Android Tutorial

MCA 5th Sem. GTU

By : Ketan Bhimani

An Overview of Android

RKUniversity™

By : Ketan Bhimani

Introducing Android

The mobile development community is at a tipping point. Mobile users demand more choice, more opportunities to customize their phones, and more functionality. Mobile operators want to provide value-added content to their subscribers in a manageable and lucrative way. Mobile developers want the freedom to develop the powerful mobile applications users demand with minimal roadblocks to success. Finally, handset manufacturers want a stable, secure, and affordable platform to power their devices. Up until now a single mobile platform has adequately addressed the needs of all the parties.

Enter Android, which is a potential game-changer for the mobile development community. An innovative and open platform, Android is well positioned to address the growing needs of the mobile marketplace.

This chapter explains what Android is, how and why it was developed, and where the platform fits in to the established mobile marketplace.

A Brief History of Mobile Software Development

To understand what makes Android so compelling, we must examine how mobile develop- ment has evolved and how Android differs from competing platforms.

Way Back When

Remember way back when a phone was just a phone? When we relied on fixed land- lines? When we ran for the phone instead of pulling it out of our pocket? When we lost our friends at a crowded ballgame and waited around for hours hoping to reunite? When we forgot the grocery list and had to find a payphone or drive back home again?

Those days are long gone. Today, commonplace problems such as these are easily solved with a one-button speed dial or a simple text message like "WRU?" or "20?" or "Milk and?"

Our mobile phones keep us safe and connected. Now we roam around freely, relying on our phones not only to keep in touch with friends, family, and coworkers, but also to tell us where to go, what to do, and how to do it. Even the most domestic of events seem to revolve around my mobile phone.

Mobile phones have become a crucial shopping accessory.

Consider the following true story, which has been slightly enhanced for effect:

Once upon a time, on a warm summer evening, I was happily minding my own business cooking dinner in my new house in rural New Hampshire when a bat swooped over my head, scaring me to death.

The first thing I did—while ducking—was to pull out my cell phone and send a text message to my husband, who was across the country at the time. I typed, "There's a bat in the house!"

My husband did not immediately respond (a divorce-worthy incident, I thought at the time), so I called my dad and asked him for suggestions on how to get rid of the bat.

He just laughed.

Annoyed, I snapped a picture of the bat with my phone and sent it to my husband and my blog, simultaneously guilt-tripping him and informing the world of my treacherous domestic wildlife encounter.

Finally, I googled "get rid of a bat" and then I followed the helpful do-it-yourself instructions provided on the Web for people in my situation. I also learned that late August is when baby bats often leave the roost for the first time and learn to fly. Newly aware that I had a baby bat on my hands, I calmly got a broom and managed to herd the bat out of the house.

Problem solved—and I did it all with the help of my trusty cell phone, the old LG VX9800.

My point here? Mobile phones can solve just about *anything*—and we rely on them for *everything* these days.

You notice that I used half a dozen different mobile applications over the course of this story. Each application was developed by a different company and had a different user interface. Some were well designed; others not so much. I paid for some of the applications, and others came on my phone.

As a user, I found the experience functional, but not terribly inspiring. As a mobile developer, I wished for an opportunity to create a more seamless and powerful application that could handle all I'd done and more. I wanted to build a better bat trap, if you will.

Before Android, mobile developers faced many roadblocks when it came to writing applications. Building the better application, the unique application, the competing application, the hybrid application, and incorporating many common tasks such as messaging and calling in a familiar way were often unrealistic goals.

To understand why, let's take a brief look at the history of mobile software development.

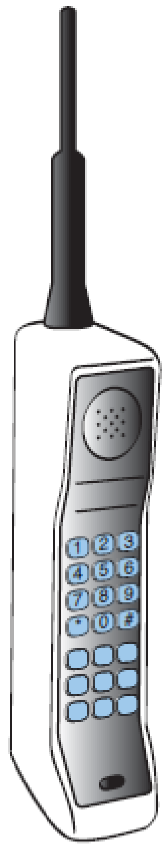
"The Brick"

The Motorola DynaTAC 8000X was the first commercially available cell phone. First marketed in 1983, it was 13 × 1.75 × 3.5 inches in dimension, weighed about 2.5 pounds, and allowed you to talk for a little more than half an hour. It retailed for $3,995, plus hefty monthly service fees and per-minute charges.

We called it "The Brick," and the nickname stuck for many of those early mobile phones we alternatively loved and hated. About the size of a brick, with a battery power just long enough for half a conversation, these early mobile handsets were mostly seen in the hands of traveling business execs, security personnel, and the wealthy. First-generation mobile phones were just too expensive. The service charges alone would bankrupt the average person, especially when roaming.

Early mobile phones were not particularly full featured. (Although, even the Motorola DynaTAC, shown in Figure, had many of the buttons we've come to know well, such as the SEND, END, and CLR buttons.) These early phones did little more than make and receive calls and, if you were lucky, there was a simple contacts application that wasn't impossible to use.

The first commercially available mobile phone: the Motorola DynaTAC.



The first-generation mobile phones were designed and developed by the handset manufacturers. Competition was fierce and trade secrets were closely guarded. Manufacturers didn't want to expose the internal workings of their handsets, so they usually developed the phone software in-house. As a developer, if you weren't part of this inner circle, you had no opportunity to write applications for the phones.

It was during this period that we saw the first "time-waster" games begin to appear. Nokia was famous for putting the 1970s video game Snake on some of its earliest monochrome phones. Other manufacturers followed suit, adding games such as Pong, Tetris, and Tic-Tac-Toe.

These early phones were flawed, but they did something important—they changed the way people thought about communication. As mobile phone prices dropped, batteries improved, and reception areas grew, more and more people began carrying these handy devices. Soon mobile phones were more than just a novelty.

Customers began pushing for more features and more games. But there was a problem. The handset manufacturers didn't have the motivation or the resources to build every application users wanted. They needed some way to provide a portal for entertainment and information services without allowing direct access to the handset.

What better way to provide these services than the Internet?

Wireless Application Protocol (WAP)

As it turned out, allowing direct phone access to the Internet didn't scale well for mobile.

By this time, professional websites were full color and chock full of text, images, and other sorts of media. These sites relied on JavaScript, Flash, and other technologies to enhance the user experience, and they were often designed with a target resolution of 800x600 pixels and higher.

When the first clamshell phone, the Motorola StarTAC, was released in 1996, it merely had an LCD 10-digit segmented display. (Later models would add a dot-matrix type display.) Meanwhile, Nokia released one of the first slider phones, the 8110—fondly referred to as "The Matrix Phone" because the phone was heavily used in films. The 8110 could display four lines of text with 13 characters per line. This shows some of the common phone form factors.

Various mobile phone form factors: the candy bar, the slider, and the clamshell.

With their postage stamp-sized low-resolution screens and limited storage and processing power, these phones couldn't handle the data-intensive operations required by traditional web browsers. The bandwidth requirements for data transmission were also costly to the user.

The Wireless Application Protocol (WAP) standard emerged to address these concerns. Simply put, WAP was a stripped-down version of HTTP, which is the backbone protocol of the Internet. Unlike traditional web browsers, WAP browsers were designed to run within the memory and bandwidth constraints of the phone. Third-party WAP sites served up pages written in a markup language called Wireless Markup Language (WML). These pages were then displayed on the phone's WAP browser. Users navigated as they would on the Web, but the pages were much simpler in design.

The WAP solution was great for handset manufacturers. The pressure was off—they could write one WAP browser to ship with the handset and rely on developers to come up with the content users wanted.

The WAP solution was great for mobile operators. They could provide a custom WAP portal, directing their subscribers to the content they wanted to provide, and rake in the data charges associated with browsing, which were often high.

Developers and content providers didn't deliver. For the first time, developers had a chance to develop content for phone users, and some did so, with limited success.

Most of the early WAP sites were extensions of popular branded websites, such as CNN.com and ESPN.com, which were looking for new ways to extend their readership. Suddenly phone users accessed the news, stock market quotes, and sports scores on their phones.

Commercializing WAP applications was difficult, and there was no built-in billing mechanism. Some of the most popular commercial WAP applications that emerged during this time were simple wallpaper and ringtone catalogues that enabled users to personalize their phones for the first time. For example, a user browsed a WAP site and requested a specific item. He filled out a simple order form with his phone number and his handset model. It was up to the content provider to deliver an image or audio file compatible with the given phone. Payment and verification were handled through various premium priced delivery mechanisms such as Short Message Service (SMS), Enhanced Messaging Service (EMS), Multimedia Messaging Service (MMS), and WAP Push.

WAP browsers, especially in the early days, were slow and frustrating. Typing long URLs with the numeric keypad was onerous. WAP pages were often difficult to navigate. Most WAP sites were written one time for all phones and did not account for individual phone specifications. It didn't matter if the end user's phone had a big color screen or a postage stamp-sized monochrome screen; the developer couldn't tailor the user's experience. The result was a mediocre and not very compelling experience for everyone involved.

Content providers often didn't bother with a WAP site and instead just advertised SMS short codes on TV and in magazines. In this case, the user sent a premium SMS message with a request for a specific wallpaper or ringtone, and the content provider sent it back. Mobile operators generally liked these delivery mechanisms because they received a large portion of each messaging fee.

WAP fell short of commercial expectations. In some markets, such as Japan, it flourished, whereas in others, such as the United States, it failed to take off. Handset screens were too small for surfing. Reading a sentence fragment at a time, and then waiting seconds for the next segment to download, ruined the user experience, especially because every second of downloading was often charged to the user. Critics began to call WAP

Finally, the mobile operators who provided the WAP portal (the default home page loaded when you started your WAP browser) often restricted which WAP sites were accessible. The portal enabled the operator to restrict the number of sites users could browse and to funnel subscribers to the operator's preferred content providers and exclude competing sites. This kind of walled garden approach further discouraged third-party developers, who already faced difficulties in monetizing applications, from writing applications.

Proprietary Mobile Platforms

It came as no surprise that users wanted more—they will always want more.

Writing robust applications with WAP, such as graphic-intensive video games, was nearly impossible. The 18-year-old to 25-year-old sweet-spot demographic—the kids < with the disposable income most likely to personalize their phones with wallpapers and ringtones—looked at their portable gaming systems and asked for a device that was both a phone and a gaming device or a phone and a music player. They argued that if devices such as Nintendo’s Game Boy could provide hours of entertainment with only five buttons, why not just add phone capabilities? Others looked to their digital cameras, Palms, Blackberries, iPods, and even their laptops and asked the same question. The market seemed to be teetering on the edge of device convergence.

Memory was getting cheaper, batteries were getting better, and PDAs and other embedded devices were beginning to run compact versions of common operating systems such as Linux and Windows. The traditional desktop application developer was suddenly a player in the embedded device market, especially with Smartphone technologies such as Windows Mobile, which they found familiar.

Handset manufacturers realized that if they wanted to continue to sell traditional handsets, they needed to change their protectionist policies pertaining to handset design and expose their internal frameworks to some extent.

A variety of different proprietary platforms emerged—and developers are still actively creating applications for them. Some smartphone devices ran Palm OS (now Garnet OS) and RIM BlackBerry OS. Sun Microsystems took its popular Java platform and J2ME emerged (now known as Java Micro Edition [Java ME]). Chipset maker Qualcomm developed and licensed its Binary Runtime Environment for Wireless (BREW). Other platforms, such as Symbian OS, were developed by handset manufacturers such as Nokia, Sony Ericsson, Motorola, and Samsung. The Apple iPhone OS (OS X iPhone) joined the ranks in 2008. This shows several different phones, all of which have different development platforms.

Many of these platforms have associated developer programs. These programs keep the developer communities small, vetted, and under contractual agreements on what they can and cannot do. These programs are often required and developers must pay for them.

Each platform has benefits and drawbacks. Of course, developers love to debate about which platform is "the best." (Hint: It's usually the platform we're currently developing for.)

The truth is that no one platform has emerged victorious. Some platforms are best suited for commercializing games and making millions—if your company has brand back- ing. Other platforms are more open and suitable for the hobbyist or vertical market applications. No mobile platform is best suited for all possible applications. As a result, the mobile phone has become increasingly fragmented, with all platforms sharing part of the pie.

Manufacturers and mobile operators, handset product lines quickly became complicated. Platform market penetration varies greatly by region and user demographic. Instead of choosing just one platform, manufacturers and operators have been forced to sell phones for all the different platforms to compete in the market. We've even seen some handsets supporting multiple platforms. (For instance, Symbian phones often also support J2ME.)

Phones from various mobile device platforms.

The mobile developer community has become as fragmented as the market. It's nearly impossible to keep track of all the changes in the market. Developer specialty niches have formed. The platform development requirements vary greatly. Mobile software developers work with distinctly different programming environments, different tools, and different programming languages. Porting among the platforms is often costly and not straightforward. Keeping track of handset configurations and testing requirements, signing and certification programs, carrier relationships, and application marketplaces have become complex spin-off businesses of their own.

It's a nightmare for the ACME Company that wants a mobile application. Should it develop a J2ME application? BREW? iPhone? Windows Mobile? Everyone has a different kind of phone. ACME is forced to choose one or, worse, all of the platforms. Some platforms allow for free applications, whereas others do not. Vertical market application opportunities are limited and expensive.

As a result, many wonderful applications have not reached their desired users, and many other great ideas have not been developed at all.

The Open Handset Alliance

Enter search advertising giant Google. Now a household name, Google has shown an interest in spreading its vision, its brand, its search and ad-revenue-based platform, and its suite of tools to the wireless marketplace. The company's business model has been amazingly successful on the Internet and, technically speaking, wireless isn't that different.

Google Goes Wireless

The company's initial forays into mobile were beset with all the problems you would expect. The freedoms Internet users enjoyed were not shared by mobile phone subscribers. Internet users can choose from the wide variety of computer brands, operating systems, Internet service providers, and web browser applications.

Nearly all Google services are free and ad driven. Many applications in the Google Labs suite directly compete with the applications available on mobile phones. The applications range from simple calendars and calculators to navigation with Google Maps and the latest tailored news from News Alerts—not to mention corporate acquisitions such as Blogger and YouTube.

When this approach didn't yield the intended results, Google decided to a different approach—to revamp the entire system upon which wireless application development was based, hoping to provide a more open environment for users and developers: the Internet model. The Internet model allows users to choose between freeware, shareware, and paid software. This enables free market competition among services.

Forming the Open Handset Alliance

With its user-centric, democratic design philosophies, Google has led a movement to turn the existing closely guarded wireless market into one where phone users can move between carriers easily and have unfettered access to applications and services. With its vast resources, Google has taken a broad approach, examining the wireless infrastructure from the FCC wireless spectrum policies to the handset manufacturers' requirements, application developer needs, and mobile operator desires.

Next, Google joined with other like-minded members in the wireless community and posed the following question: What would it take to build a better mobile phone?

The Open Handset Alliance (OHA) was formed in November 2007 to answer that very question. The OHA is a business alliance comprised of many of the largest and most successful mobile companies on the planet. Its members include chip makers, handset manufacturers, software developers, and service providers. The entire mobile supply chain is well represented.

Andy Rubin has been credited as the father of the Android platform. His company, Android Inc., was acquired by Google in 2005.Working together, OHA members, including Google, began developing a nonproprietary open standard platform based upon technology developed at Android Inc. that would aim to alleviate the aforementioned problems hindering the mobile community. The result is the Android project. To this day, most Android platform development is completed by Rubin's team at Google, where he acts as VP of Engineering and manages the Android platform roadmap.

Google's involvement in the Android project has been so extensive that the line between who takes responsibility for the Android platform (the OHA or Google) has blurred. Google hosts the Android open source project and provides online Android documentation, tools, forums, and the Software Development Kit (SDK) for developers. All major Android news originates at Google. The company has also hosted a number of events at conferences and the Android Developer Challenge (ADC), a contest to encourage developers to write killer Android applications—for $10 million dollars in prizes to spur development on the platform. The winners and their apps are listed on the Android website.

Manufacturers: Designing the Android Handsets

More than half the members of the OHA are handset manufacturers, such as Samsung, Motorola, HTC, and LG, and semiconductor companies, such as Intel, Texas Instruments, NVIDIA, and Qualcomm. These companies are helping design the first generation of Android handsets.

The first shipping Android handset—the T-Mobile G1—was developed by handset manufacturer HTC with service provided by T-Mobile. It was released in October 2008. Many other Android handsets were slated for 2009 and early 2010.The platform gained momentum relatively quickly. Each new Android device was more powerful and exciting than the last. Over the following 18 months, 60 different Android handsets (made by 21different manufacturers) debuted across 59 carriers in 48 countries around the world. By June 2010, at an announcement of a new, highly anticipated Android handset, Google announced more than 160,000 Android devices were being activated each day (for a rate of nearly 60 million devices annually).The advantages of widespread manufacturer and carrier support appear to be really paying off at this point.

The Android platform is now considered a success. It has shaken the mobile market place, gaining ground steadily against competitive platforms such as the Apple iPhone, RIM BlackBerry, and Windows Mobile. The latest numbers (as of Summer 2010) show BlackBerry in the lead with a declining 31% of the smartphone market. Trailing close behind is Apple’s iPhone at 28%. Android, however, is trailing with 19%, though it’s gaining ground rapidly and, according to some sources, is the fastest-selling smartphone platform. Microsoft Windows Mobile has been declining and now trails Android by several percentage points.

Mobile Operators: Delivering the Android Experience

After you have the phones, you have to get them out to the users. Mobile operators from North, South, and Central America; Europe, Asia, India, Australia, Africa, and the Middle East have joined the OHA, ensuring a worldwide market for the Android movement. With almost half a billion subscribers alone, telephony giant China Mobile is a founding member of the alliance.

Much of Android's success is also due to the fact that many Android handsets don't come with the traditional "Smartphone price tag"—quite a few are offered free with activation by carriers. Competitors such as the Apple iPhone have no such offering as of yet. For the first time, the average Jane or Joe can afford a feature-full phone. I've lost count of the number of times I've had a waitress, hotel night manager, or grocery store checkout person tell me that they just got an Android phone and it has changed their life. This phenomenon has only added to the Android's rising underdog status.

In the United States, the Android platform was given a healthy dose of help from carriers such as Verizon, who launched a $100 million dollar campaign for the first Droid handset. Many other Droid-style phones have followed from other carriers. Sprint recently launched the Evo 4G (America's first 4G phone) to much fanfare and record one day sales.

Content Providers: Developing Android Applications

When users have Android handsets, they need those killer apps, right?

Google has led the pack, developing Android applications, many of which, such as the email client and web browser, are core features of the platform. OHA members are also working on Android application integration. eBay, for example, is working on integration with its online auctions.

The first ADC received 1,788 submissions, with the second ADC being voted upon by 26,000 Android users to pick a final 200 applications that would be judged professionally—all newly developed Android games, productivity helpers, and a slew of location-based services (LBS) applications. We also saw humanitarian, social networking, and mash-up apps. Many of these applications have debuted with users through the Android Market—Google's software distribution mechanism for Android. For now, these challenges are over. The results, though, are still impressive.

