android-specific considerations in mind when designing your database.

- Files (such as bitmaps or audio files) are not usually stored within database tables. Use a string to store a path to the file, preferably a fully qualified URI.
- Although not strictly a requirement, it's strongly recommended that all tables include an auto-increment key field as a unique index field for each row. If you plan to share your table using a Content Provider, a unique ID field is required.

Querying a Database

Each database query is returned as a `Cursor`. This lets Android manage resources more efficiently by retrieving and releasing row and column values on demand.

To execute a query on a Database object, use the `query` method, passing in the following:

- An optional Boolean that specifies if the result set should contain only unique values.
- The name of the table to query.
- A projection, as an array of strings, that lists the columns to include in the result set.
- A `where` clause that defines the rows to be returned. You can include `?` wildcards that will be replaced by the values passed in through the selection argument parameter.
- An array of selection argument strings that will replace the `?` wildcards in the `where` clause.
- A `group by` clause that defines how the resulting rows will be grouped.
- A `having` clause that defines which row groups to include if you specified a `group by` clause.
- A string that describes the order of the returned rows.
- A string that defines the maximum number of rows in the result set.

Listing 8.3 shows how to return a selection of rows from within an SQLite database table.

Listing 8.3: Querying a database

```
// Specify the result column projection. Return the minimum
set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
  KEY_ID, KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
KEY_GOLD_HOARDED_COLUMN };

// Specify the where clause that will limit our results.
String where = KEY_GOLD_HOARD_ACCESSIBLE_COLUMN + "=" + 1;

// Replace these with valid SQL statements as necessary.
String whereArgs[] = null;
String groupBy = null;
String having = null;
String order = null;

SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
Cursor cursor = db.query(HoardDBOpenHelper.DATABASE_TABLE,
                         result_columns, where,
                         whereArgs, groupBy, having, order);
```

code snippet PA4AD_Ch08_GoldHoarder/src/MyHoardDatabase.java

In this Listing 8.3, a database instance is opened using an SQLite Open Helper implementation. The SQLite Open Helper defers the creation and opening of database instances until

they are first required and caches them after they are successfully opened.

As a result, it's good practice to request a database instance each time you perform a query or transaction on the database. For efficiency reasons, you should close your database instance only when you believe you will no longer require it—typically, when the Activity or Service using it is stopped.

Extracting Values from a Cursor

To extract values from a Cursor, first use the `moveTo<location>` methods described earlier to position the cursor at the correct row of the result Cursor, and then use the type-safe `get<type>` methods (passing in a column index) to return the value stored at the current row for the specified column. To find the column index of a particular column within a result Cursor, use its `getColumnIndexOrThrow` and `getColumnIndex` methods.

It's good practice to use `getColumnIndexOrThrow` when you expect the column to exist in all cases. Using `getColumnIndex` and checking for a –1 result, as shown in the following snippet, is a more efficient technique than catching exceptions when the column might not exist in every case.

```
int columnIndex = cursor.getColumnIndex(KEY_COLUMN_1_NAME);
if (columnIndex > -1) {
  String columnValue = cursor.getString(columnIndex);
  // Do something with the column value.
}
else {
  // Do something else if the column doesn't exist.
}
```

Database implementations should publish static constants that provide the column names. These static constants are typically exposed from within the database contract class or the

Content Provider.

Listing 8.4 shows how to iterate over a result Cursor, extracting and averaging a column of float values.

Listing 8.4: Extracting values from a Cursor

```
float totalHoard = 0f;
float averageHoard = 0f;

// Find the index to the column(s) being used.
int GOLD_HOARDED_COLUMN_INDEX =
  cursor.getColumnIndexOrThrow(KEY_GOLD_HOARDED_COLUMN);

// Iterate over the cursors rows.
// The Cursor is initialized at before first, so we can
// check only if there is a "next" row available. If the
// result Cursor is empty this will return false.
while (cursor.moveToNext()) {
  float hoard =
cursor.getFloat(GOLD_HOARDED_COLUMN_INDEX);
  totalHoard += hoard;
}

// Calculate an average -- checking for divide by zero
errors.
float cursorCount = cursor.getCount();
averageHoard = cursorCount > 0 ?
                 (totalHoard / cursorCount) : Float.NaN;

// Close the Cursor when you've finished with it.
cursor.close();
```

code snippet PA4AD_Ch08_GoldHoarder/src/MyHoardDatabase.java

Because SQLite database columns are loosely

typed, you can cast individual values into valid types, as required. For example, values stored as floats can be read back as strings.

When you have finished using your result Cursor, it's important to close it to avoid memory leaks and reduce your application's resource load:

```
cursor.close();
```

Adding, Updating, and Removing Rows

The `SQLiteDatabase` class exposes `insert`, `delete`, and `update` methods that encapsulate the SQL statements required to perform these actions. Additionally, the `execSQL` method lets you execute any valid SQL statement on your database tables, should you want to execute these (or any other) operations manually.

Any time you modify the underlying database values, you should update your Cursors by running a new query.

Inserting Rows

To create a new row, construct a `ContentValues` object and use its `put` methods to add name/value pairs representing each column name and its associated value.

Insert the new row by passing the Content Values into the `insert` method called on the target database—along with the table name—as shown in Listing 8.5.