For those working on the Android platform from the beginning, handsets couldn't come fast enough. The T-Mobile G1 was the first commercial Android device on the market, but it had the air of a developer pre-release handset. Subsequent Android handsets have had much more impressive hardware, allowing developers to dive in and design awesome new applications.As of October 2010, there are more than 80,000 applications available in the Android Market, which is growing rapidly. This takes into account only applications published through this one marketplace—not the many other applications sold individually or on other markets. This also does not take into account that, as of Android 2.2, Flash applications can run on Android handsets. This opens up even more application choices for Android users and more opportunities for Android developers.

There are now more than 180,000 Android developers writing interesting and exciting applications. By the time you finish reading this book, you will be adding your expertise to this number.

Taking Advantage of All Android Has to Offer

Android's open platform has been embraced by much of the mobile development community—extending far beyond the members of the OHA.

As Android phones and applications have become more readily available, many other mobile operators and handset manufacturers have jumped at the chance to sell Android phones to their subscribers, especially given the cost benefits compared to proprietary platforms. The open standard of the Android platform has resulted in reduced operator costs in licensing and royalties, and we are now seeing a migration to open handsets from proprietary platforms such as RIM, Windows Mobile, and the Apple iPhone. The market has cracked wide open; new types of users are able to consider smartphones for the first time. Android is well suited to fill this demand.

Android Platform Differences

Android is hailed as "the first complete, open, and free mobile platform":

- **Complete:** The designers took a comprehensive approach when they developed the Android platform. They began with a secure operating system and built a robust software framework on top that allows for rich application development opportunities.

- **Open:** The Android platform is provided through open source licensing. Developers have unprecedented access to the handset features when developing applications.
- **Free:** Android applications are free to develop. There are no licensing or royalty fees to develop on the platform. No required membership fees. No required testing fees. No required signing or certification fees. Android applications can be distributed and commercialized in a variety of ways.

Android: A Next-Generation Platform

Although Android has many innovative features not available in existing mobile platforms, its designers also leveraged many tried-and-true approaches proven to work in the wire less world. It's true that many of these features appear in existing proprietary platforms, but Android combines them in a free and open fashion while simultaneously addressing many of the flaws on these competing platforms.

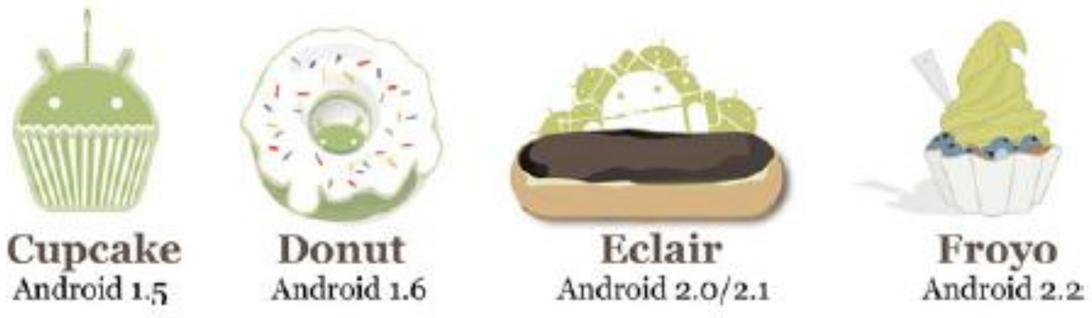
The Android mascot is a little green robot, shown .This little guy (girl?) is often used to depict Android-related materials.

Android is the first in a new generation of mobile platforms, giving its platform developers a distinct edge on the competition. Android's designers examined the benefits and drawbacks of existing platforms and then incorporated their most successful features. At the same time, Android's designers avoided the mistakes others suffered in the past.

Since the Android 1.0 SDK was released, Android platform development has continued at a fast and furious pace. For quite some time, there was a new Android SDK out every couple of months! In typical tech-sector jargon, each Android SDK has had a project name. In Android’s case, the SDKs are named alphabetically after sweets. The latest version of Android is codenamed Gingerbread.

The Android mascot and logo.

Some Android SDKs and their code names.



Free and Open Source

Android is an open source platform. Neither developers nor handset manufacturers pay royalties or license fees to develop for the platform.

The underlying operating system of Android is licensed under GNU General Public License Version 2 (GPLv2), a strong “copy left” license where any third-party improvements must continue to fall under the open source licensing agreement terms. The Android framework is distributed under the Apache Software License (ASL/Apache2), which allows for the distribution of both open- and closed-source derivations of the source code. Commercial developers (handset manufacturers especially) can choose to enhance the platform without having to provide their improvements to the open source community. Instead, developers can profit from enhancements such as handset-specific improvements and redistribute their work under whatever licensing they want.

Android application developers have the ability to distribute their applications under whatever licensing scheme they prefer. Developers can write open source freeware or traditional licensed applications for profit and everything in between.

Familiar and Inexpensive Development Tools

Unlike some proprietary platforms that require developer registration fees, vetting, and expensive compilers, there are no upfront costs to developing Android applications.

Freely Available Software Development Kit

The Android SDK and tools are freely available. Developers can download the Android SDK from the Android website after agreeing to the terms of the Android Software Development Kit License Agreement.

Familiar Language, Familiar Development Environments

Developers have several choices when it comes to integrated development environments (IDEs). Many developers choose the popular and freely available Eclipse IDE to design and develop Android applications. Eclipse is the most popular IDE for Android development, and there is an Android plug-in available for facilitating Android development. Android applications can be developed on the following operating systems:

- Windows XP (32-bit) or Vista (32-bit or 64-bit)
- Mac OS X 10.5.8 or later (x86 only)
- Linux (tested on Linux Ubuntu 8.04 LTS, Hardy Heron)

Reasonable Learning Curve for Developers

Android applications are written in a well-respected programming language: Java.

The Android application framework includes traditional programming constructs, such as threads and processes and specially designed data structures to encapsulate objects commonly used in mobile applications. Developers can rely on familiar class libraries, such as java.net and java.text. Specialty libraries for tasks such as graphics and database management are implemented using well-defined open standards such as OpenGL Embedded Systems (OpenGL ES) or SQLite.

Enabling Development of Powerful Applications

In the past, handset manufacturers often established special relationships with trusted third-party software developers (OEM/ODM relationships).This elite group of software developers wrote native applications, such as messaging and web browsers, which shipped on the handset as part of the phone's core feature set. To design these applications, the manufacturer would grant the developer privileged inside access and knowledge of a handset's internal software framework and firmware.

On the Android platform, there is no distinction between native and third-party applications, enabling healthy competition among application developers. All Android applications use the same libraries. Android applications have unprecedented access to the underlying hardware, allowing developers to write much more powerful applications. Applications can be extended or replaced altogether. For example, Android developers are now free to design email clients tailored to specific email servers, such as Microsoft Exchange or Lotus Notes.

Rich, Secure Application Integration

Recall from the bat story I previously shared that I accessed a variety of phone applications in the course of a few moments: text messaging, phone dialer, camera, email, picture messaging, and the browser. Each was a separate application running on the phone—some built-in and some purchased. Each had its own unique user interface. None were truly integrated.

Not so with Android. One of the Android platform’s most compelling and innovative features is well-designed application integration. Android provides all the tools necessary to build a better “bat trap,” if you will, by allowing developers to write applications that seamlessly leverage core functionality such as web browsing, mapping, contact management, and messaging. Applications can also become content providers and share their data among each other in a secure fashion.

Platforms such as Symbian have suffered from setbacks due to malware. Android’s vigorous application security model helps protect the user and the system from malicious software.

No Costly Obstacles to Publication

Android applications have none of the costly and time-intensive testing and certification programs required by other platforms such as BREW and Symbian.

A “Free Market” for Applications

Android developers are free to choose any kind of revenue model they want. They can develop freeware, shareware, or trial-ware applications, ad-driven, and paid applications. Android was designed to fundamentally change the rules about what kind of wireless applications could be developed. In the past, developers faced many restrictions that had little to do with the application functionality or features:

- Store limitations on the number of competing applications of a given type
- Store limitations on pricing, revenue models, and royalties
- Operator unwillingness to provide applications for smaller demographics

With Android, developers can write and successfully publish any kind of application they want. Developers can tailor applications to small demographics, instead of just large-scale money-making ones often insisted upon by mobile operators. Vertical market applications can be deployed to specific, targeted users.

Because developers have a variety of application distribution mechanisms to choose from, they can pick the methods that work for them instead of being forced to play by others' rules. Android developers can distribute their applications to users in a variety of ways:

- Google developed the Android Market, a generic Android application store with a revenue-sharing model.
- Handango.com added Android applications to its existing catalogue using their billing models and revenue-sharing model.
- Developers can come up with their own delivery and payment mechanisms.

Mobile operators are still free to develop their own application stores and enforce their own rules, but it will no longer be the only opportunity developers have to distribute their applications.

A New and Growing Platform

Android might be the next generation in mobile platforms, but the technology is still in its early stages. Early Android developers have had to deal with the typical roadblocks associated with a new platform: frequently revised SDKs, lack of good documentation, and market uncertainties.

On the other hand, developers diving into Android development now benefit from the first-to-market competitive advantages we've seen on other platforms such as BREW and Symbian. Early developers who give feedback are more likely to have an impact on the long-term design of the Android platform and what features will come in the next version of the SDK. Finally, the Android forum community is lively and friendly. Incentive programs, such as the ADC, have encouraged many new developers to dig into the platform.

Each new version of the Android SDK has provided a number of substantial improvements to the platform. In recent revisions, the Android platform has received some much needed UI "polish," both in terms of visual appeal and performance. Although most of these upgrades and improvements were welcome and necessary, new SDK versions often cause some upheaval within the Android developer community. A number of published applications have required retesting and resubmission to the Android Marketplace to conform to new SDK requirements, which are quickly rolled out to all Android phones in the field as a firmware upgrade, rendering older applications obsolete.

Some older Android handsets are not capable of running the latest versions of the platform. This means that Android developers often need to target several different SDK versions to reach all users. Luckily, the Android development tools make this easier than ever.

The Android Platform

Android is an operating system and a software platform upon which applications are developed. A core set of applications for everyday tasks, such as web browsing and email, are included on Android handsets.

As a product of the OHA's vision for a robust and open source development environment for wireless, Android is an emerging mobile development platform. The platform was designed for the sole purpose of encouraging a free and open market that all mobile applications phone users might want to have and software developers might want to develop.

Android's Underlying Architecture

The Android platform is designed to be more fault-tolerant than many of its predecessors. The handset runs a Linux operating system upon which Android applications are executed in a secure fashion. Each Android application runs in its own virtual machine. Android applications are managed code; therefore, they are much less likely to cause the phone to crash, leading to fewer instances of device corruption (also called "bricking" the phone, or rendering it useless).

The Linux Operating System

The Linux 2.6 kernel handles core system services and acts as a hardware abstraction layer (HAL) between the physical hardware of the handset and the Android software stack. Some of the core functions the kernel handles include

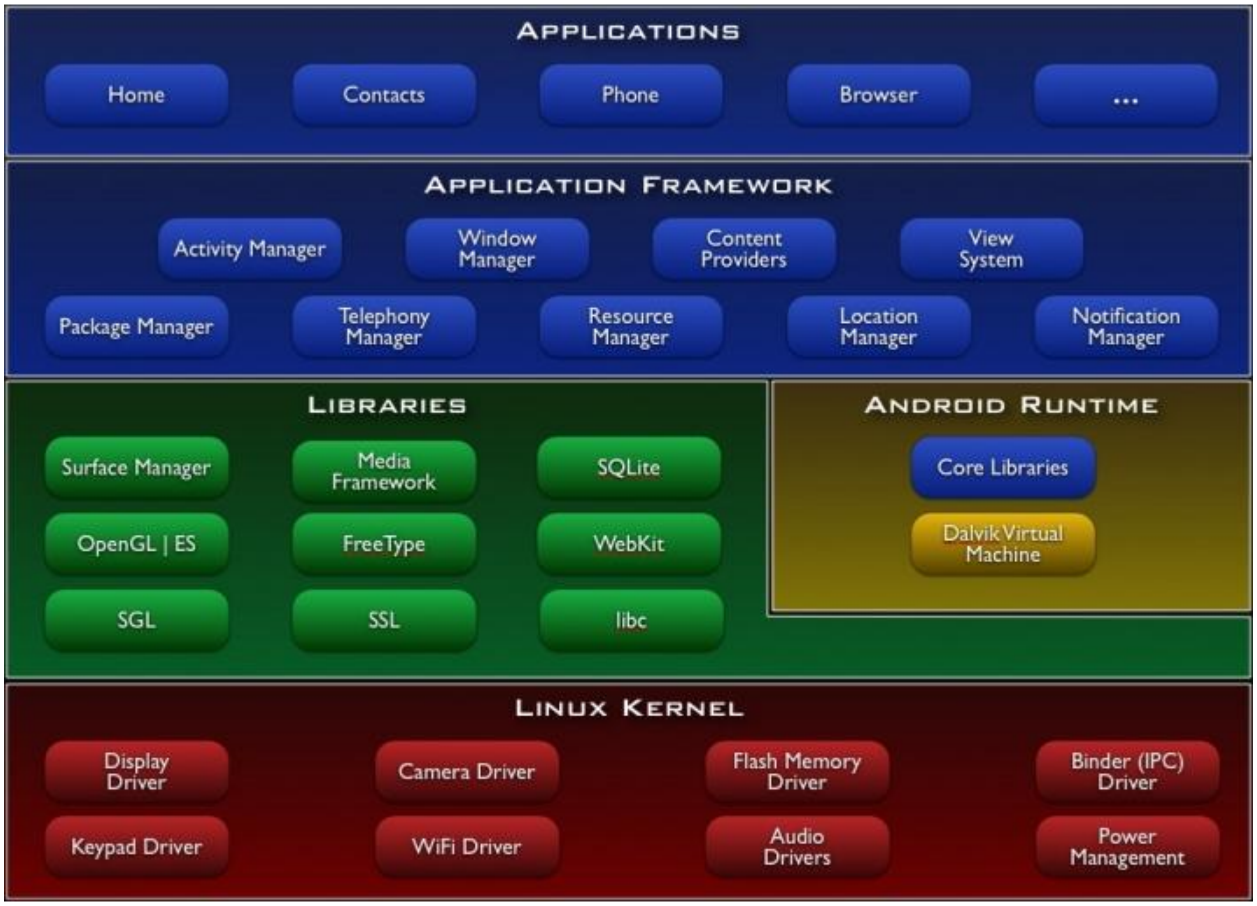
- Enforcement of application permissions and security
- Low-level memory management
- Process management and threading
- The network stack

- Display, keypad input, camera, Wi-Fi, Flash memory, audio, and binder (IPC) driver access.

Android Application Runtime Environment

Each Android application runs in a separate process, with its own instance of the Dalvik virtual machine (VM). Based on the Java VM, the Dalvik design has been optimized for mobile devices. The Dalvik VM has a small memory footprint, and multiple instances of the Dalvik VM can run concurrently on the handset.

Diagram of the Android platform architecture.



Security and Permissions

The integrity of the Android platform is maintained through a variety of security measures. These measures help ensure that the user's data is secure and that the device is not subjected to malware.

Applications as Operating System Users

When an application is installed, the operating system creates a new user profile associa- ted with the application. Each application runs as a different user, with its own private files on the file system, a user ID, and a secure operating environment.

The application executes in its own process with its own instance of the Dalvik VM and under its own user ID on the operating system.

Explicitly Defined Application Permissions

To access shared resources on the system, Android applications register for the specific privileges they require. Some of these privileges enable the application to use phone functionality to make calls, access the network, and control the camera and other hardware sensors. Applications also require permission to access shared data containing private and personal information, such as user preferences, user's location, and contact information.

Applications might also enforce their own permissions by declaring them for other applications to use. The application can declare any number of different permission types, such as read-only or read-write permissions, for finer control over the application.

Limited Ad-Hoc Permissions

Applications that act as content providers might want to provide some on-the-fly permissions to other applications for specific information they want to share openly. This is done using ad-hoc granting and revoking of access to specific resources using Uniform Resource Identifiers (URIs).

URIs index specific data assets on the system, such as images and text. Here is an example of a URI that provides the phone numbers of all contacts:

content://contacts/phones

To understand how this permission process works, let's look at an example.

Let's say we have an application that keeps track of the user's public and private birthday wish lists. If this application wanted to share its data with other applications, it could grant URI permissions for the public wish list, allowing another application permission to access this list without explicitly having to ask for it.

Application Signing for Trust Relationships

All Android applications packages are signed with a certificate, so users know that the application is authentic. The private key for the certificate is held by the developer. This helps establish a trust relationship between the developer and the user. It also enables the developer to control which applications can grant access to one another on the system. No certificate authority is necessary; self-signed certificates are acceptable.

Marketplace Developer Registration

To publish applications on the popular Android Market, developers must create a developer account. The Android Market is managed closely and no malware is tolerated.

Developing Android Applications

The Android SDK provides an extensive set of application programming interfaces (APIs) that is both modern and robust. Android handset core system services are exposed and accessible to all applications. When granted the appropriate permissions, Android applica- tions can share data among one another and access shared resources on the system securely.

Android Programming Language Choices

Android applications are written in Java. For now, the Java language is the developer's only choice on the Android platform.

There has been some speculation that other programming languages, such as C++, might be added in future versions of Android. If your application must rely on native code in another language such as C or C++, you might want to consider integrating it using the Android Native Development Kit (NDK).We talk more about this in Chapter "Using the Android NDK."

No Distinctions Made Between Native and Third-Party Applications

Unlike other mobile development platforms, there is no distinction between native applications and developer-created applications on the Android platform. Provided the application is granted the appropriate permissions, all applications have the same access to core libraries and the underlying hardware interfaces.

Android handsets ship with a set of native applications such as a web browser and contact manager. Third-party applications might integrate with these core applications, extend them to provide a rich user experience, or replace them entirely with alternative applica- tions.

Commonly Used Packages

With Android, mobile developers no longer have to reinvent the wheel. Instead, developers use familiar class libraries exposed through Android's Java packages to perform common tasks such as graphics, database access, network access, secure communications, and utilities (such as XML parsing).

- The Android packages include support for
- Common user interface widgets (Buttons, Spin Controls,Text Input)
- User interface layout
- Secure networking and web browsing features (SSL,WebKit)
- Structured storage and relational databases (SQLite)
- Powerful 2D and 3D graphics (including SGL and OpenGL ES)
- Audio and visual media formats (MPEG4, MP3, Still Images)
- Access to optional hardware such as location-based services (LBS),Wi-Fi, Bluetooth, and hardware sensors

Android Application Framework

The Android application framework provides everything necessary to implement your average application. The Android application lifecycle involves the following key components:

- Activities are functions the application performs.
- Groups of views define the application's layout.

- Intents inform the system about an application's plans.
- Services allow for background processing without user interaction.
- Notifications alert the user when something interesting happens.

Android applications can interact with the operating system and underlying hardware using a collection of managers. Each manager is responsible for keeping the state of some underlying system service. For example, there is a LocationManager that facilitates interaction with the location-based services available on the handset. The ViewManager and WindowManager manage user interface fundamentals.

Applications can interact with one another by using or acting as ContentProvider Built-in applications such as the Contact manager are content providers, allowing third-party applications to access contact data and use it in an infinite number of ways. The sky is the limit.

Setting up your Android Development environment

Setting Up Your Android Development Environment

Android developers write and test applications on their computers and then deploy those applications onto the actual device hardware for further testing.

In this chapter, you become familiar with all the tools you need master in order to develop Android applications. You also explore the Android Software Development Kit (SDK) installation and all it has to offer.

Configuring Your Development Environment

To write Android applications, you must configure your programming environment for Java development. The software is available online for download at no cost. Android applications can be developed on Windows, Macintosh, or Linux systems.

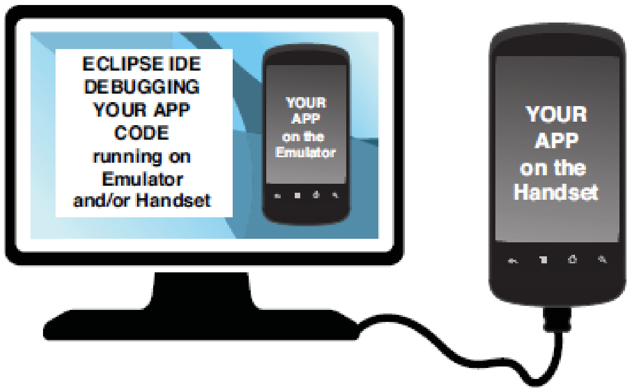
To develop Android applications, you need to have the following software installed on your computer:

- The **Java Development Kit** (JDK) Version 5 or 6, available for download.
- A compatible **Java IDE such as Eclipse** along with its JDT plug-in, available for download.
- The **Android SDK**, tools and documentation, available for download .
- The **Android Development Tools** (ADT) plug-in for Eclipse, available for download through the Eclipse software update mechanism. For instructions on how to install. Although this tool is optional for development, we highly recommend it and will use its features frequently throughout this book.

A complete list of Android development system requirements is available.

Configuring Your Operating System for Device Debugging

To install and debug Android applications on Android devices, you need to configure your operating system to access the phone via the USB cable. On some operating systems, such as Mac OS, this may just work. However, for Windows installations, you need to install the appropriate USB driver. You can download the Windows USB driver.


ECLIPSE IDE DEBUGGING YOUR APP CODE running on Emulator and/or Handset
YOUR APP on the Emulator
YOUR APP on the Handset

Android application debugging using the emulator and an Android handset.

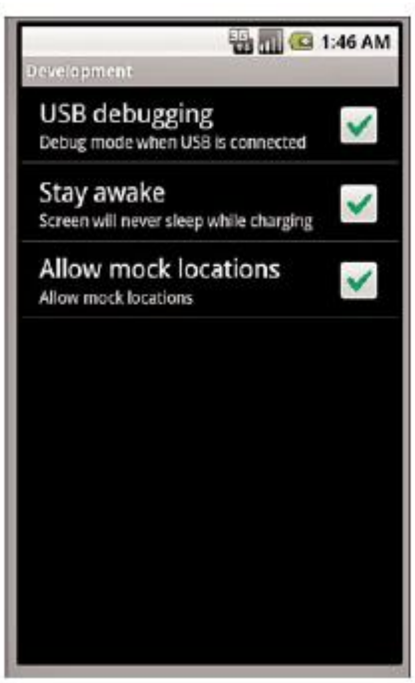
Configuring Your Android Hardware for Debugging

Android devices have debugging disabled by default. Your Android device must be enabled for debugging via a USB connection in order to develop applications and run them on the device.

First, you need to enable your device to install Android applications other than those from the Android Market. This setting is reached by selecting Home, Menu, Settings, Applications. Here you should check (enable) the option called Unknown Sources.

More important development settings are available on the Android device by selecting Home, Menu, Settings, Applications, Development. Here you should enable the following options:

- **USB Debugging:** This setting enables you to debug your applications via the USB connection.



USB debugging
Debug mode when USB is connected ✓

Stay awake
Screen will never sleep while charging ✓

Allow mock locations
Allow mock locations ✓

- **Stay Awake:** This convenient setting keeps the phone from sleeping in the middle of your development work, as long as the device is plugged in.
- **Allow Mock Locations:** This setting enables you to send mock location information to the phone for development purposes and is very convenient for applications using location-based services (LBS).

Android debug settings.

Upgrading the Android SDK

The Android SDK is upgraded from time to time. You can easily upgrade the Android SDK and tools from within Eclipse using the Android SDK and AVD Manager, which is installed as part of the ADT plug-in for Eclipse.

Changes to the Android SDK might include addition, update, and removal of features; package name changes; and updated tools. With each new version of the SDK, Google provides the following useful documents:

- **An Overview of Changes:** A brief description of major changes to the SDK.
- **An API Diff Report:** A complete list of specific changes to the SDK.
- **Release Notes:** A list of known issues with the SDK.

You can find out more about adding and updating SDK components.

Problems with the Android Software Development Kit

Because the Android SDK is constantly under active development, you might come across problems with the SDK. If you think you've found a problem, you can find a list of open issues and their status at the Android project Issue Tracker website. You can also submit new issues for review.

Exploring the Android SDK

The Android SDK comes with five major components: the Android SDK License Agreement, the Android Documentation, Application Framework, Tools, and Sample Applications.

Understanding the Android SDK License Agreement

Before you can download the Android SDK, you must review and agree to the Android SDK License Agreement. This agreement is a contract between you (the developer) and Google (copyright holder of the Android SDK).

Even if someone at your company has agreed to the Licensing Agreement on your behalf, it is important for you, the developer, to be aware of a few important points:

1. **Rights granted:** Google (as the copyright holder of Android) grants you a limited, worldwide, royalty-free, non-assignable, and non-exclusive license to use the SDK solely to develop applications for the Android platform. Google (and third-party contributors) are granting you license, but they still hold all copyrights and intellectual property rights to the material. Using the Android SDK does not grant you permission to use any Google brands, logos, or trade names. You will not remove any of the copyright notices therein. Third-party applications that your applications interact with (other Android apps) are subject to separate terms and fall outside this agreement.

2. **SDK usage:** You may only develop Android applications. You may not make derivative works from the SDK or distribute the SDK on any device or distribute part of the SDK with other software.
3. **SDK changes and backward compatibility:** Google may change the Android SDK at any time, without notice, without regard to backward compatibility. Although Android API changes were a major issue with prerelease versions of the SDK, recent releases have been reasonably stable. That said, each SDK update does tend to affect a small number of existing applications in the field, necessitating updates.
4. **Android application developer rights:** You retain all rights to any Android software you develop with the SDK, including intellectual property rights.You also retain all responsibility for your own work.
5. **Android application privacy requirements:** You agree that your applications will protect the privacy and legal rights of its users. If your application uses or accesses personal and private information about the user (usernames, passwords, and so on), then your application will provide an adequate privacy notice and keep that data stored securely. Note that privacy laws and regulations may vary by user location; you as a developer are solely responsible for managing this data appropriately.
6. **Android application malware requirements:** You are responsible for all applications you develop. You agree not to write disruptive applications or malware. You are solely responsible for all data transmitted through your application.
7. **Additional terms for specific Google APIs:** Use of the Android Maps API is subject to further Terms of Service (specifically use of the following packages: com. Google. android. maps and com.android.location.Geocoder).You must agree to these additional terms before using those specific APIs and always include the Google Maps copyright notice provided. Use of Google Data APIs (Google Apps such as Gmail, Blogger, Google Calendar, Google Finance Portfolio Data, Picasa, YouTube, and so on) is limited to access that the user has explicitly granted permission to your application by accepted permissions provided by the developer during installation time.
8. **Develop at your own risk:** Any harm that comes about from developing with the Android SDK is your own fault and not Google's.

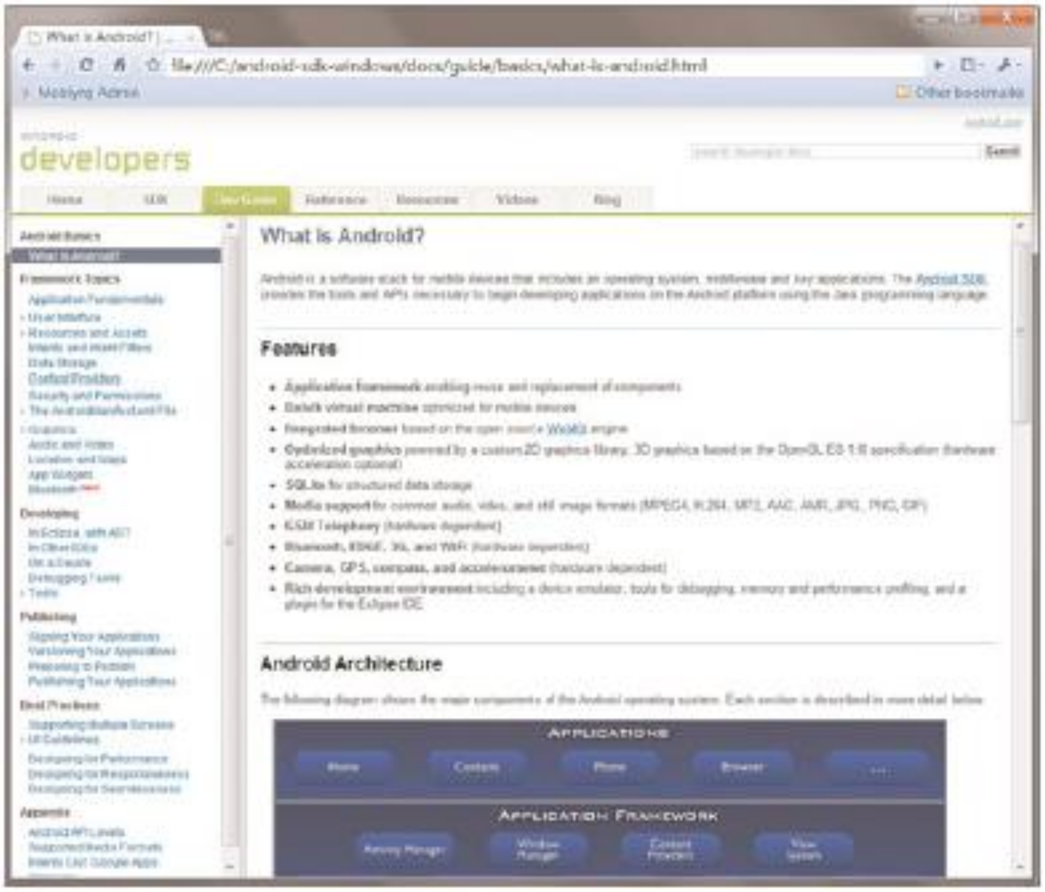
Reading the Android SDK Documentation

A local copy of the Android documentation is provided in the /docs subfolder on disk.

The documentation is now divided into seven main sections:

- The **Home** tab is your general starting point within the Android documentation. Here you find developer announcements and important links to the latest hot topics in Android development.
- The **SDK** tab provides information about the different Android SDK versions available, as well as information about the Android Native Development Kit (NDK).You find the Android SDK release notes here as well.

The Android SDK documentation.



file:///C:/android-sdk-windows/docs/guide/basics/what-is-android.html

What is Android?

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android SDK provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language.

Features

- Application framework enabling reuse and replacement of components
- Dalvik virtual machine optimized for mobile devices
- Integrated browser based on the open source WebKit engine
- Optimized graphics powered by a custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0 specification (hardware acceleration optional)
- SQLite for structured data storage
- Media support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- GSM Telephony (hardware dependent)
- Bluetooth, EDGE, 3G, and WiFi (hardware dependent)
- Camera, GPS, compass, and accelerometer (hardware dependent)
- Rich development environment including a device emulator, tools for debugging, memory and performance profiling, and a plugin for the Eclipse IDE

Android Architecture

The following diagram shows the major components of the Android operating system. Each section is described in more detail below.

- The **Dev Guide** tab introduces the Android platform and covers best practices for Android application design and development, as well as information about publishing applications.
- The **Reference** tab provides a drill-down listing of the Android APIs with detailed coverage of specific classes and interfaces.
- The **Resources** tab provides access to Android technical articles and tutorials. Here you also find links to the Android community online (groups, mailing list, and official Twitter feed), as well as the sample applications provided along with the Android SDK.
- The **Videos** tab provides access to online videos pertaining to Android development, including videos about the platform, developer tips, Android development sessions from the annual Google I/O conference, and developer sandbox interviews.
- The **Blog** tab provides access to the online blog published by the Android development team. Here you find announcements about SDK releases, helpful development tips, and notices of upcoming Android events.

The Android documentation is provided in HTML format locally and online . Certain networked features of the Android documentation (such as the Blog and Video tabs) are only available online.

Exploring the Android Application Framework

The Android application framework is provided in the android.jar file. The Android SDK is made up of several important packages, as shown in Table.

There is also an optional Google APIs Add-On, which is an extension to the Android SDK that helps facilitate development using Google Maps and other Google APIs and services. For example, if you want to include the MapView control in your application, you need to install and use this feature. This Add-On corresponds to the com. google.* package (including com.google.android.maps) and requires agreement to additional Terms of Service and registration for an API Key. For more information on the Google APIs Add-On.

Important Packages in the Android SDK

Top-Level Package	Purpose
`android.*`	Android application fundamentals
`dalvik.*`	Dalvik Virtual Machine support classes
`java.*`	Core classes and familiar generic utilities for networking, security, math, and such
`javax.*`	Java extension classes including encryption support, parsers, SQL, and such
`junit.*`	Unit testing support
`org.apache.http.*`	Hypertext Transfer Protocol (HTTP) protocol
`org.json`	JavaScript Object Notation (JSON) support
`org.w3c.dom`	W3C Java bindings for the Document Object Model Core (XML and HTML)
`org.xml.sax.*`	Simple API for XML (SAX) support for XML
`org.xmlpull.*`	High-performance XML parsing

Getting to Know the Android Tools

The Android SDK provides many tools to design, develop, debug, and deploy your Android applications. The Eclipse Plug-In incorporates many of these tools seamlessly into your development environment and provides various wizards for creating and debugging Android projects.

Settings for the ADT plug-in are found in Eclipse under Window, Preferences, Android. Here you can set the disk location where you installed the Android SDK and tools, as well as numerous other build and debugging settings.

The ADT plug-in adds a number of useful functions to the default Eclipse IDE. Several new buttons are available on the toolbar, including buttons to

- Launch the Android SDK and AVD Manager
- Create a new project using the Android Project Wizard
- Create a new test project using the Android Project Wizard
- Create a new Android XML resource file

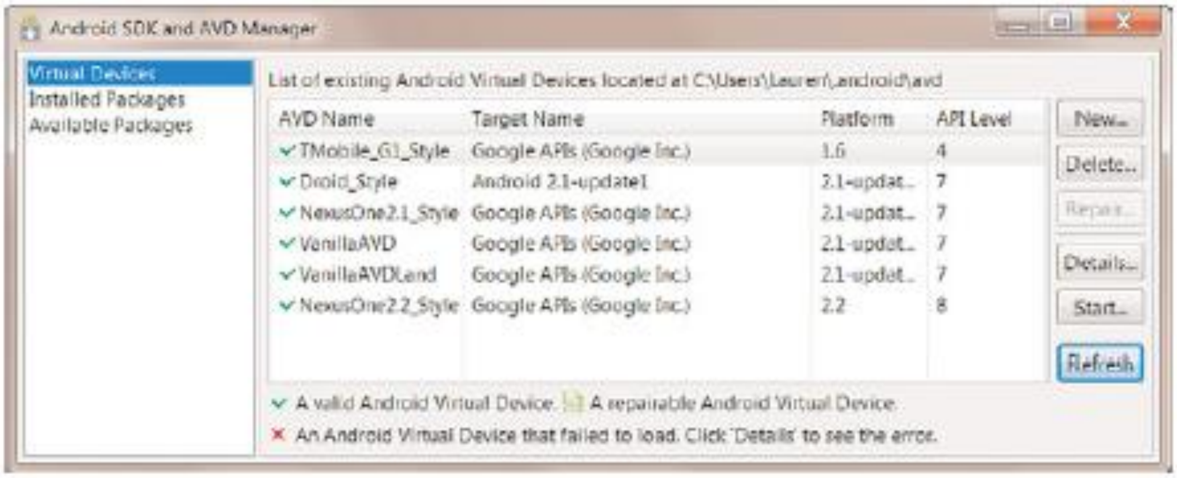
These features are accessible through the Eclipse toolbar buttons shown in Figure.

There is also a special Eclipse perspective for debugging Android applications called DDMS (Dalvik Debug Monitor Server).You can switch to this perspective within Eclipse by choosing Window, Open Perspective, DDMS or by changing to the DDMS perspective in the top-right corner of the screen. We talk more about DDMS later in this chapter. After you have designed an Android application, you can also use the ADT plug-in for Eclipse to launch a wizard to package and sign your Android application for publication. We talk more about this in Chapter "Selling Your Android Application."

Android SDK and AVD Manager

The Android SDK and AVD Manager, shown in Figure, is a tool integrated into Eclipse. This tool performs two major functions: management of multiple versions of the Android SDK on the development machine and management of the developer's Android Virtual Device (AVD) configurations.

The Android SDK and AVD Manager.



List of existing Android Virtual Devices located at C:\Users\Lauren\.android\avd

AVD Name	Target Name	Platform	API Level
✓ TMobile_G1_Style	Google APIs (Google Inc.)	1.6	4
✓ Droid_Style	Android 2.1-update1	2.1-updat...	7
✓ NexusOne2.1_Style	Google APIs (Google Inc.)	2.1-updat...	7
✓ VanillaAVD	Google APIs (Google Inc.)	2.1-updat...	7
✓ VanillaAVDLand	Google APIs (Google Inc.)	2.1-updat...	7
✓ NexusOne2.2_Style	Google APIs (Google Inc.)	2.2	8

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

Much like desktop computers, different Android devices run different versions of the Android operating system. Developers need to be able to target different Android SDK versions with their applications. Some applications target a specific Android SDK, whereas others try to provide simultaneous support for as many versions as possible.