Listing 8.5: Inserting new rows into a database

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Assign values for each row.
newValues.put(KEY_GOLD_HOARD_NAME_COLUMN, hoardName);
newValues.put(KEY_GOLD_HOARDED_COLUMN, hoardValue);
newValues.put(KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
hoardAccessible);
// [ ... Repeat for each column / value pair ... ]

// Insert the row into your table
SQLiteDatabase db = hoardDBOpenHelper.getWritableDatabase();
db.insert(HoardDBOpenHelper.DATABASE_TABLE, null, newValues);
```

code snippet PA4AD_Ch08_GoldHoarder/src/MyHoardDatabase.java

The second parameter used in the insert method shown in Listing 8.5 is known as the *null column hack*.

If you want to add an empty row to an SQLite database, by passing in an empty Content Values object, you must also pass in the name of a column whose value can be explicitly set to null.

When inserting a new row into an SQLite database, you must always explicitly specify at least one column and a corresponding value, the latter of which can be null. If you set the null column hack parameter to null, as shown in Listing 8.5, when inserting an empty Content Values object SQLite will throw an exception.

It's generally good practice to ensure that your code doesn't attempt to insert empty Content Values into an SQLite database.

Updating Rows

Updating rows is also done with Content Values. Create a new `ContentValues` object, using the `put` methods to assign new values to each column you want to update. Call the `update` method on the database, passing in the table name, the updated Content Values object, and a `where` clause that specifies the row(s) to update, as shown in Listing 8.6.

Listing 8.6: Updating a database row

```
// Create the updated row Content Values.
ContentValues updatedValues = new ContentValues();

// Assign values for each row.
updatedValues.put(KEY_GOLD_HOARDED_COLUMN,
newHoardValue);
// [ ... Repeat for each column to update ... ]

// Specify a where clause the defines which rows should
be
// updated. Specify where arguments as necessary.
String where = KEY_ID + "=" + hoardId;
String whereArgs[] = null;

// Update the row with the specified index with the new
values.
SQLiteDatabase db =
hoardDBOpenHelper.getWritableDatabase();
db.update(HoardDBOpenHelper.DATABASE_TABLE,
updatedValues,
          where, whereArgs);
```

code snippet PA4AD_Ch08_GoldHoarder/src/MyHoardDatabase.java

Deleting Rows

To delete a row, simply call the `delete` method on a database, specifying the table name and a `where` clause that returns the rows you want to delete, as shown in Listing 8.7.

Listing 8.7: Deleting a database row

```
// Specify a where clause that determines which
row(s) to delete.
// Specify where arguments as necessary.
String where = KEY_GOLD_HOARDED_COLUMN + "=" + 0;
String whereArgs[] = null;

// Delete the rows that match the where clause.
SQLiteDatabase db =
hoardDBOpenHelper.getWritableDatabase();
db.delete(HoardDBOpenHelper.DATABASE_TABLE, where,
whereArgs);
```

code snippet PA4AD_Ch08_GoldHoarder/src/MyHoardDatabase.java

Creating Content Providers

Content Providers provide an interface for publishing data that will be consumed using a Content Resolver. They allow you to decouple the application components that consume data from their underlying data sources, providing a generic mechanism through which applications can share their data or consume data provided by others.

To create a new Content Provider, extend the abstract `ContentProvider` class:

```
public class MyContentProvider extends ContentProvider
```

Like the database contract class described in the previous section, it's good practice to include static database constants—particularly column names and the Content Provider authority—that will be required for transacting with, and querying, the database.

You will also need to override the `onCreate` handler to initialize the underlying data source, as well as the `query`, `update`, `delete`, `insert`, and `getType` methods to implement the interface used by the Content Resolver to interact with the data, as described in the following sections.

Registering Content Providers

Like Activities and Services, Content Providers must be registered in your application manifest before the Content Resolver can discover them. This is done using a `provider` tag that includes a `name` attribute describing the Provider's class name and an `authorities` tag.

Use the `authorities` tag to define the base URI of the Provider's authority. A Content Provider's authority is used by the Content Resolver as an address and used to find the database you want to interact with.

Each Content Provider authority must be unique, so it's good practice to base the URI path on your package name. The general form for defining a Content Provider's authority is as follows:

```
com.<CompanyName>.provider.<ApplicationName>
```

The completed `provider` tag should follow the format show in the following XML snippet:

```
<provider android:name=".MyContentProvider"

 android:authorities="com.paad.skeletondatabaseprovider"/>
```

Publishing Your Content Provider's URI Address

Each Content Provider should expose its authority using a public static `CONTENT_URI` property to make it more easily discoverable. This should include a data path to the primary content—for example:

```
public static final Uri CONTENT_URI =
  Uri.parse("content://com.paad.skeletondatabaseprovider/elements");
```

These content URIs will be used when accessing your Content Provider using a Content Resolver. A query made using this form represents a request for all rows, whereas an appended trailing `/<rownumber>`, as shown in the following snippet, represents a request for a single record:

```
content://com.paad.skeletondatabaseprovider/elements/5
```

It's good practice to support access to your provider for both of these forms. The simplest way to do this is to use a `UriMatcher`, a useful class that parses URIs and determines their forms.

Listing 8.8 shows the implementation pattern for defining a URI Matcher that analyzes the form of a URI—specifically determining if a URI is a request for all data or for a single row.

Listing 8.8: Defining a UriMatcher to determine if a request is for all elements or a single row

```
// Create the constants used to differentiate
between the different URI
// requests.
private static final int ALLROWS = 1;
private static final int SINGLE_ROW = 2;

private static final UriMatcher uriMatcher;

// Populate the UriMatcher object, where a URI
ending in
// 'elements' will correspond to a request for all
items,
// and 'elements/[rowID]' represents a single row.
static {
  uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
  uriMatcher.addURI("com.paad.skeletondatabaseprovider",
                    "elements", ALLROWS);
  uriMatcher.addURI("com.paad.skeletondatabaseprovider",
                    "elements/#", SINGLE_ROW);
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyContentProvider.java

You can use the same technique to expose alternative URIs within the same Content Provider that represent different subsets of data, or different tables within your database.