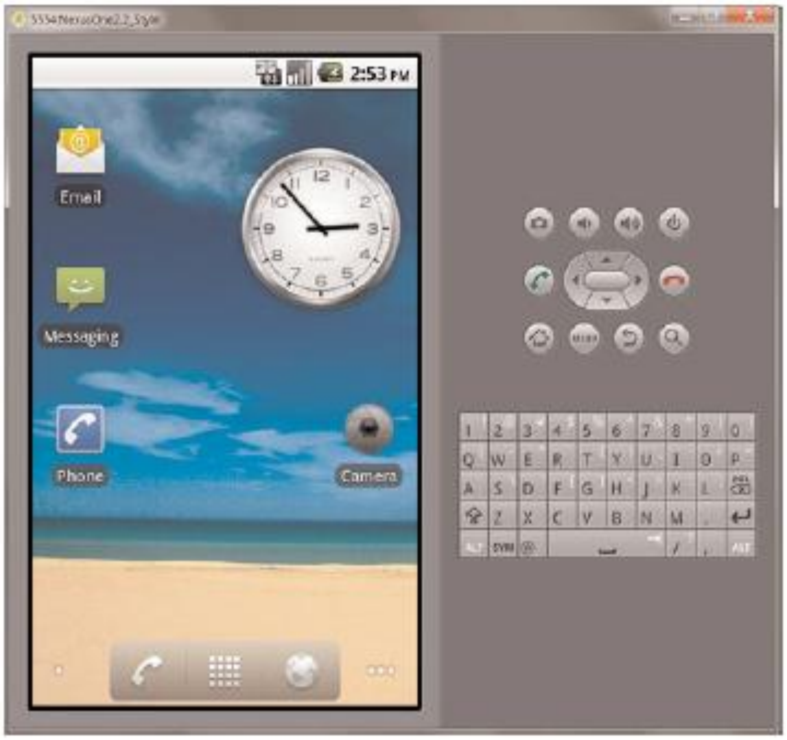
The Android SDK and AVD Manager facilitate Android development across multiple platform versions simultaneously. When a new Android SDK is released, you can use this tool to download and update your tools while still maintaining backward compatibility and use older versions of the Android SDK.

The tool also manages the AVD configurations. To manage applications in the Android emulator, you must configure an AVD. This AVD profile describes what type of device you want the emulator to simulate, including which Android platform to support. You can specify different screen sizes and orientations, and you can specify whether the emulator has an SD card and, if so, what capacity.

Android Emulator

The Android emulator, shown in Figure, is one of the most important tools provided with the Android SDK. You will use this tool frequently when designing and developing Android applications. The emulator runs on your computer and behaves much as a mobile device would. You can load Android applications into the emulator, test, and debug them.

The Android emulator.

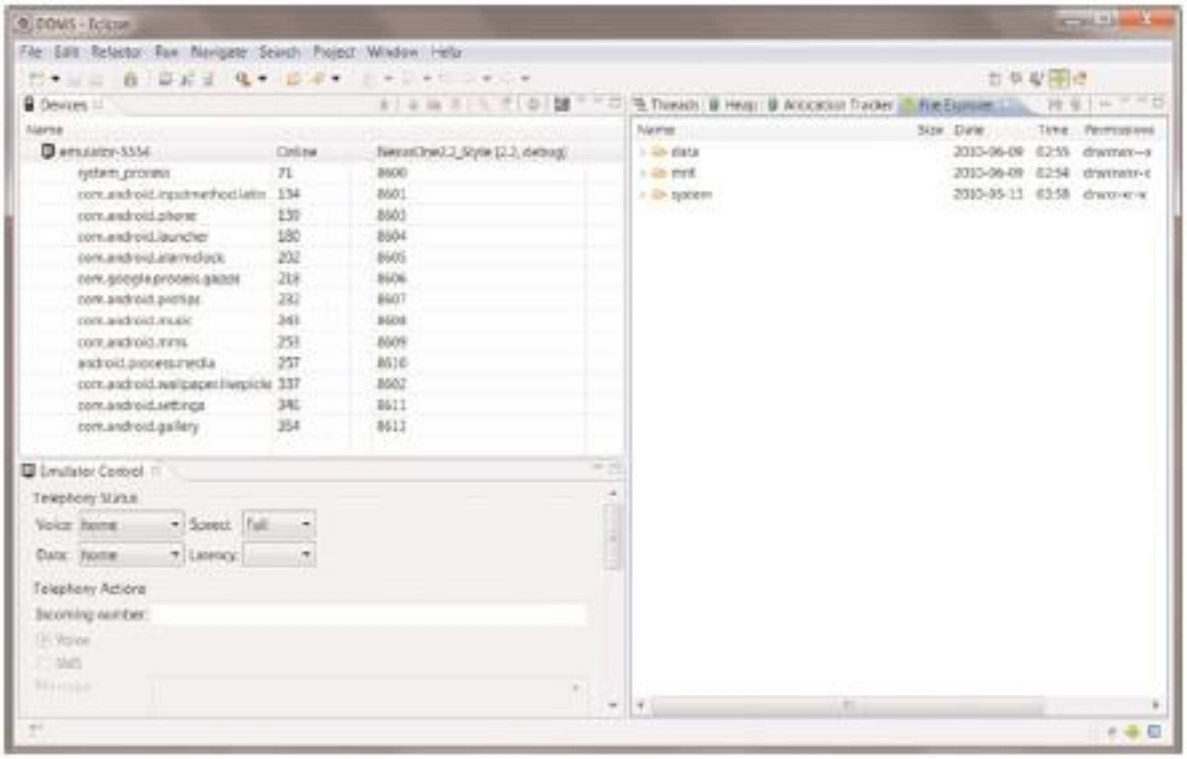


The emulator is a generic device and is not tied to any one specific phone configuration. You describe the hardware and software configuration details that the emulator is to simulate by providing an AVD configuration.

Dalvik Debug Monitor Server (DDMS)

The Dalvik Debug Monitor Server (DDMS) is a command-line tool that has also been integrated into Eclipse as a perspective. This tool provides you with direct access to the device—whether it's the emulator virtual device or the physical device. You use DDMS to view and manage processes and threads running on the device, view heap data, attach to processes to debug, and a variety of other tasks.

Using DDMS integrated into an Eclipse perspective.



Name		
emulator-5554	Online	NexusOne2.2_Style [2.2, debug]
system_process	71	8600
com.android.inputmethod.latin	134	8601
com.android.phone	139	8603
com.android.launcher	180	8604
com.android.alarmclock	202	8605
com.google.process.gapps	218	8606
com.android.protips	232	8607
com.android.music	243	8608
com.android.mms	253	8609
android.process.media	257	8610
com.android.wallpaper.livepicker	337	8602
com.android.settings	346	8611
com.android.gallery	354	8613

Name	Size	Date	Time	Permissions
data		2010-06-09	02:55	drwxrwx--x
mnt		2010-06-09	02:54	drwxrwxr-x
system		2010-05-11	03:58	drwxr-xr-x

Android Debug Bridge (ADB)

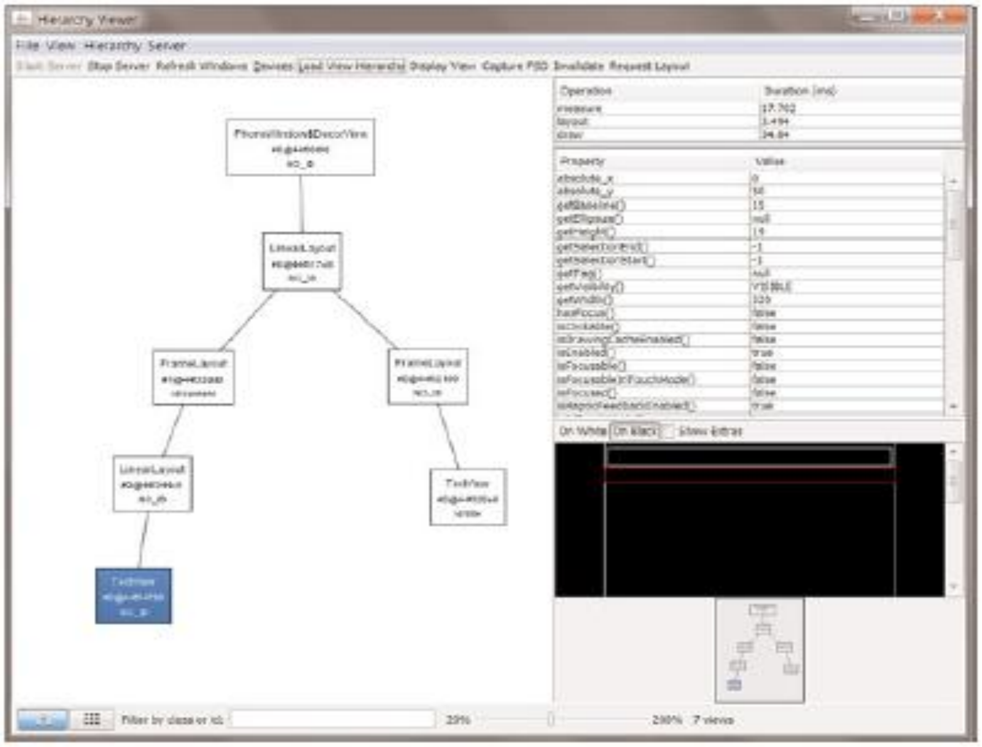
The Android Debug Bridge (ADB) is a client-server tool used to enable developers to debug Android code on the emulator and the device using a standard Java IDE such as Eclipse. The DDMS and the Android Development Plug-In for Eclipse both use the ADB to facilitate interaction between the development environment and the device (or emulator). Developers can also use ADB to interact with the device file system, install Android applications manually, and issue shell commands. For example, the sqlite3 shell commands enable you to access device database.The Application Exerciser

Monkey commands generate random input and system events to stress test your application. One of the most important aspects of the ADB for the developer is its logging system (Logcat).

Android Hierarchy Viewer

The Android Hierarchy Viewer, a visual tool that illustrates layout component relationships, helps developers design and debug user interfaces. Developers can use this tool to inspect the View properties and develop pixel-perfect layouts. For more information about user interface design and the Hierarchy Viewer, see Chapter "Designing User Interfaces with Layouts."

Screenshot of the Android Hierarchy Viewer in action.


Hierarchy Viewer
File View Hierarchy Server
Start Server Stop Server Refresh Windows Devices Load View Hierarchy Display View Capture PSD Invalidate Request Layout
PhoneWindow$DecorView
LinearLayout
FrameLayout
FrameLayout
LinearLayout
TextView
TextView
On White On Black Show Extras
Filter by class or id:
20%
200% 7 views

Operation	Duration (ms)
measure	17.702
layout	1.494
draw	34.84

Property	Value
absolute_x	0
absolute_y	50
getBaseline()	15
getEllipsize()	null
getHeight()	19
getSelectionEnd()	-1
getSelectionStart()	-1
getTag()	null
getVisibility()	VISIBLE
getWidth()	320
hasFocus()	false
isClickable()	false
isDrawingCacheEnabled()	false
isEnabled()	true
isFocusable()	false
isFocusableInTouchMode()	false
isFocused()	false
isHapticFeedbackEnabled()	true

Other Tools

Android SDK provides a number of other tools provided with the Android SDK. Many of these tools provide the underlying functionality that has been integrated into Eclipse using the Android Development Tools (ADT) plug-in. However, if you are not using Eclipse, these tools may be used on the command-line.

Other tools are special-purpose utilities. For example, the Draw Nine-patch tool enables you to design stretchable PNG images, which is useful for supporting different screen sizes. Likewise, the layout opt tool helps developers optimize their user interfaces for performance. We discuss a number of these special tools in later chapters as they become relevant.

Exploring the Android Sample Applications

The Android SDK provides many samples and demo applications to help you learn the ropes of Android Development. Many of these demo applications are provided as part of the Android SDK and are located in the /samples subdirectory of the Android SDK. You can find more sample applications on the Android Developer website under the Resources tab.

Some of the most straightforward demo applications to take a look at are

- **ApiDemos:** A menu-driven utility that demonstrates a wide variety of Android APIs, from user interface widgets to application lifecycle components such as services, alarms, and notifications.
- **Snake:** A simple game that demonstrates bitmap drawing and key events.
- **NotePad:** A simple list application that demonstrates database access and Live Folder functionality.
- **LunarLander:** A simple game that demonstrates drawing and animation.

Writing your first Android Application

RKUniversity™

By : Ketan Bhimani

Writing Your First Android Application

You should now have a workable Android development environment set up on your computer. Hopefully, you also have an Android device as well. Now it's time for you to start writing some Android code.

In this chapter, you learn how to add and create Android projects in Eclipse and verify that your Android development environment is set up correctly. You also write and debug your first Android application in the software emulator and on an Android handset.

Testing Your Development Environment

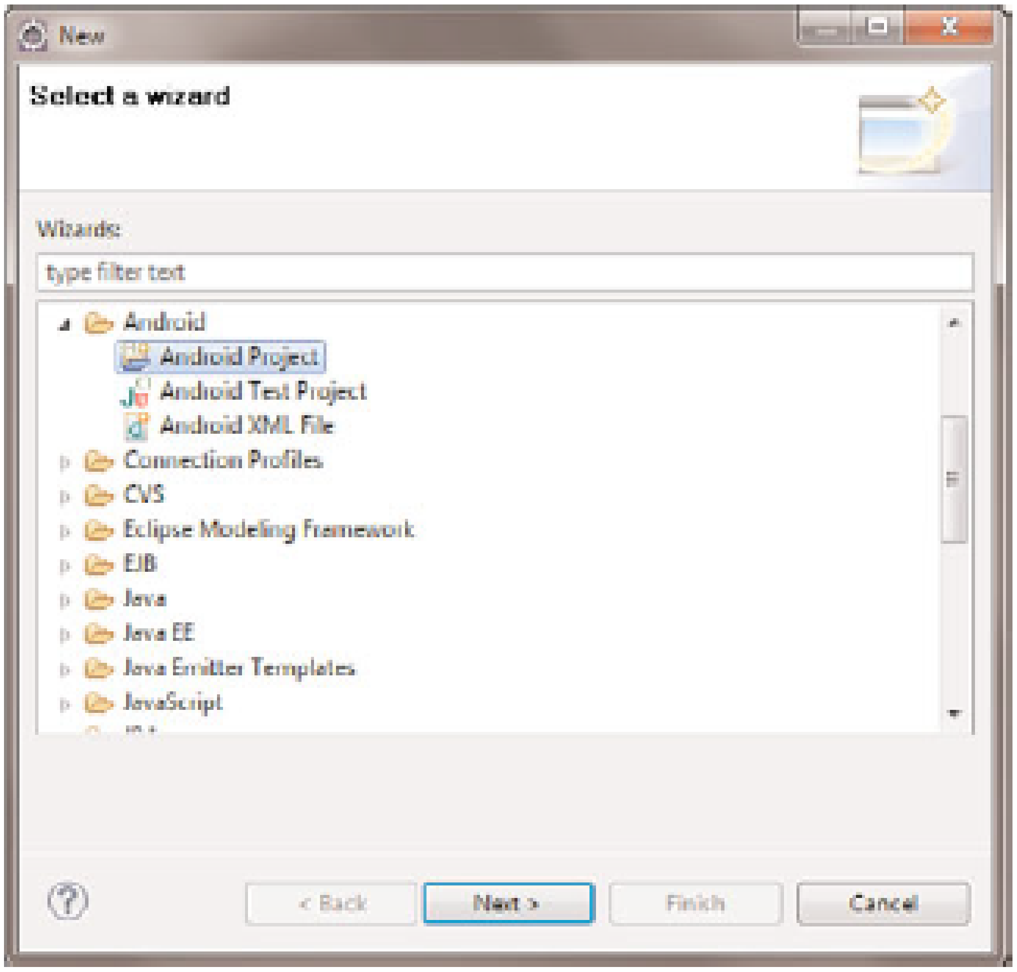
The best way to make sure you configured your development environment correctly is to take an existing Android application and run it. You can do this easily by using one of the sample applications provided as part of the Android SDK in the /samples subdirectory.

Within the Android SDK sample applications, you can find a classic game called Snake. To build and run the Snake application, you must create a new Android project in your Eclipse workspace, create an appropriate Android Virtual Device (AVD) profile, and configure a launch configuration for that project. After you have everything set up correctly, you can build the application and run it on the Android emulator or an Android device.

Adding the Snake Application to a Project in Your Eclipse Workspace

The first thing you need to do is add the Snake project to your Eclipse workspace. To do this, follow these steps:

Creating a new Android project.

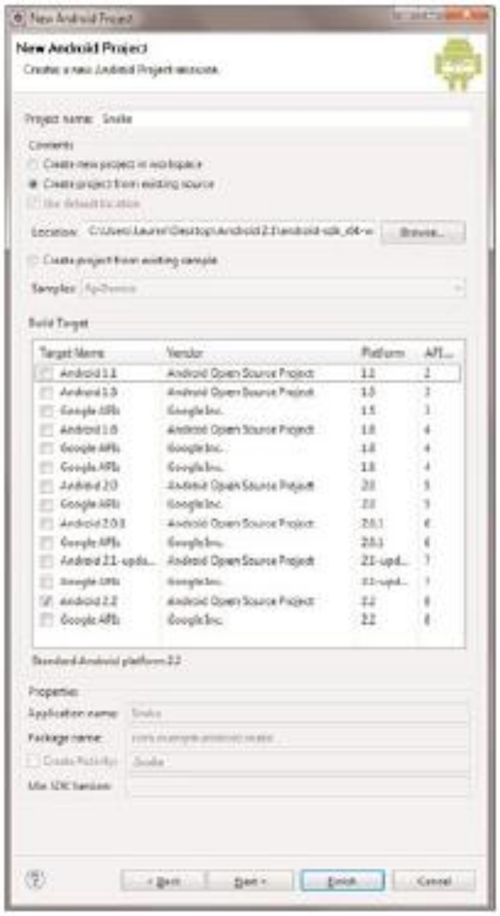


1. Choose File, New, Project.
2. Choose Android, Android Project Wizard.
3. Change the Contents to Create Project from Existing Source.
4. Browse to your Android samples directory.
5. Choose the Snake directory. All the project fields should be filled in for you from the Manifest file. You might want to set the Build Target to whatever version of Android your test device is running.
6. Choose Finish. You now see the Snake project files in your workspace.

Creating an Android Virtual Device (AVD) for Your Snake Project

The next step is to create an AVD that describes what type of device you want to emulate when running the Snake application. This AVD profile describes what type of device you want the emulator to simulate, including which Android platform to support. You can specify different screen sizes and orientations, and you can specify whether the emulator has an SD card and, if it does, what capacity the card is.

The Snake project details.



New Android Project

Create a new Android Project resource.

Project name: Snake

Contents
- ( ) Create new project in workspace
- (●) Create project from existing source
- [ ] Use default location

Location: C:\Users\Lauren\Desktop\Android 2.1\android-sdk_r04-w

( ) Create project from existing sample

Samples: ApiDemos

Build Target

	Target Name	Vendor	Platform	API...
[ ]	Android 1.1	Android Open Source Project	1.1	2
[ ]	Android 1.5	Android Open Source Project	1.5	3
[ ]	Google APIs	Google Inc.	1.5	3
[ ]	Android 1.6	Android Open Source Project	1.6	4
[ ]	Google APIs	Google Inc.	1.6	4
[ ]	Google APIs	Google Inc.	1.6	4
[ ]	Android 2.0	Android Open Source Project	2.0	5
[ ]	Google APIs	Google Inc.	2.0	5
[ ]	Android 2.0.1	Android Open Source Project	2.0.1	6
[ ]	Google APIs	Google Inc.	2.0.1	6
[ ]	Android 2.1-upda...	Android Open Source Project	2.1-upd...	7
[ ]	Google APIs	Google Inc.	2.1-upd...	7
[✓]	Android 2.2	Android Open Source Project	2.2	8
[ ]	Google APIs	Google Inc.	2.2	8

Standard Android platform 2.2

Properties

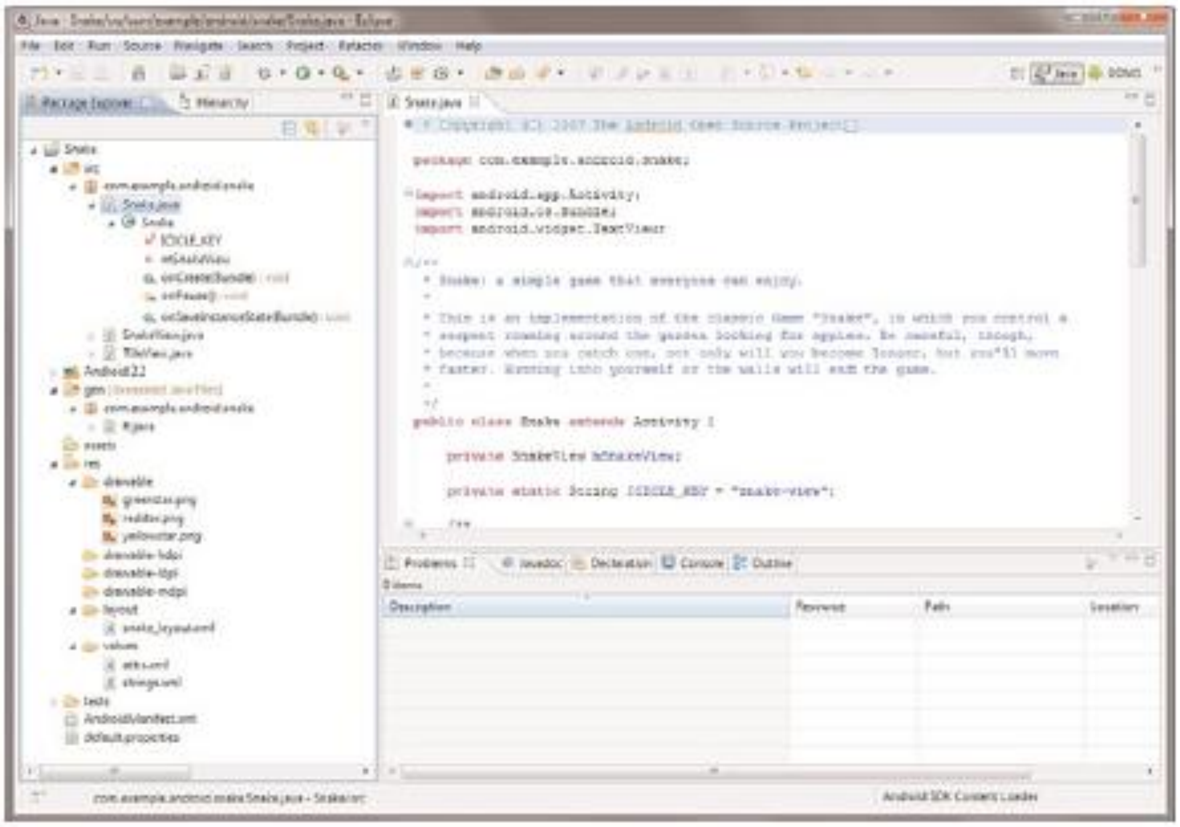
Application name: Snake

Package name: com.example.android.snake

[ ] Create Activity: .Snake

Min SDK Version:

The Snake project files.



```
/* Copyright (C) 2007 The Android Open Source Project

package com.example.android.snake;

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

/**
 * Snake: a simple game that everyone can enjoy.
 *
 * This is an implementation of the classic Game "Snake", in which you control a
 * serpent roaming around the garden looking for apples. Be careful, though,
 * because when you catch one, not only will you become longer, but you'll move
 * faster. Running into yourself or the walls will end the game.
 *
 */
public class Snake extends Activity {

    private SnakeView mSnakeView;

    private static String ICICLE_KEY = "snake-view";
```

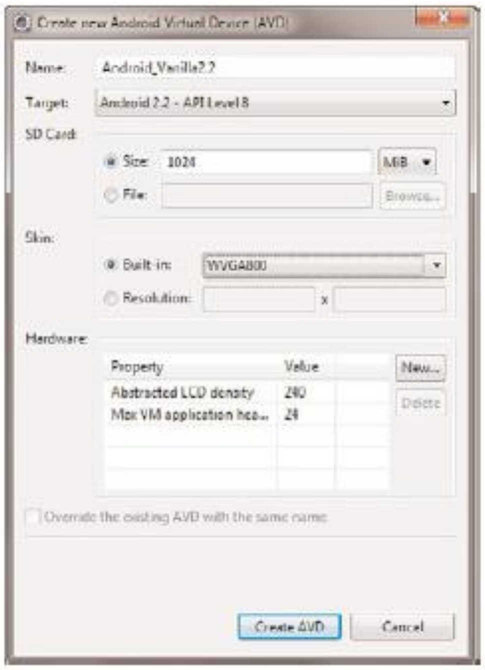
For the purposes of this example, an AVD for the default installation of Android 2.2 suffices. Here are the steps to create a basic AVD:

1. Launch the Android Virtual Device Manager from within Eclipse by clicking the little Android icon with the downward arrow on the toolbar. If you cannot find the icon, you can also launch the manager through the Window menu of Eclipse.
2. On the Virtual Devices menu, click the New button.
3. Choose a name for your AVD. Because we are going to take all the defaults, give this AVD a name of Android_Vanilla2.2.

4. Choose a build target. We want a basic Android 2.2 device, so choose Android 2.2 from the drop-down menu.
5. Choose an SD card capacity. This can be in kilobytes or megabytes and takes up space on your hard drive. Choose something reasonable, such as 1 gigabyte (1024M). If you're low on drive space or you know you won't need to test external storage to the SD card, you can use a much smaller value, such as 64 megabytes.
6. Choose a skin. This option controls the different resolutions of the emulator. In this case, use the WVGA800 screen. This skin most directly correlates to the popular Android handsets, such as the HTC Nexus One and the Evo 4G, both of which are currently sitting on my desk.
7. Click the Create AVD button, and wait for the operation to complete.
8. Click Finish. Because the AVD manager formats the memory allocated for SD card images, creating AVDs with SD cards could take a few moments.

Creating a Launch Configuration for Your Snake Project

Next, you must create a launch configuration in Eclipse to configure under what circumstances the Snake application builds and launches. The launch configuration is where you configure the emulator options to use and the entry point for your application.


Create new Android Virtual Device (AVD)

Name: Android_Vanilla2.2
Target: Android 2.2 - API Level 8
SD Card:
(●) Size: 1024 MiB
( ) File:
Skin:
(●) Built-in: WVGA800
( ) Resolution: x
Hardware:

Property	Value
Abstracted LCD density	240
Max VM application hea...	24

☐ Override the existing AVD with the same name

Creating a new AVD in Eclipse.

You can create Run Configurations and Debug Configurations separately, each with different options. These configurations are created under the Run menu in Eclipse (Run, Run Configurations and Run, Debug Configurations).

Follow these steps to create a basic Run Configuration for the Snake application:

1. Choose Run, Run Configurations (or right-click the Project and choose Run As).
2. Double-click Android Application.
3. Name your Run Configuration SnakeRunConfiguration
4. Choose the project by clicking the Browse button and choosing the Snake project.
5. Switch to the Target tab and, from the preferred AVD list, choose the Android_Vanilla2.2 AVD created earlier, as shown in Figure.

You can set other options on the Target and Common tabs, but for now we are leaving the defaults as they are.

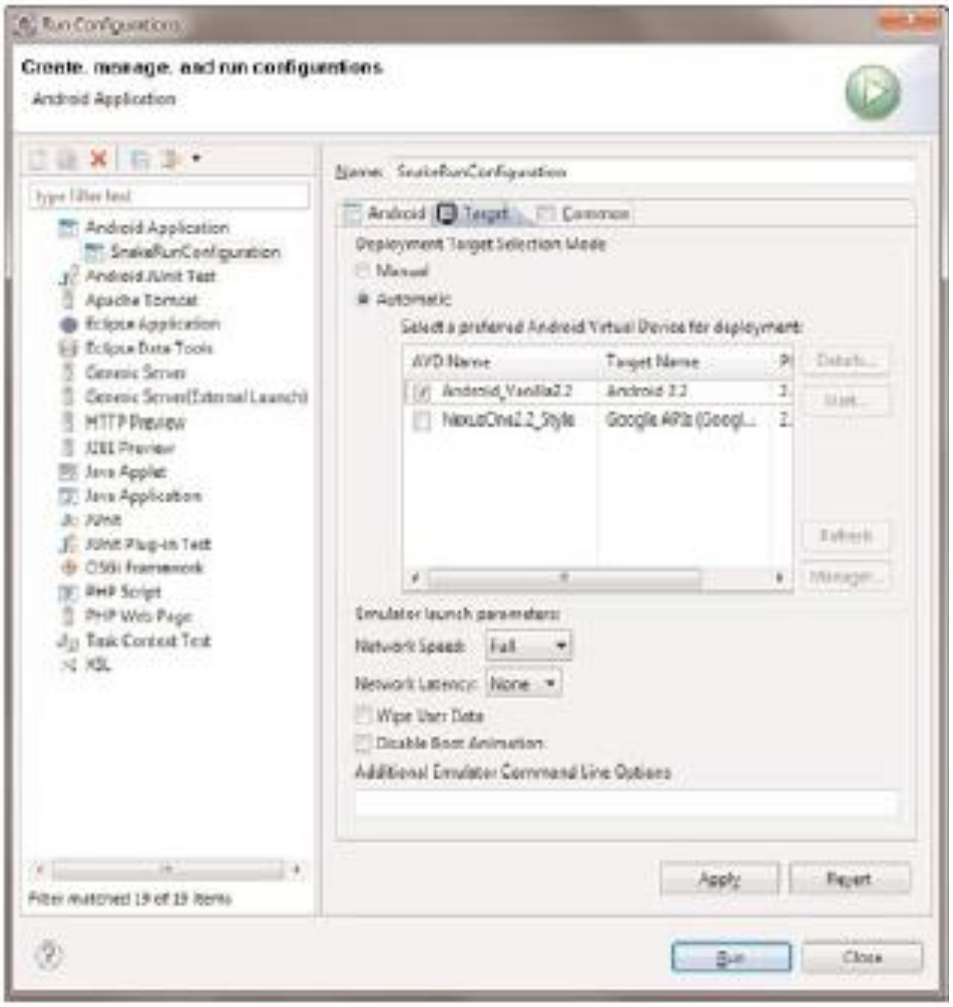
Running the Snake Application in the Android Emulator

Now you can run the Snake application using the following steps:

1. Choose the Run As icon drop-down menu on the toolbar (the green circle with the triangle).
2. Pull the drop-down menu and choose the SnakeRunConfiguration you created.
3. The Android emulator starts up; this might take a moment.
4. If necessary, swipe the screen from left to right to unlock the emulator, as shown in Figure.
5. The Snake application now starts, as shown in Figure.

You can interact with the Snake application through the emulator and play the game. You can also launch the Snake application from the Application drawer at any time by clicking on its application icon.

The Snake project launch configuration, Target tab with the AVD selected.



Create, manage, and run configurations
Android Application

type filter text
- Android Application
  - SnakeRunConfiguration
- Android JUnit Test
- Apache Tomcat
- Eclipse Application
- Eclipse Data Tools
- Generic Server
- Generic Server(External Launch)
- HTTP Preview
- J2EE Preview
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- OSGi Framework
- PHP Script
- PHP Web Page
- Task Context Test
- XSL

Filter matched 19 of 19 items

Name: SnakeRunConfiguration

Android | Target | Common

Deployment Target Selection Mode
- Manual
- [✓ Automatic]

Select a preferred Android Virtual Device for deployment:

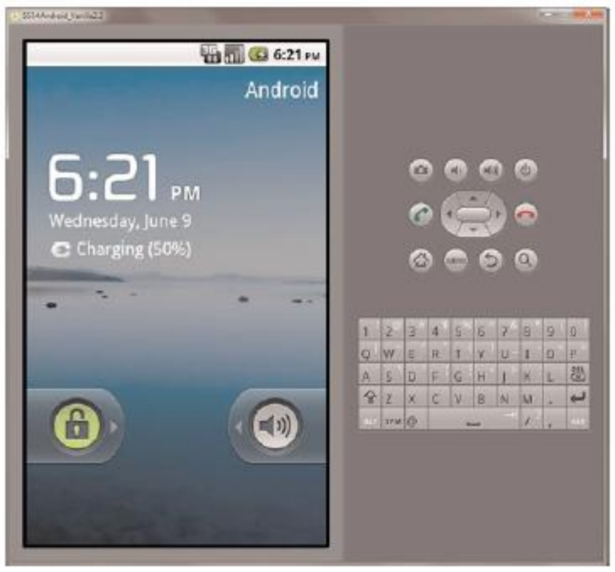
	AVD Name	Target Name	Pl
✓	Android_Vanilla2.2	Android 2.2	2.
	NexusOne2.2_Style	Google APIs (Googl...	2.

Emulator launch parameters:
Network Speed: Full
Network Latency: None
Wipe User Data
Disable Boot Animation
Additional Emulator Command Line Options

Building Your First Android Application

Now it’s time to write your first Android application. You start with a simple Hello World project and build upon it to explore some of the features of Android.

The Android emulator launching (locked).



The first thing you need to do is create a new project in your Eclipse workspace. The Android Project Wizard creates all the required files for an Android application. Follow these steps within Eclipse to create a new project:

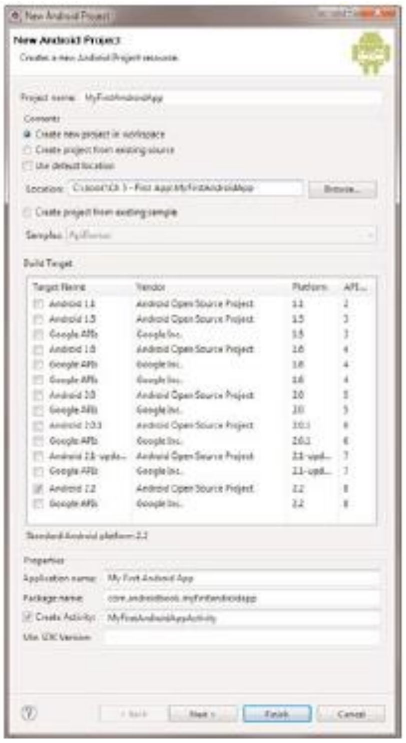
1. Choose File, New, Android Project, or choose the Android Project creator icon, which looks like a folder (with the letter *a* and a plus sign), on the Eclipse toolbar.
2. Choose a Project Name. In this case, name the project MyFirstAndroidApp.
3. Choose a Location for the project files. Because this is a new project, select the Create New Project in Workspace radio button. Check the Use Default Location checkbox or change the directory to wherever you want to store the source files.
4. Select a build target for your application. Choose a target that is compatible with the Android handsets you have in your possession. For this example, you might use the Android 2.2 target.

5. Choose an application name. The application name is the “friendly” name of the application and the name shown with the icon on the application launcher. In this case, the Application Name is My First Android App.
6. Choose a package name. Here you should follow standard package namespace conventions for Java. Because all our code examples in this book fall under the com.androidbook.* namespace, we will use the package name, but you are free to choose your own com.androidbook.myfirstandroidapp package name.
7. Check the Create Activity checkbox. This instructs the wizard to create a default launch activity for the application. Call this Activity class. My First Android App Activity
8. Finally, click the Finish button.

Core Files and Directories of the Android Application

Every Android application has a set of core files that are created and are used to define the functionality of the application. The following files are created by default with a new Android application.

Configuring My First Android App using the Android Project Wizard.



Project name: MyFirstAndroidApp

Contents
- [✓ Create new project in workspace]
- Create project from existing source
- Use default location

Location: [illegible]

- Create project from existing sample

Samples: [illegible]

Build Target

Target Name	Vendor	Platform	API...
Android 1.1	Android Open Source Project	1.1	2
Android 1.5	Android Open Source Project	1.5	3
Google APIs	Google Inc.	1.5	3
Android 1.6	Android Open Source Project	1.6	4
Google APIs	Google Inc.	1.6	4
Google APIs	Google Inc.	1.6	4
Android 2.0	Android Open Source Project	2.0	5
Google APIs	Google Inc.	2.0	5
Android 2.0.1	Android Open Source Project	2.0.1	6
Google APIs	Google Inc.	2.0.1	6
Android 2.1-upda...	Android Open Source Project	2.1-upd...	7
Google APIs	Google Inc.	2.1-upd...	7
✓ Android 2.2	Android Open Source Project	2.2	8
Google APIs	Google Inc.	2.2	8

Standard Android platform 2.2

Properties

Application name: My First Android App
Package name: com.androidbook.myfirstandroidapp
✓ Create Activity: MyFirstAndroidAppActivity
Min SDK Version:

There are a number of other files saved on disk as part of the Eclipse project in the workspace. However, the files included in Table are the important project files you will use on a regular basis.

Creating an AVD for Your Project

The next step is to create an AVD that describes what type of device you want to emulate when running the application. For this example, we can use the AVD we created for the Snake application. An AVD describes a device, not an application. Therefore, you can use the same AVD for multiple applications. You can also create similar AVDs with the same configuration, but different data (such as different applications installed and different SD card contents).

Important Android Project Files and Directories

Android File	General Description
`AndroidManifest.xml`	Global application description file. It defines your application's capabilities and permissions and how it runs.
`default.properties`	Automatically created project file. It defines your application's build target and other build system options, as required.
`src` Folder	Required folder where all source code for the application resides.
`src/com.androidbook.myfirst-androidapp/MyFirstAndroidApp-Activity.java`	Core source file that defines the entry point of your Android application.
`gen` Folder	Required folder where auto-generated resource files for the application reside.
`gen/com.androidbook.myfirst-androidapp/R.java`	Application resource management source file generated for you; it should not be edited.
`res` Folder	Required folder where all application resources are managed. Application resources include animations, drawable image assets, layout files, XML files, data resources such as strings, and raw files.
`res/drawable-*/icon.png`	Resource folders that store different resolutions of the application icon.
`res/layout/main.xml`	Single screen layout file.
`res/values/strings.xml`	Application string resources.
`assets` Folder	Folder where all application assets are stored. Application assets are pieces of application data (files, directories) that you do not want managed as application resources.

Creating Launch Configurations for Your Project

Next, you must create a Run and Debug launch configuration in Eclipse to configure the circumstances under which the MyFirstAndroidApp application builds and launches. The launch configuration is where you configure the emulator options to use and the entry point for your application.

You can create Run Configurations and Debug Configurations separately, with different options for each. Begin by creating a Run Configuration for the application.

Follow these steps to create a basic Run Configuration for the My First Android App application:

1. Choose Run, Run Configurations (or right-click the Project and Choose Run As).
2. Double-click Android Application.
3. Name your Run Configuration MyFirstAndroidAppRunConfig
4. Choose the project by clicking the Browse button and choosing the MyFirstAn- droidApp project.
5. Switch to the Target tab and set the Device Target Selection Mode to Manual.