Having distinguished between full table and single row queries, you can use the `SQLiteQueryBuilder` class to easily apply the

additional selection condition to a query, as shown in the following snippet:

```
SQLiteQueryBuilder queryBuilder = new
SQLiteQueryBuilder();

// If this is a row query, limit the result
set to the passed in row.
switch (uriMatcher.match(uri)) {
  case SINGLE_ROW :
    String rowID =
uri.getPathSegments().get(1);
    queryBuilder.appendWhere(KEY_ID + "=" +
rowID);
  default: break;
}
```

You'll learn how to perform a query using the SQLite Query Builder later in the “Implementing Content Provider Queries” section.

Creating the Content Provider's Database

To initialize the data source you plan to access through the Content Provider, override the `onCreate` method, as shown in Listing 8.9. This is typically handled using an SQLite Open Helper implementation, of the type described in the previous section, allowing you to effectively defer creating and opening the database until it's required.

Listing 8.9: Creating the Content Provider's database

```
private MySQLiteOpenHelper myOpenHelper;

@Override
public boolean onCreate() {
  // Construct the underlying database.
  // Defer opening the database until you need to perform
  // a query or transaction.
  myOpenHelper = new MySQLiteOpenHelper(getContext(),
      MySQLiteOpenHelper.DATABASE_NAME, null,
      MySQLiteOpenHelper.DATABASE_VERSION);

  return true;
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyContentProvider.java

When your application is launched, the `onCreate` handler of each of its Content Providers is called on the main application thread.

Like the database examples in the previous section, it's best practice to use an SQLite Open Helper to defer opening (and where necessary, creating) the underlying database until it is required within the query or transaction methods of your Content Provider.

For efficiency reasons, it's preferable to leave your Content Provider open while your application is running; it's not necessary to manually close the database at any stage. If the system requires additional resources, your application will be killed and the associated databases closed.

Implementing Content Provider Queries

To support queries with your Content Provider, you must implement the `query` and `getType` methods. Content Resolvers use these methods to access the underlying data, without knowing its structure or implementation. These methods enable applications to share data across application boundaries without having to publish a specific interface for each data source.

The most common scenario is to use a Content Provider to provide access to an SQLite database, but within these methods you can access any source of data (including files or application instance variables).

Notice that the `UriMatcher` object is used to refine the transaction and query requests, and the SQLite Query Builder is used as a convenient helper for performing row-based queries.

Listing 8.10 shows the skeleton code for implementing queries within a Content Provider using an underlying SQLite database.

Listing 8.10: Implementing queries and transactions within a Content Provider

```
@Override
public Cursor query(Uri uri, String[] projection, String
selection,
  String[] selectionArgs, String sortOrder) {

  // Open the database.
  SQLiteDatabase db;
  try {
    db = myOpenHelper.getWritableDatabase();
  } catch (SQLiteException ex) {
    db = myOpenHelper.getReadableDatabase();
  }

  // Replace these with valid SQL statements if necessary.
  String groupBy = null;
  String having = null;

  // Use an SQLite Query Builder to simplify constructing the
  // database query.
  SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();

  // If this is a row query, limit the result set to the
passed in row.
  switch (uriMatcher.match(uri)) {
    case SINGLE_ROW :
      String rowID = uri.getPathSegments().get(1);
      queryBuilder.appendWhere(KEY_ID + "=" + rowID);
    default: break;
  }

  // Specify the table on which to perform the query. This can
  // be a specific table or a join as required.
  queryBuilder.setTables(MySQLiteOpenHelper.DATABASE_TABLE);

  // Execute the query.
  Cursor cursor = queryBuilder.query(db, projection,
selection,
      selectionArgs, groupBy, having, sortOrder);

  // Return the result Cursor.
  return cursor;
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyContentProvider.java

Having implemented queries, you must also specify a MIME type to identify the data returned. Override the `getType` method to return a string that uniquely describes your data type.

The type returned should include two forms, one for a single entry and another for all the entries, following these forms:

- Single item:

```
vnd.android.cursor.item/vnd.<companyname>.<contenttype>
```

- All items:

```
vnd.android.cursor.dir/vnd.<companyname>.<contenttype>
```

Listing 8.11 shows how to override the `getType` method to return the correct MIME type based on the URI passed in.

Listing 8.11: Returning a Content Provider MIME type

```
@Override
public String getType(Uri uri) {
  // Return a string that identifies the MIME
type
  // for a Content Provider URI
```

```
  switch (uriMatcher.match(uri)) {
    case ALLROWS:
      return
"vnd.android.cursor.dir/vnd.paad.elemental";
    case SINGLE_ROW:
      return
"vnd.android.cursor.item/vnd.paad.elemental";
    default:
      throw new
IllegalArgumentException("Unsupported URI: " +

uri);
  }
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyContentProvider.java

Content Provider Transactions

To expose delete, insert, and update transactions on your Content Provider, implement the corresponding `delete`, `insert`, and `update` methods.

Like the `query` method, these methods are used by Content Resolvers to perform transactions on the underlying data without knowing its implementation—allowing applications to update data across application boundaries.

When performing transactions that modify the dataset, it's good practice to call the Content Resolver's `notifyChange` method. This will notify any Content Observers, registered for a given Cursor using the `Cursor.registerContentObserver` method, that the underlying table (or a particular row) has been removed, added, or updated.

As with Content Provider queries, the most common use case is performing transactions on an SQLite database, though this is not a requirement. Listing 8.12 shows the skeleton code for implementing transactions within a Content Provider on an underlying SQLite database.

Listing 8.12: Typical Content Provider transaction implementations