Now create a Debug Configuration for the application. This process is similar to creating a Run Configuration. Follow these steps to create a basic Debug Configuration for the MyFirstAndroidApp application:

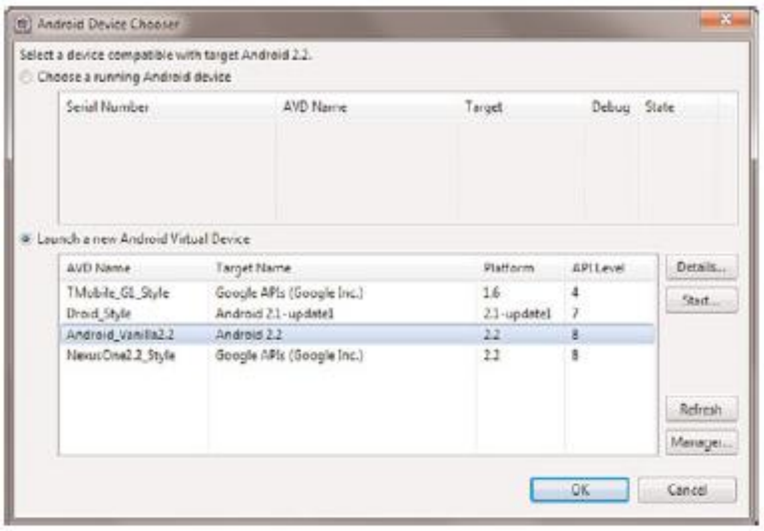
1. Choose Run, Debug Configurations (or right-click the Project and Choose Debug As).
2. Double-click Android Application.
3. Name your Debug Configuration MyFirstAndroidAppDebugConfig
4. Choose the project by clicking the Browse button and choosing the My First Android App project.

5. Switch to the Target tab and set the Device Target Selection Mode to Manual.
6. Click Apply, and then click Close.

You now have a Debug Configuration for your application.

Running Your Android Application in the Emulator

Now you can run the MyFirstAndroidApp application using the following steps:


Android Device Chooser

Select a device compatible with target Android 2.2.

( ) Choose a running Android device

Serial Number	AVD Name	Target	Debug	State

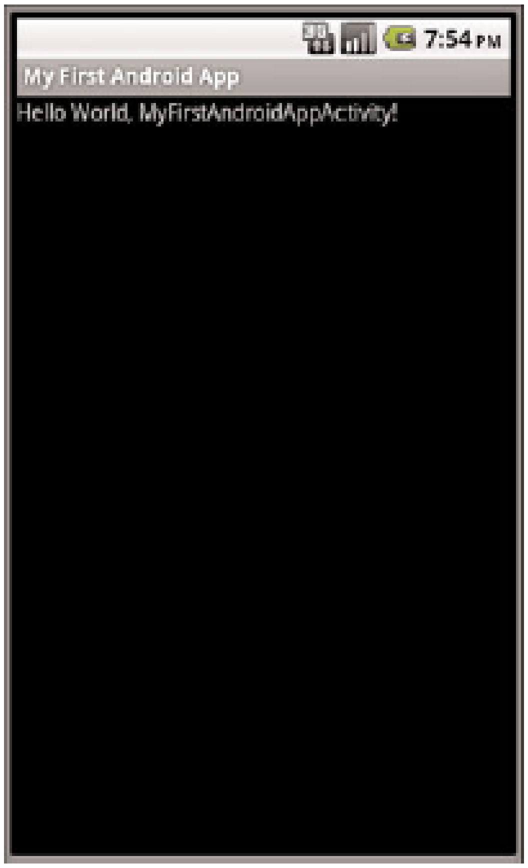
(•) Launch a new Android Virtual Device

AVD Name	Target Name	Platform	API Level
TMobile_G1_Style	Google APIs (Google Inc.)	1.6	4
Droid_Style	Android 2.1-update1	2.1-update1	7
Android_Vanilla2.2	Android 2.2	2.2	8
NexusOne2.2_Style	Google APIs (Google Inc.)	2.2	8

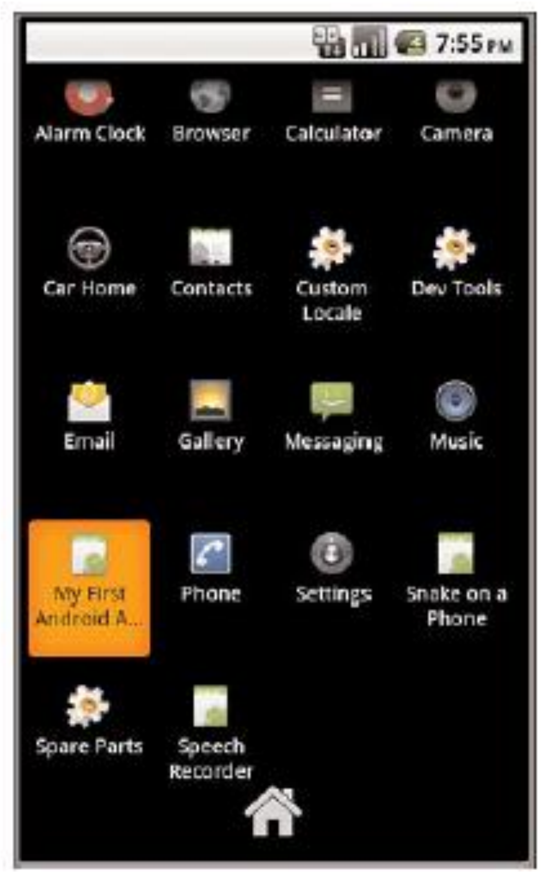
1. Choose the Run As icon drop-down menu on the toolbar (the little green circle with the play button and a drop-down arrow).
2. Pull the drop-down menu and choose the Run Configuration you created. (If you do not see it listed, choose the Run Configurations... item and select the appropriate configuration. The Run Configuration shows up on this drop-down list the next time you run the configuration.)
3. Because you chose the Manual Target Selection mode, you are now prompted for your emulator instance. Change the selection to start a new emulator instance, and check the box next to the AVD you created, as shown in Figure.
4. The Android emulator starts up, which might take a moment.
5. Press the Menu button to unlock the emulator.
6. The application starts, as shown in Figure.
7. Click the Home button in the Emulator to end the application.

8. Pull up the Application Drawer to see installed applications. Your screen looks something like Figure.
9. Click on the My First Android Application icon to launch the application again.

My First Android App running in the emulator.



Hello World, MyFirstAndroidAppActivity!



Debugging Your Android Application in the Emulator

Before we go any further, you need to become familiar with debugging in the emulator. To illustrate some useful debugging tools, let's manufacture an error in the My First Android Application.

In your project, edit the file MyFirstAndroidApp.java and create a new method called forceError() in your class and make a call to this method in your onCreate() method.

The forceError() method forces a new unhandled error in your application.

The forceError() method should look something like this:

```
public void forceError()  {
       if(true) {
              throw new Error("Whoops");
      }
}
```

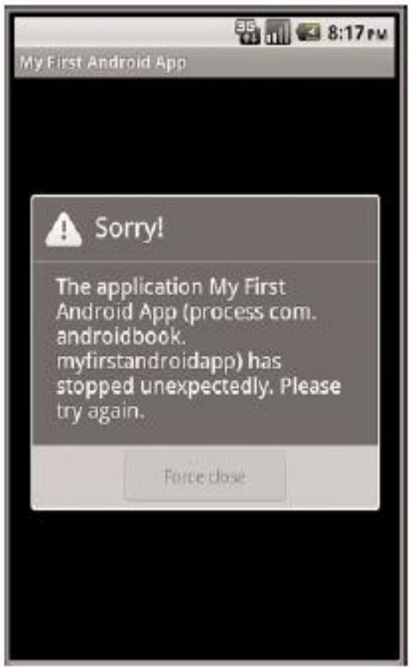
It's probably helpful at this point to run the application and watch what happens. Do this using the Run Configuration first. In the emulator, you see that the application has stopped unexpectedly. You are prompted by a dialog that enables you to forcefully close the application, as shown in Figure.

Shut down the application and the emulator. Now it's time to debug. You can debug the MyFirstAndroidApp application using the following steps:

1. Choose the Debug As icon drop-down menu on the toolbar (the little green bug with the drop-down arrow) .
2. Pull the drop-down menu and choose the Debug Configuration you created. (If you do not see it listed, choose the Debug Configurations... item and select the appropriate configuration. The Debug Configuration shows up on this drop-down list the next time you run the configuration.)
3. Continue as you did with the Run Configuration and choose the appropriate AVD and launch the emulator again, unlocking it if needed.

It takes a moment for the emulator to start up and for the debugger to attach. If this is the first time you've debugged an Android application, you need to click through some dialog boxes, such as the one shown in Figure, the first time your application attaches to the debugger.

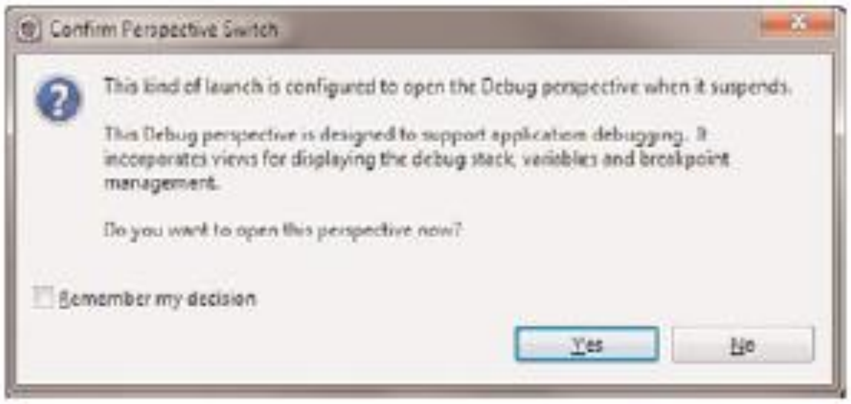
My First Android App crashing gracefully.



Sorry!

The application My First Android App (process com.androidbook.myfirstandroidapp) has stopped unexpectedly. Please try again.

Switching debug perspectives for Android emulator debugging.



Confirm Perspective Switch

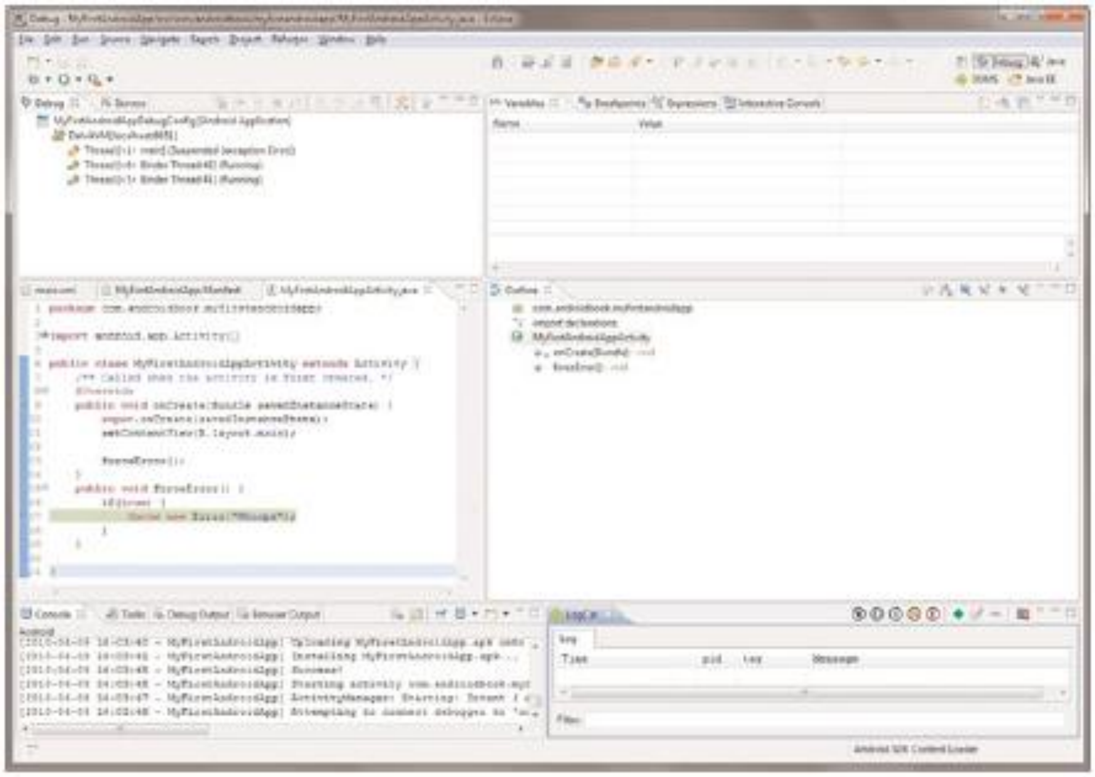
This kind of launch is configured to open the Debug perspective when it suspends.

This Debug perspective is designed to support application debugging. It incorporates views for displaying the debug stack, variables and breakpoint management.

Do you want to open this perspective now?

In Eclipse, use the Debug perspective to set breakpoints, step through code, and watch the LogCat logging information about your application. This time, when the application fails, you can determine the cause using the debugger. You might need to click through several dialogs as you set up to debug within Eclipse. If you allow the application to continue after throwing the exception, you can examine the results in the Debug perspective of Eclipse. If you examine the LogCat logging pane, you see that your application was forced to exit due to an unhandled exception.

Debugging My First Android App in Eclipse.



```
package com.androidbook.myfirstandroidapp;

import android.app.Activity;

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

        forceError();
    }
    public void forceError() {
        if(true) {
            throw new Error("Whoops");
        }
    }
}
```

Specifically, there's a red AndroidRuntime error: java.lang.Error: Whoops Back in the emulator, click the Force Close button. Now set a breakpoint on the forceError() method by right-clicking on the left side of the line of code and choosing Toggle Breakpoint (or Ctrl+Shift+B).

In the emulator, restart your application and step through your code. You see My First Android App has thrown the exception and then the exception shows up in the Variable Browser pane of the Debug Perspective. Expanding the variables contents shows that it is the "Whoops" error.

This is a great time to crash your application repeatedly and get used to the controls. While you're at it, switch over to the DDMS perspective. You note the emulator has a list of processes running on the phone, such as system_process and com.android.phone. If you launch MyFirstAndroidApp, you see com. androidbook .myfirs tandroid app show up as a process on the emulator listing. Force the app to close because it crashes, and you note that it disappears from the process list. You can use DDMS to kill processes, inspect threads and the heap, and access the phone file system.

Adding Logging Support to Your Android Application

Before you start diving into the various features of the Android SDK, you should familiarize yourself with logging, a valuable resource for debugging and learning Android. Android logging features are in the Log class of the android.util package.

Some helpful methods in the android.util.Log class are shown in Table.

Method	Purpose
`Log.e()`	Log errors
`Log.w()`	Log warnings
`Log.i()`	Log informational messages
`Log.d()`	Log Debug messages
`Log.v()`	Log Verbose mesages

To add logging support to MyFirstAndroidApp, edit the file MyFirst AndroidApp. java. First, you must add the appropriate import statement for the log class:

import android.util.Log;

Next, within the MyFirstAndroidApp class, declare a constant string that you use to tag all logging messages from this class. You can use the LogCat utility within Eclipse to filter your logging messages to this debug tag:

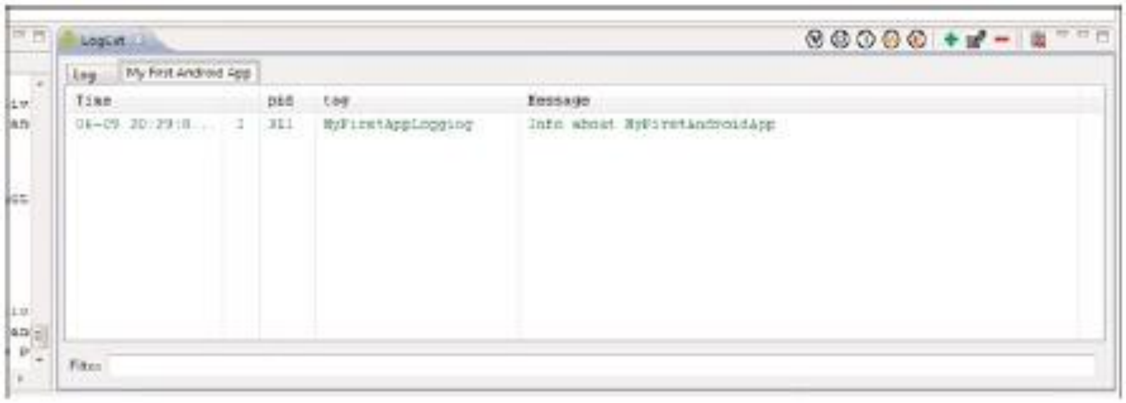
private static final String DEBUG_TAG= “MyFirstAppLogging”;

Now, within the onCreate() method, you can log something informational:

Log.i(DEBUG_TAG, “Info about MyFirstAndroidApp”);

Now you’re ready to run MyFirstAndroidApp. Save your work and debug it in the emulator. You notice that your logging messages appear in the LogCat listing, with the Tag field MyFirstAppLogging.

A Filtered LogCat log for My First Android App.



```
Time               pid   tag                 Message
06-09 20:29:0...  I  311   MyFirstAppLogging   Info about MyFirstAndroidApp
```

Adding Some Media Support to Your Application

Next, let’s add some pizzazz to MyFirstAndroidApp by having the application play an MP3 music file. Android media player features are found in the MediaPlayer

class of the android.media package.

You can create MediaPlayer objects from existing application resources or by specifying a target file using a Uniform Resource Identifier (URI). For simplicity, we begin by accessing an MP3 using the Uri class from the android.net package.

Some methods in the android.media.MediaPlayer and android.net.Uri classes are shown in Table.

Important MediaPlayer and URI Parsing Methods

Method	Purpose
`MediaPlayer.create()`	Creates a new Media Player with a given target to play
`MediaPlayer.start()`	Starts media playback
`MediaPlayer.stop()`	Stops media playback
`MediaPlayer.release()`	Releases the resources of the Media Player object
`Uri.parse()`	Instantiates a `Uri` object from an appropriately formatted URI address

To add MP3 playback support to MyFirstAndroidApp, edit the file MyFirst AndroidApp .java. First, you must add the appropriate import statements for the MediaPlayer class.

```
import android.media.MediaPlayer;
import android.net.Uri;
```

Next, within the MyFirstAndroidApp class, declare a member variable for your MediaPlayer object.

```
private MediaPlayer mp;
```

Now, create a new method called playMusicFromWeb() in your class and make a call to this method in your onCreate() method.The playMusicFromWeb() method creates a valid Uri object, creates a MediaPlayer object, and starts the MP3 playing. If the operation should fail for some reason, the method logs a custom error with your logging tag.

The playMusicFromWeb() method should look something like this:

```
public void playMusicFromWeb() {
     try {
          Uri file =
                Uri.parse("http://www.perlgurl.org/podcast/archives"
                + "/podcasts/PerlgurlPromo.mp3");
                mp = MediaPlayer.create(this, file);
                   mp.start();
     }
     catch (Exception e) {
            Log.e(DEBUG_TAG, "Player failed", e);
     }
 }
```

And finally, you want to cleanly exit when the application shuts down.To do this, you need to override the onStop() method and stop the MediaPlayer object and release its resources.

The onStop() method should look something like this:

```
protected void onStop()  {
     if (mp != null) {
            mp.stop();
            mp.release();
     }
     super.onStop();
}
```

Now, if you run MyFirstAndroidApp in the emulator (and you have an Internet connection to grab the data found at the URI location), your application plays the MP3.When you shut down the application, the MediaPlayer is stopped and released appropriately.

Adding Location-Based Services to Your Application

Your application knows how to say Hello, but it doesn't know where it's located. Now is a good time to become familiar with some simple location-based calls to get the GPS coordinates.

Creating an AVD with Google APIs

To have some fun with location-based services and maps integration, you should use some of the Google applications often available on Android handsets—most notably, the Google Maps application. Therefore, you must create another AVD. This AVD should have exactly the same settings as the Android_Vanilla2.2 AVD, with one exception: Its Target should be the Google APIs equivalent for that API level (which is 8).You can call this AVD Android_ with_ GoogleAPIs_2.2.

Configuring the Location of the Emulator

After you have created a new AVD with the Google APIs support, you need to shut down the emulator you've been running. Then debug the My First Android App application again, this time choosing the new AVD.

The emulator does not have location sensors, so the first thing you need to do is seed your emulator with GPS coordinates. To do this, launch your emulator in debug mode with an AVD supporting the Google Maps add-ins and follow these steps:

In the Emulator:

1. Press the Home key to return to the Home screen.
2. Launch the Maps application from the Application drawer.
3. Click the Menu button.
4. Choose the My Location menu item. (It looks like a target.)
5. Click the DDMS perspective in the top-right corner of Eclipse.
6. You see an Emulator Control pane on the left side of the screen. Scroll down to the Location Control.

7. Manually enter the longitude and latitude of your location. (Note they are in reverse order.)
8. Click Send.

Back in the emulator, notice that the Google Map now shows the location you seeded. Your screen should now display your location as Yosemite Valley, as shown in Figure.

Your emulator now has a simulated location.

Finding the Last Known Location

To add location support to MyFirstAndroidApp, edit the file My First AndroidApp .java. First, you must add the appropriate import statements:

```
import android.location.Location;
import android.location.LocationManager;
```

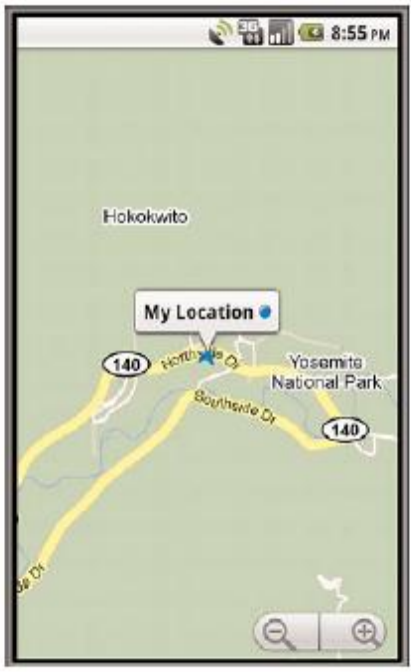
Now, create a new method called getLocation() in your class and make a call to this method in your onCreate() method. The getLocation() method gets the last known location on the phone and logs it as an informational message. If the operation fails for some reason, the method logs an error.

The getLocation() method should look something like this:

```
public void getLocation() {
     try {
            LocationManager locMgr = (LocationManager)
            getSystemService(LOCATION_SERVICE);
            Location recentLoc = locMgr.
            getLastKnownLocation(LocationManager.GPS_PROVIDER);
            Log.i(DEBUG_TAG,  "loc: " + recentLoc.toString());
     }
     catch (Exception e) {
            Log.e(DEBUG_TAG, "Location failed", e);
     }
}
```

Setting the location of the emulator to Yosemite Valley.

Finally, your application requires special permissions to access location-based functionality. You must register this permission in your Android Manifest.xml file. To add location-based service permissions to your application, perform the following steps:



1. Double-click the AndroidManifest.xml file.
2. Switch to the Permissions tab.
3. Click the Add button and choose Uses Permission.
4. In the right pane, select android.permission.ACCESS_FINE_LOCATION
5. Save the file.

Now, if you run My First Android App in the emulator, your application logs the GPS coordinates you provided to the emulator as an informational message, viewable in the LogCat pane of Eclipse. Debugging Your Application on the Hardware

You mastered running applications in the emulator. Now let's put the application on real hardware. First, you must register your application as Debuggable in your Android Manifest .xml file. To do this, perform the following steps:

1. Double-click the AndroidManifest.xml file.
2. Change to the Application tab.
3. Set the Debuggable Application Attribute to True.
4. Save the file.

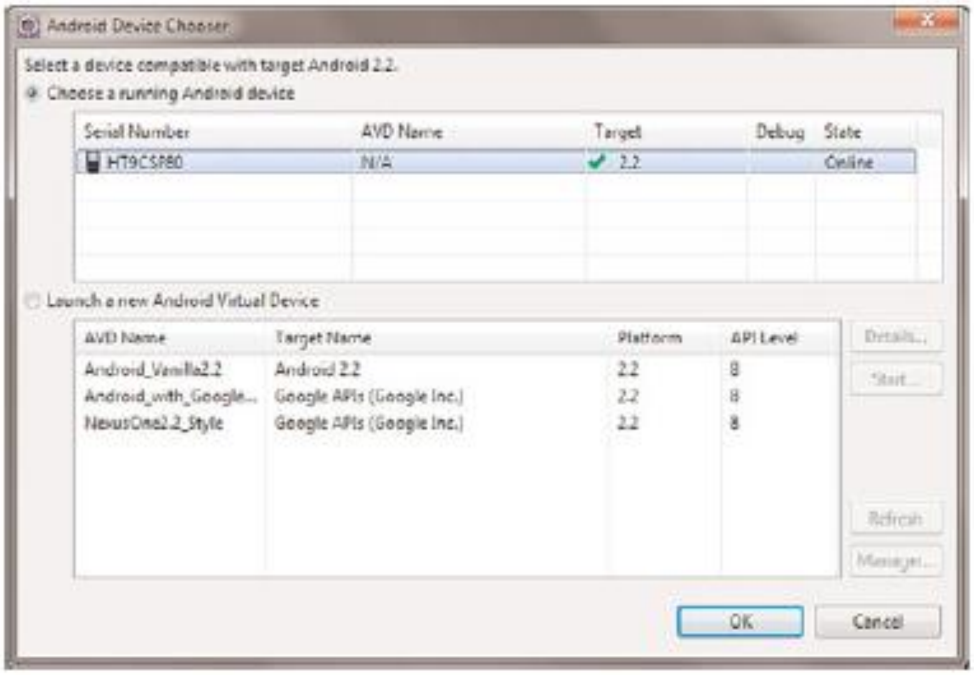
You can also modify the application element of the AndroidManifest.xml file directly with the attribute, as shown here:

android:debuggable <application ... android:debuggable="true">

If you forget to set the debuggable attribute to true, the handset shows the dialog for waiting for the debugger to connect until you choose Force Close and update the manifest file.

Now, connect an Android device to your computer via USB and re-launch the Run Configuration or Debug Configuration of the application. Because you chose Manual mode for the configuration, you should now see a real Android device listed as an option in the Android Device Chooser.

Android Device Chooser with USB-connected Android handset.



Android Device Chooser

Select a device compatible with target Android 2.2.

[✓ Choose a running Android device]

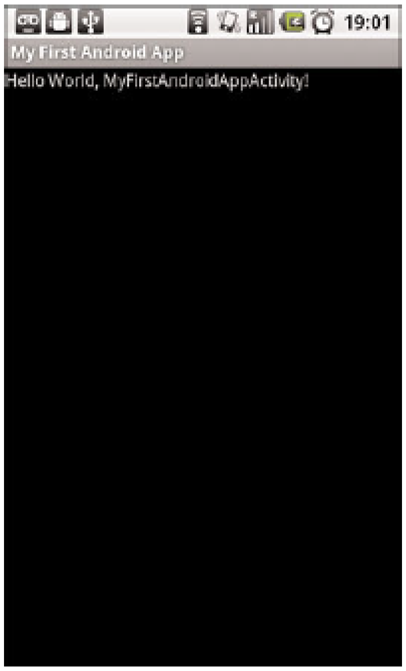
Serial Number	AVD Name	Target	Debug	State
HT9CSP80	N/A	✓ 2.2		Online

Launch a new Android Virtual Device

AVD Name	Target Name	Platform	API Level
Android_Vanilla2.2	Android 2.2	2.2	8
Android_with_Google...	Google APIs (Google Inc.)	2.2	8
NexusOne2.2_Style	Google APIs (Google Inc.)	2.2	8

Choose the Android Device as your target, and you see that the My First Android App application gets loaded onto the Android handset and launched, just as before. Provided you have enabled the development debugging options on the handset, you can debug the application here as well. You can tell the handset is actively using a USB debugging connection, because there is a little Android bug-like icon in the notification bar. A screenshot of the application running on a real handset is shown in Figure.

My First Android App running on Android device hardware.



Hello World, MyFirstAndroidAppActivity!

Debugging on the handset is much the same as debugging on the emulator, but with a couple of exceptions. You cannot use the emulator controls to do things such as send an SMS or configure the location to the device, but you can perform real actions (true SMS, actual location data) instead.

Understanding the Anatomy of an Android Application

Understanding the Anatomy of an Android Application

Classical computer science classes often define a program in terms of functionality and data, and Android applications are no different. They perform tasks, display information to the screen, and act upon data from a variety of sources.

Developing Android applications for mobile devices with limited resources requires a thorough understanding of the application lifecycle. Android also uses its own terminology for these application building blocks—terms such as Context, Activity, and Intent. This chapter familiarizes you with the most important components of Android applications.

Mastering Important Android Terminology

This chapter introduces you to the terminology used in Android application development and provides you with a more thorough understanding of how Android applications function and interact with one another. Some of the important terms covered in this chapter are

- **Context:** The context is the central command center for an Android application. All application-specific functionality can be accessed through the context.
- **Activity:** An Android application is a collection of tasks, each of which is called an Activity. Each Activity within an application has a unique task or purpose.
- **Intent:** The Android operating system uses an asynchronous messaging mechanism to match task requests with the appropriate Activity. Each

request is packaged as Intent. You can think of each such request as a message stating intent to *do* something.

- **Service:** Tasks that do not require user interaction can be encapsulated in a service. A service is most useful when the operations are lengthy (offloading time-consuming processing) or need to be done regularly (such as checking a server for new mail).

Using the Application Context

The application Context is the central location for all top-level application functionality. The Context class can be used to manage application-specific configuration details as well as application-wide operations and data. Use the application Context to access settings and resources shared across multiple Activity instances.

Retrieving the Application Context

You can retrieve the Context for the current process using the getApplicationContext() method, like this:

```
Context context = getApplicationContext();
```

Using the Application Context

After you have retrieved a valid application Context, it can be used to access application wide features and services.

Retrieving Application Resources

You can retrieve application resources using the getResources() method of the application Context. The most straightforward way to retrieve a resource is by using its resource identifier, a unique number automatically generated within the R.java class. The following example retrieves a String instance from the application resources by its resource ID:

```
String greeting = getResources().getString(R.string.hello);
```

We talk more about application resources in Chapter "Managing Application Resources."

Accessing Application Preferences

You can retrieve shared application preferences using the getSharedPreferences() method of the application Context.The SharedPreferences class can be used to save simple application data, such as configuration settings. We talk more about application preferences in Chapter "Using Android Data and Storage APIs."

Accessing Other Application Functionality Using Context

The application Context provides access to a number of other top-level application features. Here are a few more things you can do with the application Context:

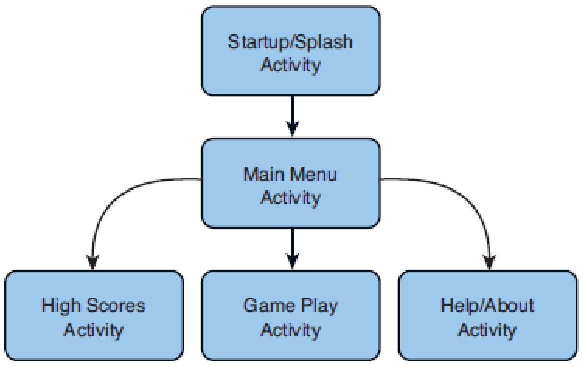
- Launch Activity instances
- Retrieve assets packaged with the application
- Request a system service (for example, location service)
- Manage private application files, directories, and databases
- Inspect and enforce application permissions

The first item on this list—launching Activity instances—is perhaps the most common reason you use the application Context.

Performing Application Tasks with Activities

The Android Activity class (android.app.Activity) is core to any Android application. Much of the time, you define and implement an Activity class for each screen in your application. For example, a simple game application might have the following five Activities, as shown in Figure:

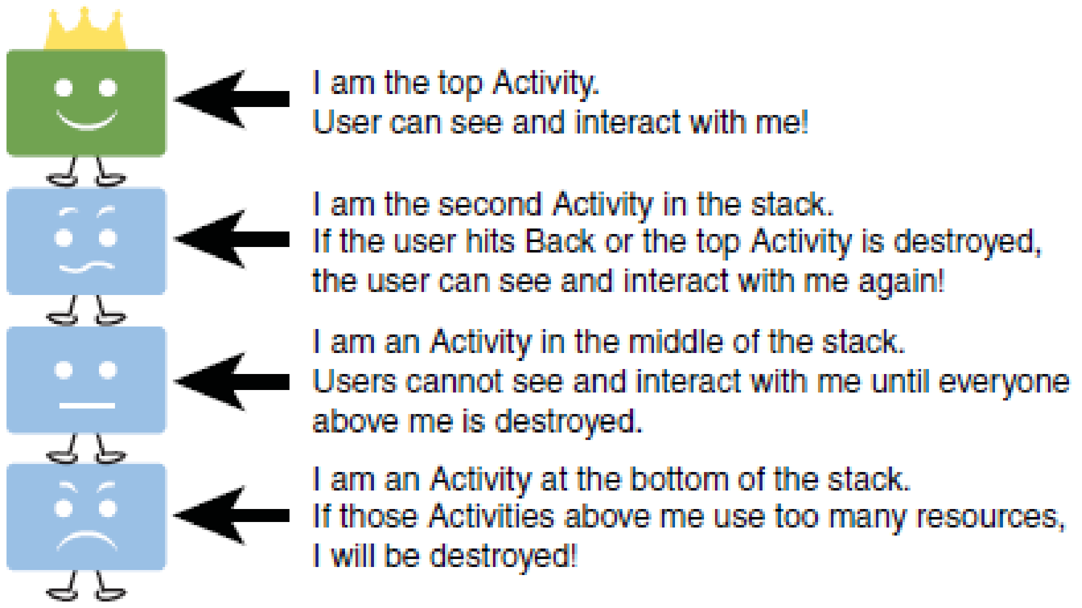
- **A Startup or Splash screen:** This activity serves as the primary entry point to the application. It displays the application name and version information and transitions to the Main menu after a short interval.
- **A Main Menu screen:** This activity acts as a switch to drive the user to the core Activities of the application. Here the users must choose what they want to do within the application.
- **A Game Play screen:** This activity is where the core game play occurs.
- **A High Scores screen:** This activity might display game scores or settings.
- **A Help/About screen:** This activity might display the information the user might need to play the game.



The Lifecycle of an Android Activity

Android applications can be multi-process, and the Android operating system allows multiple applications to run concurrently, provided memory and processing power is available. Applications can have background processes, and applications can be interrupted and paused when events such as phone calls occur. There can be only one active application visible to the user at a time—specifically, a single application Activity is in the foreground at any given time.

The Android operating system keeps track of all Activity objects running by placing them on an Activity stack. When a new Activity starts, the Activity on the top of the stack (the current foreground Activity) pauses, and the new Activity pushes onto the top of the stack. When that Activity finishes, that Activity is removed from the activity stack, and the previous Activity in the stack resumes.


I am the top Activity.
User can see and interact with me!
I am the second Activity in the stack.
If the user hits Back or the top Activity is destroyed, the user can see and interact with me again!
I am an Activity in the middle of the stack.
Users cannot see and interact with me until everyone above me is destroyed.
I am an Activity at the bottom of the stack.
If those Activities above me use too many resources, I will be destroyed!

Android applications are responsible for managing their state and their memory, resources, and data. They must pause and resume seamlessly. Understanding the different states within the Activity lifecycle is the first step to designing and developing robust Android applications.

Using Activity Callbacks to Manage Application State and Resources

Different important state changes within the Activity lifecycle are punctuated by a series of important method callbacks. These callbacks are shown in Figure.

Here are the method stubs for the most important callbacks of the Activity class:

```
public class MyActivity extends Activity {
     protected void onCreate(Bundle savedInstanceState);
     protected void onStart();
     protected void onRestart();
     protected void onResume();
     protected void onPause();
     protected void onStop();
     protected void onDestroy();
}
```

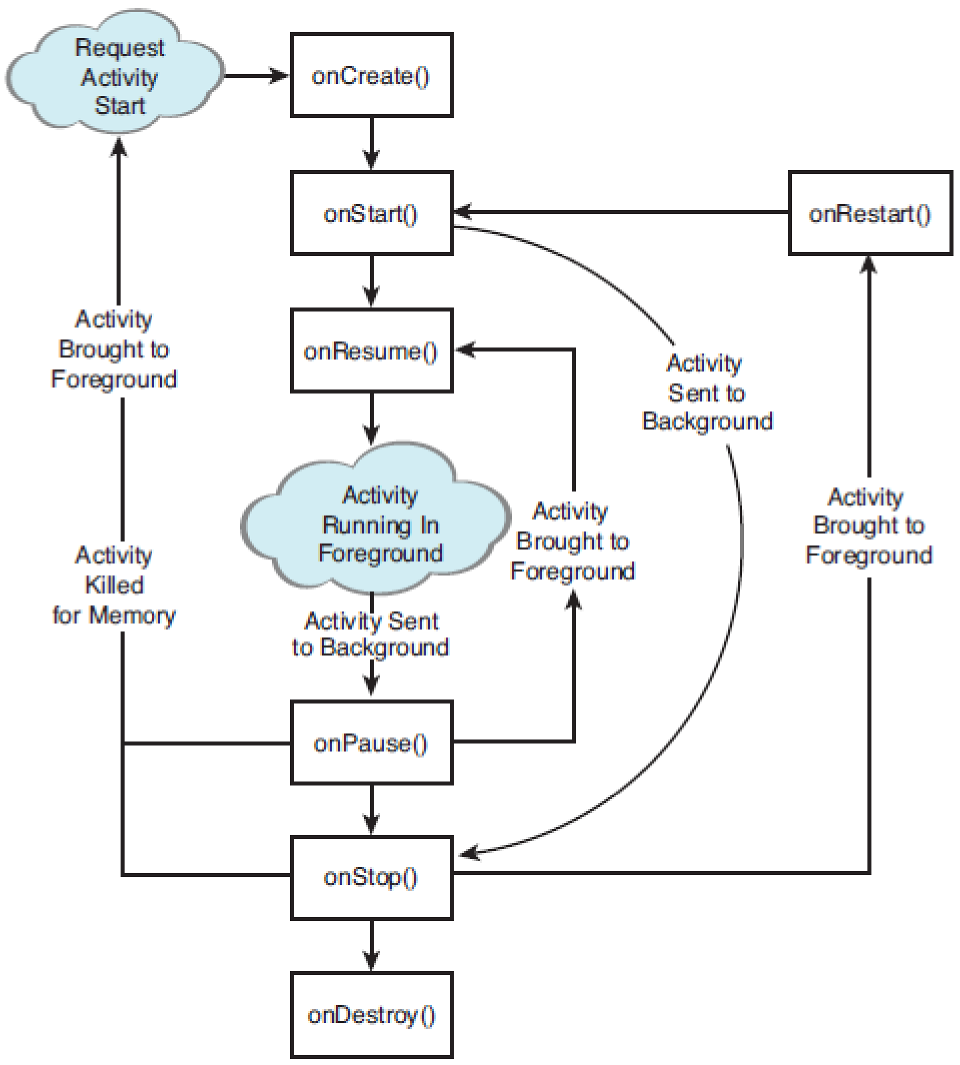
The lifecycle of an Android Activity.