```
@Override
public int delete(Uri uri, String selection, String[]
selectionArgs) {
  // Open a read / write database to support the transaction.
  SQLiteDatabase db = myOpenHelper.getWritableDatabase();

  // If this is a row URI, limit the deletion to the specified
row.
  switch (uriMatcher.match(uri)) {
    case SINGLE_ROW :
      String rowID = uri.getPathSegments().get(1);
      selection = KEY_ID + "=" + rowID
          + (!TextUtils.isEmpty(selection) ?
            " AND (" + selection + ')' : "");
    default: break;
  }

  // To return the number of deleted items you must specify a
where
  // clause. To delete all rows and return a value pass in
"1".
  if (selection == null)
    selection = "1";

  // Perform the deletion.
  int deleteCount =
db.delete(MySQLiteOpenHelper.DATABASE_TABLE,
    selection, selectionArgs);

  // Notify any observers of the change in the data set.
  getContext().getContentResolver().notifyChange(uri, null);

  // Return the number of deleted items.
  return deleteCount;
}
```

```
@Override
public Uri insert(Uri uri, ContentValues values) {
  // Open a read / write database to support the transaction.
  SQLiteDatabase db = myOpenHelper.getWritableDatabase();

  // To add empty rows to your database by passing in an empty
  // Content Values object you must use the null column hack
  // parameter to specify the name of the column that can be
  // set to null.
  String nullColumnHack = null;

  // Insert the values into the table
  long id = db.insert(MySQLiteOpenHelper.DATABASE_TABLE,
      nullColumnHack, values);

  // Construct and return the URI of the newly inserted row.
  if (id > -1) {
    // Construct and return the URI of the newly inserted row.
    Uri insertedId = ContentUris.withAppendedId(CONTENT_URI,
id);

    // Notify any observers of the change in the data set.
    getContext().getContentResolver().notifyChange(insertedId,
null);

    return insertedId;
  }
  else
    return null;
}

@Override
public int update(Uri uri, ContentValues values, String
selection,
  String[] selectionArgs) {

  // Open a read / write database to support the transaction.
  SQLiteDatabase db = myOpenHelper.getWritableDatabase();

  // If this is a row URI, limit the deletion to the specified
row.
  switch (uriMatcher.match(uri)) {
    case SINGLE_ROW :
      String rowID = uri.getPathSegments().get(1);
```

```
      selection = KEY_ID + "=" + rowID
          + (!TextUtils.isEmpty(selection) ?
            " AND (" + selection + ')' : "");
    default: break;
  }

  // Perform the update.
  int updateCount =
db.update(MySQLiteOpenHelper.DATABASE_TABLE,
    values, selection, selectionArgs);

  // Notify any observers of the change in the data set.
  getContext().getContentResolver().notifyChange(uri, null);

  return updateCount;
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyContentProvider.java

When working with content URIs, the `ContentUris` class includes the `withAppendedId` convenience method to easily append a specific row ID to the `CONTENT_URI` of a Content Provider. This is used in Listing 8.12 to construct the URI of newly insert rows and will be used in the following sections to address a particular row when making database queries and transactions.

Storing Files in a Content Provider

Rather than store large files within your Content Provider, you should represent them within a table as fully qualified URIs to a file stored somewhere else on the filesystem.

To support files within your table, you must include a column labeled `_data` that will contain the path to the file represented by that record. This column should not be used by client applications. Override the `openFile` handler to provide a `ParcelFileDescriptor` when the Content Resolver requests the file associated with that record.

It's typical for a Content Provider to include two tables, one that is used only to store the external files, and another that includes a user-facing column containing a URI reference to the rows in the file table.

Listing 8.13 shows the skeleton code for overriding the `openFile` handler within a Content Provider. In this instance, the name of the file will be represented by the ID of the row to which it belongs.

Listing 8.13: Storing files within your Content Provider

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
  throws FileNotFoundException {

  // Find the row ID and use it as a filename.
  String rowID = uri.getPathSegments().get(1);

  // Create a file object in the application's external
  // files directory.
  String picsDir = Environment.DIRECTORY_PICTURES;
  File file =
    new File(getContext().getExternalFilesDir(picsDir),
rowID);

  // If the file doesn't exist, create it now.
  if (!file.exists()) {
    try {
      file.createNewFile();
    } catch (IOException e) {
      Log.d(TAG, "File creation failed: " + e.getMessage());
    }
  }

  // Translate the mode parameter to the corresponding Parcel
File
  // Descriptor open mode.
  int fileMode = 0;
  if (mode.contains("w"))
    fileMode |= ParcelFileDescriptor.MODE_WRITE_ONLY;
  if (mode.contains("r"))
    fileMode |= ParcelFileDescriptor.MODE_READ_ONLY;
  if (mode.contains("+"))
    fileMode |= ParcelFileDescriptor.MODE_APPEND;

  // Return a Parcel File Descriptor that represents the file.
  return ParcelFileDescriptor.open(file, fileMode);
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyHoardContentProvider.java

Because the files associated with rows in the database are stored externally, it's important to consider what the effect of deleting a row should have on the underlying file.

A Skeleton Content Provider Implementation

Listing 8.14 shows a skeleton implementation of a Content Provider. It uses an SQLite Open Helper to manage the database, and simply passes each query or transaction directly to the underlying SQLite database.

Listing 8.14: A skeleton Content Provider implementation

```
import android.content.ContentProvider;
import android.content.ContentUris;
import android.content.ContentValues;
import android.content.Context;
import android.content.UriMatcher;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteDatabase.CursorFactory;
import android.database.sqlite.SQLiteOpenHelper;
import android.database.sqlite.SQLiteQueryBuilder;
import android.net.Uri;
import android.text.TextUtils;
import android.util.Log;

public class MyContentProvider extends ContentProvider {

  public static final Uri CONTENT_URI =

 Uri.parse("content://com.paad.skeletondatabaseprovider/elements");

  // Create the constants used to differentiate between
  // the different URI requests.
```

```
  private static final int ALLROWS = 1;
  private static final int SINGLE_ROW = 2;

  private static final UriMatcher uriMatcher;

  // Populate the UriMatcher object, where a URI ending
  // in 'elements' will correspond to a request for all
  // items, and 'elements/[rowID]' represents a single row.
  static {
   uriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
   uriMatcher.addURI("com.paad.skeletondatabaseprovider",
     "elements", ALLROWS);
   uriMatcher.addURI("com.paad.skeletondatabaseprovider",
     "elements/#", SINGLE_ROW);
  }

  // The index (key) column name for use in where clauses.
  public static final String KEY_ID = "_id";

  // The name and column index of each column in your
database.
  // These should be descriptive.
  public static final String KEY_COLUMN_1_NAME =
"KEY_COLUMN_1_NAME";
  // TODO: Create public field for each column in your table.

  // SQLite Open Helper variable
  private MySQLiteOpenHelper myOpenHelper;

  @Override
  public boolean onCreate() {
    // Construct the underlying database.
    // Defer opening the database until you need to perform
    // a query or transaction.
    myOpenHelper = new MySQLiteOpenHelper(getContext(),
        MySQLiteOpenHelper.DATABASE_NAME, null,
        MySQLiteOpenHelper.DATABASE_VERSION);

    return true;
  }

  @Override
  public Cursor query(Uri uri, String[] projection, String
selection,
      String[] selectionArgs, String sortOrder) {
    // Open the database.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();
```

```
    // Replace these with valid SQL statements if necessary.
    String groupBy = null;
    String having = null;

    SQLiteQueryBuilder queryBuilder = new
SQLiteQueryBuilder();
    queryBuilder.setTables(MySQLiteOpenHelper.DATABASE_TABLE);

    // If this is a row query, limit the result set to the
    // passed in row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        queryBuilder.appendWhere(KEY_ID + "=" + rowID);
      default: break;
    }

    Cursor cursor = queryBuilder.query(db, projection,
selection,
        selectionArgs, groupBy, having, sortOrder);

    return cursor;
  }

  @Override
  public int delete(Uri uri, String selection, String[]
selectionArgs)
  {
    // Open a read / write database to support the
transaction.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();

    // If this is a row URI, limit the deletion to the
specified row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        selection = KEY_ID + "=" + rowID
            + (!TextUtils.isEmpty(selection) ?
              " AND (" + selection + ')' : "");
      default: break;
    }

    // To return the number of deleted items, you must specify
a where
    // clause. To delete all rows and return a value, pass in
```

```
"1".
    if (selection == null)
      selection = "1";

    // Execute the deletion.
    int deleteCount =
db.delete(MySQLiteOpenHelper.DATABASE_TABLE,
      selection, selectionArgs);

    // Notify any observers of the change in the data set.
    getContext().getContentResolver().notifyChange(uri, null);

    return deleteCount;
  }


  @Override
  public Uri insert(Uri uri, ContentValues values) {
    // Open a read / write database to support the
transaction.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();

    // To add empty rows to your database by passing in an
empty
    // Content Values object, you must use the null column
hack
    // parameter to specify the name of the column that can be
    // set to null.
    String nullColumnHack = null;

    // Insert the values into the table
    long id = db.insert(MySQLiteOpenHelper.DATABASE_TABLE,
        nullColumnHack, values);

    if (id > -1) {
      // Construct and return the URI of the newly inserted
row.
      Uri insertedId = ContentUris.withAppendedId(CONTENT_URI,
id);

      // Notify any observers of the change in the data set.

 getContext().getContentResolver().notifyChange(insertedId,
null);

      return insertedId;
    }
```

```
    else
      return null;
  }

  @Override
  public int update(Uri uri, ContentValues values, String
selection,
    String[] selectionArgs) {

    // Open a read / write database to support the
transaction.
    SQLiteDatabase db = myOpenHelper.getWritableDatabase();

    // If this is a row URI, limit the deletion to the
specified row.
    switch (uriMatcher.match(uri)) {
      case SINGLE_ROW :
        String rowID = uri.getPathSegments().get(1);
        selection = KEY_ID + "=" + rowID
            + (!TextUtils.isEmpty(selection) ?
              " AND (" + selection + ')' : "");
      default: break;
    }

    // Perform the update.
    int updateCount =
db.update(MySQLiteOpenHelper.DATABASE_TABLE,
      values, selection, selectionArgs);

    // Notify any observers of the change in the data set.
    getContext().getContentResolver().notifyChange(uri, null);

    return updateCount;
  }

  @Override
  public String getType(Uri uri) {
    // Return a string that identifies the MIME type
    // for a Content Provider URI
    switch (uriMatcher.match(uri)) {
      case ALLROWS:
        return "vnd.android.cursor.dir/vnd.paad.elemental";
      case SINGLE_ROW:
        return "vnd.android.cursor.item/vnd.paad.elemental";
      default:
        throw new IllegalArgumentException("Unsupported URI: "
+ uri);
```

```
    }
  }

  private static class MySQLiteOpenHelper extends
SQLiteOpenHelper {
    // [ ... SQLite Open Helper Implementation ... ]
  }
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/MyContentProvider.java

Using Content Providers

The following sections introduce the `ContentResolver` class and how to use it to query and transact with a Content Provider.

Introducing the Content Resolver

Each application includes a `ContentResolver` instance, accessible using the `getContentResolver` method, as follows:

```
ContentResolver cr = getContentResolver();
```

When Content Providers are used to expose data, Content Resolvers are the corresponding class used to query and perform transactions on those Content Providers. Whereas Content Providers provide an abstraction from the underlying data, Content Resolvers provide an abstraction from the Content Provider being queried or transacted.

The Content Resolver includes query and transaction methods corresponding to those defined within your Content Providers. The Content Resolver does not need to know the implementation of the Content Providers it is interacting with—each query and transaction method simply accepts a URI that specifies the Content Provider to interact with.

A Content Provider's URI is its *authority* as defined by its manifest node and typically published as a static constant on the Content Provider implementation.

Content Providers usually accept two forms of URI, one for requests against all data and another that specifies only a single row. The form for the latter

appends the row identifier (in the form `/<rowID>`) to the base URI.

Querying Content Providers

Content Provider queries take a form very similar to that of database queries. Query results are returned as Cursors over a result set in the same way as described previously in this chapter for databases.

You can extract values from the result Cursor using the same techniques described in the section "Extracting Results from a Cursor."

Using the `query` method on the `ContentResolver` object, pass in the following:

- A URI to the Content Provider you want to query.
- A projection that lists the columns you want to include in the result set.
- A `where` clause that defines the rows to be returned. You can include `?` wildcards that will be replaced by the values passed into the selection argument parameter.
- An array of selection argument strings that will replace the `?` wildcards in the `where` clause.
- A string that describes the order of the returned rows.

Listing 8.15 shows how to use a Content Resolver to apply a query to a Content Provider.

Listing 8.15: Querying a Content Provider with a Content Resolver

```
// Get the Content Resolver.
ContentResolver cr = getContentResolver();

// Specify the result column projection. Return the minimum
set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
    MyHoardContentProvider.KEY_ID,
    MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
    MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN };

// Specify the where clause that will limit your results.
String where =
MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN
               + "=" + 1;

// Replace these with valid SQL statements as necessary.
String whereArgs[] = null;
String order = null;

// Return the specified rows.
Cursor resultCursor =
cr.query(MyHoardContentProvider.CONTENT_URI,
  result_columns, where, whereArgs, order);
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

In this example the query is made using static constants provided by the `MyHoardContentProvider` class; however, it's worth noting that a third-party application can perform the same query, provided it knows the content URI and column names, and has the appropriate permissions.

Most Content Providers also include a shortcut URI pattern that allows you to address a particular row by appending a row ID to the content URI. You can use the static `withAppendedId` method from the `ContentUris` class to simplify this, as shown in Listing 8.16.

Listing 8.16: Querying a Content Provider for a particular row

```
// Get the Content Resolver.
ContentResolver cr = getContentResolver();

// Specify the result column projection. Return the
minimum set
// of columns required to satisfy your requirements.
String[] result_columns = new String[] {
    MyHoardContentProvider.KEY_ID,
    MyHoardContentProvider.KEY_GOLD_HOARD_NAME_COLUMN,
    MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN };

// Append a row ID to the URI to address a specific row.
Uri rowAddress =
  ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI,
  rowId);

// These are null as we are requesting a single row.
String where = null;
String whereArgs[] = null;
String order = null;

// Return the specified rows.
Cursor resultCursor = cr.query(rowAddress,
  result_columns, where, whereArgs, order);
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

To extract values from a result Cursor, use the same techniques described earlier in this chapter, using the `moveTo<location>` methods in combination with the `get<type>` methods to

extract values from the specified row and column.

Listing 8.17 extends the code from Listing 8.15, by iterating over a result Cursor and displaying the name of the largest hoard.

Listing 8.17: Extracting values from a Content Provider result Cursor

```
float largestHoard = 0f;
String hoardName = "No Hoards";

// Find the index to the column(s) being used.
int GOLD_HOARDED_COLUMN_INDEX =
resultCursor.getColumnIndexOrThrow(
  MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN);
int HOARD_NAME_COLUMN_INDEX =
resultCursor.getColumnIndexOrThrow(
  MyHoardContentProvider.KEY_GOLD_HOARD_NAME_COLUMN);

// Iterate over the cursors rows.
// The Cursor is initialized at before first, so we
can
// check only if there is a "next" row available. If
the
// result Cursor is empty, this will return false.
while (resultCursor.moveToNext()) {
  float hoard =
resultCursor.getFloat(GOLD_HOARDED_COLUMN_INDEX);
  if (hoard > largestHoard) {
    largestHoard = hoard;
    hoardName =
resultCursor.getString(HOARD_NAME_COLUMN_INDEX);
  }
}

// Close the Cursor when you've finished with it.
resultCursor.close();
```

*code snippet PA4AD_*

Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

When you have finished using your result Cursor it's important to close it to avoid memory leaks and reduce your application's resource load.

```
resultCursor.close();
```

You'll see more examples of querying for content later in this chapter when the native Android Content Providers are introduced.

Database queries can take significant time to execute. By default, the Content Resolver will execute queries—as well as other transactions—on the main application thread.

To ensure your application remains smooth and responsive, you must execute all queries asynchronously, as described in the following section.

Querying for Content Asynchronously Using the Cursor Loader

Database operations can be time-consuming, so it's particularly important that any database and Content Provider queries are not performed on the main application thread.

It can be difficult to manage Cursors, synchronize correctly with the UI thread, and ensure all queries occur on a background. To help simplify the process, Android 3.0 (API level 11) introduced the `Loader` class. Loaders are now also available within the Android Support Library, making them available for use with every Android platform back to Android 1.6.

Introducing Loaders

Loaders are available within every Activity and Fragment via the `LoaderManager`. They are designed to asynchronously load data and monitor the underlying data source for changes.

While loaders can be implemented to load any kind of data from any data source, of particular interest is the `CursorLoader` class. The Cursor Loader allows you to perform asynchronous queries against Content Providers,

returning a result Cursor and notifications of any updates to the underlying provider.

To maintain concise and encapsulated code, not all the examples in this chapter utilize a Cursor Loader when making a Content Provider query. For your own applications it's best practice to always use a Cursor Loader to manage Cursors within your Activities and Fragments.

Using the Cursor Loader

The Cursor Loader handles all the management tasks required to use a Cursor within an Activity or Fragment, effectively deprecating the `managedQuery` and `startManagingCursor` Activity methods. This includes managing the Cursor lifecycle to ensure Cursors are closed when the Activity is terminated.

Cursor Loaders also observe changes in the underlying query, so you no longer need to implement your own Content Observers.

Implementing Cursor Loader Callbacks

To use a Cursor Loader, create a new `LoaderManager.LoaderCallbacks` implementation. Loader Callbacks are implemented using generics, so you should

specify the explicit type being loaded, in this case Cursors, when implementing your own.

```
LoaderManager.LoaderCallbacks<Cursor> loaderCallback
  = new LoaderManager.LoaderCallbacks<Cursor>() {
```

If you require only a single Loader implementation within your Fragment or Activity, this is typically done by having that component implement the interface.

The Loader Callbacks consist of three handlers:

- `onCreateLoader`—Called when the loader is initialized, this handler should create and return new Cursor Loader object. The Cursor Loader constructor arguments mirror those required for executing a query using the Content Resolver. Accordingly, when this handler is executed, the query parameters you specify will be used to perform a query using the Content Resolver.
- `onLoadFinished`—When the Loader Manager has completed the asynchronous query, the `onLoadFinished` handler is called, with the result Cursor passed in as a parameter. Use this Cursor to update adapters and other UI elements.
- `onLoaderReset`—When the Loader Manager resets your Cursor Loader, `onLoaderReset` is called. Within this handler you should release any references to data returned by the query and reset the UI accordingly. The Cursor will be

closed by the Loader Manager, so you shouldn't attempt to close it.

The `onLoadFinished` and `onLoaderReset` are *not* synchronized to the UI thread. If you want to modify UI elements directly, you will first need to synchronize with the UI thread using a Handler or similar mechanism. Synchronizing with the UI thread is covered in more details in Chapter 9, "Working in the Background."

Listing 8.18 show a skeleton implementation of the Cursor Loader Callbacks.

Listing 8.18: Implementing Loader Callbacks

```
public Loader<Cursor> onCreateLoader(int id, Bundle args)
{
  // Construct the new query in the form of a Cursor
Loader. Use the id
  // parameter to construct and return different loaders.
  String[] projection = null;
  String where = null;
  String[] whereArgs = null;
  String sortOrder = null;

  // Query URI
  Uri queryUri = MyContentProvider.CONTENT_URI;

  // Create the new Cursor loader.
  return new CursorLoader(DatabaseSkeletonActivity.this,
queryUri,
```

```
    projection, where, whereArgs, sortOrder);
}

public void onLoadFinished(Loader<Cursor> loader, Cursor
cursor) {
  // Replace the result Cursor displayed by the Cursor
Adapter with
  // the new result set.
  adapter.swapCursor(cursor);

  // This handler is not synchronized with the UI thread,
so you
  // will need to synchronize it before modifying any UI
elements
  // directly.
}

public void onLoaderReset(Loader<Cursor> loader) {
  // Remove the existing result Cursor from the List
Adapter.
  adapter.swapCursor(null);

  // This handler is not synchronized with the UI thread,
so you
  // will need to synchronize it before modifying any UI
elements
  // directly.
}
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

Initializing and Restarting the Cursor Loader

Each Activity and Fragment provides access to its Loader Manager through a call to `getLoaderManager`.

```
LoaderManager loaderManager = getLoaderManager();
```

To initialize a new Loader, call the Loader Manager's `initLoader` method, passing in a reference to your Loader Callback implementation, an optional arguments Bundle, and a loader identifier.

```
Bundle args = null;
loaderManager.initLoader(LOADER_ID, args,
myLoaderCallbacks);
```

This is generally done within the `onCreate` method of the host Activity (or the `onActivityCreated` handler in the case of Fragments).

If a loader corresponding to the identifier used doesn't already exist, it is created within the associated Loader Callback's `onCreateLoader` handler as described in the previous section.

In most circumstances this is all that is required. The Loader Manager will handle the lifecycle of any Loaders you initialize and the underlying queries and cursors. Similarly, it will manage changes to the query results.

After a Loader has been created, repeated calls to `initLoader` will simply return the existing Loader. Should you want to discard the previous Loader and re-create it, use the `restartLoader`

method.

```
loaderManager.restartLoader(LOADER_ID,
args, myLoaderCallbacks);
```

This is typically necessary where your query parameters change, such as search queries or changes in sort order.

Adding, Deleting, and Updating Content

To perform transactions on Content Providers, use the `insert`, `delete`, and `update` methods on the Content Resolver. Like queries, unless moved to a worker thread, Content Provider transactions will execute on the main application thread.

Database operations can be time-consuming, so it's important to execute each transaction asynchronously.

Inserting Content

The Content Resolver offers two methods for inserting new records into a Content Provider: `insert` and `bulkInsert`. Both methods accept the URI of the Content Provider into which you're inserting; the `insert` method takes a single new `ContentValues` object, and the `bulkInsert` method takes an array.

The `insert` method returns a URI to the newly added record, whereas the `bulkInsert` method returns the number of successfully added rows.

Listing 8.19 shows how to use the `insert` method to add new rows to a Content Provider.

Listing 8.19: Inserting new rows into a Content Provider

```
// Create a new row of values to insert.
ContentValues newValues = new ContentValues();

// Assign values for each row.
newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_NAME_COLUMN,
              hoardName);
newValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN,
              hoardValue);
newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
              hoardAccessible);
// [ ... Repeat for each column / value pair ... ]

// Get the Content Resolver
ContentResolver cr = getContentResolver();

// Insert the row into your table
Uri myRowUri = cr.insert(MyHoardContentProvider.CONTENT_URI,
                         newValues);
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

Deleting Content

To delete a single record, call `delete` on the Content Resolver, passing in the URI of the row you want to remove. Alternatively, you can specify a `where` clause to remove multiple rows. Listing 8.20 demonstrates how to delete a number of rows matching a given condition.

Listing 8.20: Deleting rows from a Content Provider

```
// Specify a where clause that determines which row(s) to
delete.
// Specify where arguments as necessary.
String where =
MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN +
                "=" + 0;
String whereArgs[] = null;

// Get the Content Resolver.
ContentResolver cr = getContentResolver();

// Delete the matching rows
int deletedRowCount =
  cr.delete(MyHoardContentProvider.CONTENT_URI, where,
whereArgs);
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

Updating Content

You can update rows by using the Content Resolver's `update` method. The `update` method takes the URI of the target Content Provider, a `ContentValues` object that maps column names to updated values, and a `where` clause that indicates which rows to update.

When the update is executed, every row matched by the `where` clause is updated using the specified Content Values, and the number of successful updates is returned.

Alternatively, you can choose to update a specific row by specifying its unique URI, as shown in Listing 8.21.

Listing 8.21: Updating a record in a Content Provider

```
// Create the updated row content, assigning values
for each row.
ContentValues updatedValues = new ContentValues();
```

```
updatedValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN,
                  newHoardValue);
// [ ... Repeat for each column to update ... ]

// Create a URI addressing a specific row.
Uri rowURI =
  ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI,
  hoardId);

// Specify a specific row so no selection clause is
required.
String where = null;
String whereArgs[] = null;

// Get the Content Resolver.
ContentResolver cr = getContentResolver();

// Update the specified row.
int updatedRowCount =
  cr.update(rowURI, updatedValues, where,
whereArgs);
```

code snippet PA4AD_Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

Accessing Files Stored in Content Providers

Content Providers represent large files as fully qualified URIs rather than raw file blobs; however, this is abstracted away when using the Content Resolver.

To access a file stored in, or to insert a new file into, a Content Provider, simply use the Content Resolver's `openOutputStream` or `openInputStream` methods, respectively, passing in the URI to the Content Provider row containing the file you require. The Content Provider will interpret your request and return an input or output stream to the requested file, as shown in Listing 8.22.

Listing 8.22: Reading and writing files from and to a Content Provider

```
public void addNewHoardWithImage(String hoardName, float
hoardValue,
  boolean hoardAccessible, Bitmap bitmap) {

  // Create a new row of values to insert.
  ContentValues newValues = new ContentValues();

  // Assign values for each row.
  newValues.put(MyHoardContentProvider.KEY_GOLD_HOARD_NAME_COLUMN,
                hoardName);
  newValues.put(MyHoardContentProvider.KEY_GOLD_HOARDED_COLUMN,
                hoardValue);
```

```
  newValues.put(
    MyHoardContentProvider.KEY_GOLD_HOARD_ACCESSIBLE_COLUMN,
    hoardAccessible);

  // Get the Content Resolver
  ContentResolver cr = getContentResolver();

  // Insert the row into your table
  Uri myRowUri =
    cr.insert(MyHoardContentProvider.CONTENT_URI, newValues);

  try {
    // Open an output stream using the new row's URI.
    OutputStream outStream = cr.openOutputStream(myRowUri);
    // Compress your bitmap and save it into your provider.
    bitmap.compress(Bitmap.CompressFormat.JPEG, 80,
outStream);
  }
  catch (FileNotFoundException e) {
    Log.d(TAG, "No file found for this record.");
  }
}

public Bitmap getHoardImage(long rowId) {
  Uri myRowUri =

 ContentUris.withAppendedId(MyHoardContentProvider.CONTENT_URI,
                            rowId);

  try {
    // Open an input stream using the new row's URI.
    InputStream inStream =
      getContentResolver().openInputStream(myRowUri);

    // Make a copy of the Bitmap.
    Bitmap bitmap = BitmapFactory.decodeStream(inStream);
    return bitmap;
  }
    catch (FileNotFoundException e) {
    Log.d(TAG, "No file found for this record.");
  }

  return null;
}
```

*code snippet PA4AD_*

Ch08_DatabaseSkeleton/src/DatabaseSkeletonActivity.java

Creating a To-Do List Database and Content Provider

In Chapter 4, “Building User Interfaces,” you created a To-Do List application. In the following example, you'll create a database and Content Provider to save each of the to-do items added to the list.

1. Start by creating a new