Initializing Static Activity Data in onCreate()

When an Activity first starts, the onCreate() method is called. The onCreate() method has a single parameter, a Bundle, which is null if this is a newly started Activity. If this Activity was killed for memory reasons and is now restarted, the Bundle contains the previous state information for this Activity so that it can reinitiate. It is appropriate to perform any setup, such as layout and data binding, in the onCreate() method. This includes calls to the setContentView() method.

Initializing and Retrieving Activity Data in onResume()

When the Activity reaches the top of the activity stack and becomes the foreground process, the onResume() method is called.Although the Activity might not be visible yet to the user, this is the most appropriate place to retrieve any instances to resources (exclusive or otherwise) that the Activity needs to run. Often, these resources are the most process-intensive, so we only keep these around while the Activity is in the foreground.


Request Activity Start
onCreate()
onStart()
onRestart()
onResume()
Activity Brought to Foreground
Activity Sent to Background
Activity Running In Foreground
Activity Brought to Foreground
Activity Brought to Foreground
Activity Killed for Memory
Activity Sent to Background
onPause()
onStop()
onDestroy()

Stopping, Saving, and Releasing Activity Data in onPause()

When another Activity moves to the top of the activity stack, the current Activity is informed that it is being pushed down the activity stack by way of the onPause() method.

Here, the Activity should stop any audio, video, and animations it started in the onResume() method. This is also where you must deactivate resources such as database Cursor objects if you have opted to manage them manually, as opposed to having them managed automatically.

The onPause() method can also be the last chance for the Activity to clean up and release any resources it does not need while in the background. You need to save any uncommitted data here, in case your application does not resume.

The Activity can also save state information to Activity-specific preferences, or application wide preferences.

The Activity needs to perform anything in the onPause() method quickly. The new fore ground Activity is not started until the onPause() method returns.

Avoiding Activity Objects Being Killed

Under low-memory conditions, the Android operating system can kill the process for any Activity that has been paused, stopped, or destroyed. This essentially means that any Activity not in the foreground is subject to a possible shutdown.

If the Activity is killed after onPause(), the onStop() and onDestroy() methods might not be called.The more resources released by an Activity in the onPause() method, the less likely the Activity is to be killed while in the background.

The act of killing an Activity does not remove it from the activity stack. Instead, the Activity state is saved into a Bundle object, assuming the Activity implements and uses on Save Instance State() for custom data, though some View data is automatically saved. When the user returns to the Activity later, the onCreate() method is called again, this time with a valid Bundle object as the parameter.

Saving Activity State into a Bundle with onSaveInstanceState()

If an Activity is vulnerable to being killed by the Android operating system due to low memory, the Activity can save state information to a Bundle object using the on Save Instance State() callback method. This call is not guaranteed under all circumstances, so use the onPause() method for essential data commits.

When this Activity is returned to later, this Bundle is passed into the onCreate() method, allowing the Activity to return to the exact state it was in when the Activity paused. You can also read Bundle information after the onStart() callback method using the on Restore Instance State() call back.

Destroy Static Activity Data in onDestroy()

When an Activity is being destroyed, the onDestroy() method is called. The onDestroy() method is called for one of two reasons: The Activity has completed its lifecycle voluntarily, or the Activity is being killed by the Android operating system because it needs the resources.

Managing Activity Transitions with Intents

In the course of the lifetime of an Android application, the user might transition between a number of different Activity instances. At times, there might be multiple Activity instances on the activity stack. Developers need to pay attention to the lifecycle of each Activity during these transitions.

Some Activity instances—such as the application splash/startup screen—are shown and then permanently discarded when the Main menu screen Activity takes over. The user cannot return to the splash screen Activity without re-launching the application.

Other Activity transitions are temporary, such as a child Activity displaying a dialog box, and then returning to the original Activity (which was paused on the activity stack and now resumes). In this case, the parent Activity launches the child Activity and expects a result.

Transitioning Between Activities with Intents

As previously mentioned,Android applications can have multiple entry points. There is no main() function, such as you find in iPhone development. Instead, a specific Activity can be designated as the main Activity to launch by default within the Android Manifest.xml file; we talk more about this file in Chapter “Defining Your Application Using the Android Manifest File.”

Other Activities might be designated to launch under specific circumstances. For example, a music application might designate a generic Activity to launch by default from the Application menu, but also define specific alternative entry point Activities for accessing specific music playlists by playlist ID or artists by name.

Launching a New Activity by Class Name

You can start activities in several ways. The simplest method is to use the Application Context object to call the startActivity() method, which takes a single parameter, an Intent.

An Intent (android.content.Intent) is an asynchronous message mechanism used by the Android operating system to match task requests with the appropriate Activity or Service (launching it, if necessary) and to dispatch broadcast Intents events to the system at large.

For now, though, we focus on Intents and how they are used with Activities. The following line of code calls the startActivity() method with an explicit Intent. This Intent requests the launch of the target Activity named My Draw Activity by its class. This class is implemented elsewhere within the package.

```
startActivity(new Intent(getApplicationContext(),
MyDrawActivity.class));
```

This line of code might be sufficient for some applications, which simply transition from one Activity to the next. However, you can use the Intent mechanism in a much more robust manner. For example, you can use the Intent structure to pass data between Activities.

Creating Intents with Action and Data

You’ve seen the simplest case to use an Intent to launch a class by name. Intents need not specify the component or class they want to launch explicitly. Instead, you can create an Intent Filter and register it within the Android Manifest file. The Android operating system attempts to resolve the Intent requirements and launch the appropriate Activity based on the filter criteria.

The guts of the Intent object are composed of two main parts: the *action* to be performed and the *data* to be acted upon.You can also specify action/data pairs using Intent Action types and Uri objects.As you saw in Chapter “Writing Your First Android Application,” a Uri object represents a string that gives the location and name of an object. Therefore, an Intent is basically saying “do this” (the action) to “that” (the Uri describing what resource to do the action to).

The most common action types are defined in the Intent class, including ACTION_MAIN (describes the main entry point of an Activity) and ACTION_EDIT (used in conjunction with a Uri to the data edited).You also find Action types that generate integration points with Activities in other applications, such as the Browser or Phone Dialer.

Launching an Activity Belonging to Another Application

Initially, your application might be starting only Activities defined within its own package. However, with the appropriate permissions, applications might also launch external Activities within other applications. For example, a Customer Relationship Management (CRM) application might launch the Contacts application to browse the Contact database, choose a specific contact, and return that Contact's unique identifier to the CRM application for use.

Here is an example of how to create a simple Intent with a predefined Action (ACTION _DIAL) to launch the Phone Dialer with a specific phone number to dial in the form of a simple Uri object:

```
Uri number = Uri.parse(tel:5555551212);
Intent dial = new Intent(Intent.ACTION_DIAL, number);
startActivity(dial);
```

You can find a list of commonly used Google application Intents. Also available is the developer managed Registry of Intents protocols at OpenIntents, which has a growing list of Intents available from third-party applications and those within the Android SDK.

Passing Additional Information Using Intents

You can also include additional data in Intent. The Extras property of Intent is stored in a Bundle object. The Intent class also has a number of helper methods for getting and setting name/value pairs for many common datatypes.

For example, the following Intent includes two extra pieces of information—a string value and a boolean:

```
Intent intent = new Intent(this, MyActivity.class);
intent.putExtra("SomeStringData","Foo");
intent.putExtra("SomeBooleanData",false);
```

Organizing Activities and Intents in Your Application Using Menus

As previously mentioned, your application likely has a number of screens, each with its own Activity. There is a close relationship between menus, Activities, and Intents. You often see a menu used in two different ways with Activities and Intents:

- **Main Menu:** Acts as a switch in which each menu item launches a different Activity in your application. For instance, menu items for launching the Play Game Activity, the High Scores Activity, and the Help Activity.
- **Drill-Down:** Acts as a directory in which each menu item launches the same Activity, but each item passes in different data as part of the Intent (for example, a menu of all database records). Choosing a specific item might launch the Edit Record Activity, passing in that particular item's unique identifier.

Working with Services

Trying to wrap your head around Activities, Intents, Intent Filters, and the lot when you start with Android development can be daunting. We have tried to distill everything you need to know to start writing Android applications with multiple Activity classes, but we'd be remiss if we didn't mention that there's a lot more here, much of which is discussed throughout the book using practical examples. However, we need to give you a "heads up" about some of these topics now because we talk about these concepts very soon when we cover configuring the Android Manifest file for your application in the next chapter.

One application component is the service. An Android Service is basically an Activity without a user interface. It can run as a background process or act much like a web service does, processing requests from third parties. You can use Intents and Activities to launch services using the startService() and bindService() methods. Any Services exposed by an Android application must be registered in the Android Manifest file. You can use services for different purposes. Generally, you use a service when no input is required from the user. Here are some circumstances in which you might want to implement or use an Android service:

- A weather, email, or social network app might implement a service to routinely check for updates. (Note: There are other implementations for polling, but this is a common use of services.)
- A photo or media app that keeps its data in sync online might implement a service to package and upload new content in the background when the device is idle.
- A video-editing app might offload heavy processing to a queue on its service in order to avoid affecting overall system performance for non-essential tasks.
- A news application might implement a service to “pre-load” content by downloading news stories in advance of when the user launches the application, to improve performance.

A good rule of thumb is that if the task requires the use of a worker thread and might affect application responsiveness and performance, consider implementing a service to handle the task outside the main application lifecycle.

Receiving and Broadcasting Intents

Intents serve yet another purpose. You can broadcast an Intent object (via a call to broadcastIntent()) to the Android system, and any application interested can receive that broadcast (called a BroadcastReceiver).Your application might do both sending of and listening for Intent objects. These types of Intent objects are generally used to inform the greater system that something interesting has happened and use special Intent Action types.

For example, the Intent action ACTION_BATTERY_LOW broadcasts a warning when the battery is low. If your application is a battery-hogging Service of some kind, you might want to listen for this Broadcast and shut down your Service until the battery power is sufficient. You can register to listen for battery/charge level changes by listening for the broadcast Intent object with the Intent action ACTION_BATTERY_CHANGED. There are also broadcast Intent objects for other interesting system events, such as SD card state changes, applications being installed or removed, and the wallpaper being changed.

Your application can also share information using the broadcast mechanism. For example, an email application might broadcast Intent whenever a new email arrives so that other applications (such as spam or anti-virus apps) that might be interested in this type of event can react to it.

Defining your Application using the Android Manifest file

RKUniversity™

By : Ketan Bhimani

Defining Your Application Using the Android Manifest File

Android projects use a special configuration file called the Android manifest file to determine application settings—settings such as the application name and version, as well as what permissions the application requires to run and what application components it is comprised of. In this chapter, you explore the Android manifest file in detail and learn how different applications use it to define and describe application behavior.

Configuring the Android Manifest File

The Android application manifest file is a specially formatted XML file that must accompany each Android application. This file contains important information about the application's identity. Here you define the application's name and version information and what applica- tion components the application relies upon, what permissions the application requir- es to run, and other application configuration information.

The Android manifest file is named AndroidManifest.xml and must be included at the top level of any Android project. The information in this file is used by the Android system to

- Install and upgrade the application package.
- Display the application details such as the application name, description, and icon to users.
- Specify application system requirements, including which Android SDKs are support- ed, what hardware configurations are required (for example, d-pad navigation), and which platform features the application relies upon (for example, uses multitouch capabilities).
- Launch application activities.
- Manage application permissions.

- Configure other advanced application configuration details, including acting as a service, broadcast receiver, or content provider.
- Enable application settings such as debugging and configuring instrumentation for application testing.

Editing the Android Manifest File

The manifest resides at the top level of your Android project. You can edit the Android manifest file using the Eclipse Manifest File resource editor (a feature of the Android ADT plug-in for Eclipse) or by manually editing the XML.

Editing the Manifest File Using Eclipse

You can use the Eclipse Manifest File resource editor to edit the project manifest file. The Eclipse Manifest File resource editor organizes the manifest information into categories:

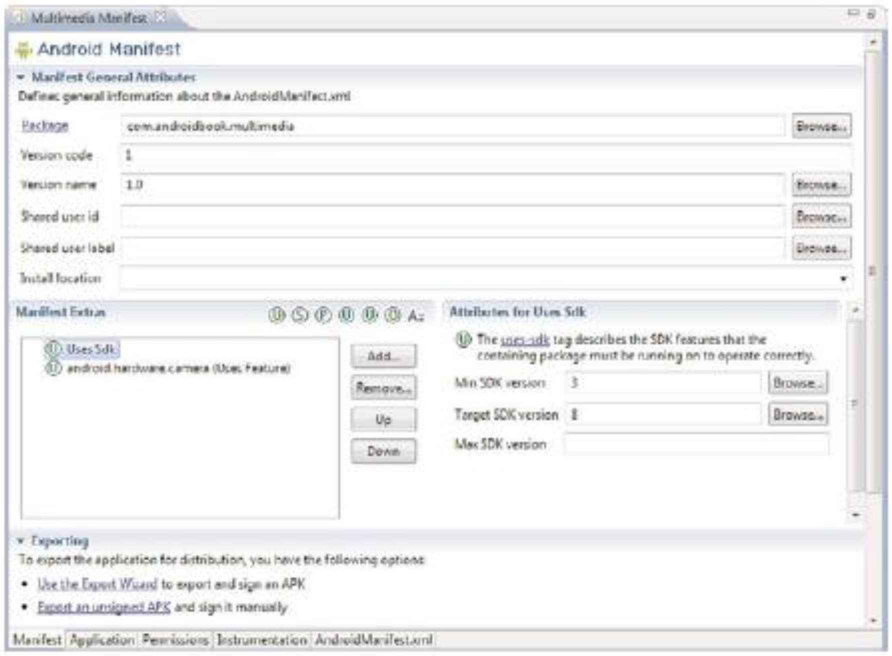
- The Manifest tab
- The Application tab
- The Permissions tab
- The Instrumentation tab
- The AndroidManifest.xml tab

Let's take a closer look at a sample Android manifest file. The figures and samples come from the Android application called Multimedia, which you build in Chapter "Using Android Multimedia APIs."We chose this project because it illustrates a number of different characteristics of the Android manifest file, as opposed to the very simple default manifest file you configured for the My First Android App project.

Configuring Package-Wide Settings Using the Manifest Tab

The Manifest tab contains package-wide settings, including the package name, version information, and supported Android SDK information. You can also set any hardware or feature requirements here.

The Manifest tab of the Eclipse Manifest File resourceeditor.



Defines general information about the AndroidManifest.xml

Package: com.androidbook.multimedia
Version code: 1
Version name: 1.0
Shared user id:
Shared user label:
Install location:

Uses Sdk
android.hardware.camera (Uses Feature)

The uses-sdk tag describes the SDK features that the containing package must be running on to operate correctly.

Min SDK version: 3
Target SDK version: 8
Max SDK version:

To export the application for distribution, you have the following options:

- Use the Export Wizard to export and sign an APK
- Export an unsigned APK and sign it manually

Managing Application and Activity Settings Using the Application Tab

The Application tab contains application-wide settings, including the application label and icon, as well as information about the application components such as activities, intent filters, and other application components, including configuration for services, intent filters, and content providers.

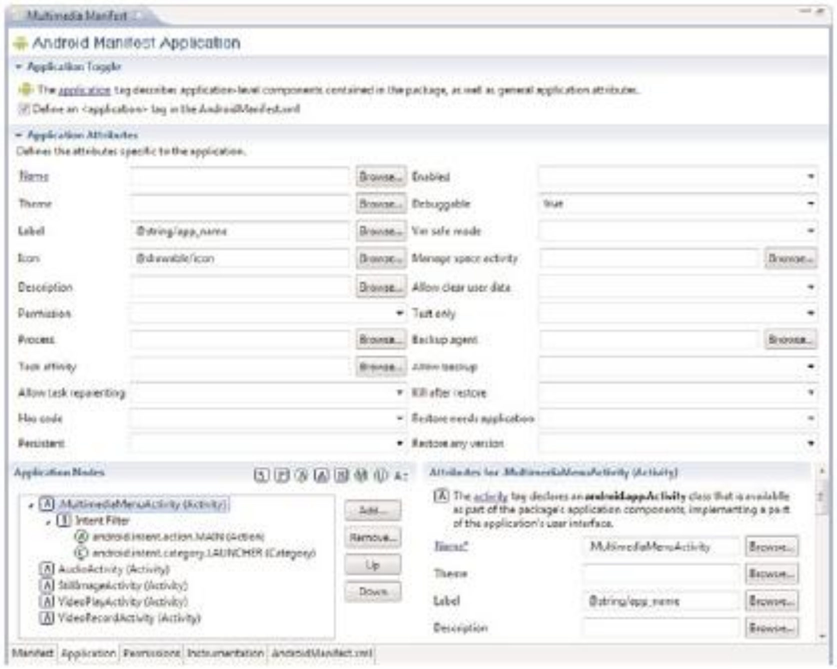
Enforcing Application Permissions Using the Permissions Tab

The Permissions tab contains any permission rules required by your application. This tab can also be used to enforce custom permissions created for the application.

Managing Test Instrumentation Using the Instrumentation Tab

The Instrumentation tab allows the developer to declare any instrumentation classes for monitoring the application. We talk more about instrumentation and testing in Chapter "Testing Android Applications."

The Application tab of the Eclipse Manifest File resource editor.



The application tag describes application-level components contained in the package, as well as general application attributes.

[✓] Define an <application> tag in the AndroidManifest.xml

Defines the attributes specific to the application.

Name		Enabled	
Theme		Debuggable	true
Label	@string/app_name	Vm safe mode	
Icon	@drawable/icon	Manage space activity	
Description		Allow clear user data	
Permission		Test only	
Process		Backup agent	
Task affinity		Allow backup	
Allow task reparenting		Kill after restore	
Has code		Restore needs application	
Persistent		Restore any version	

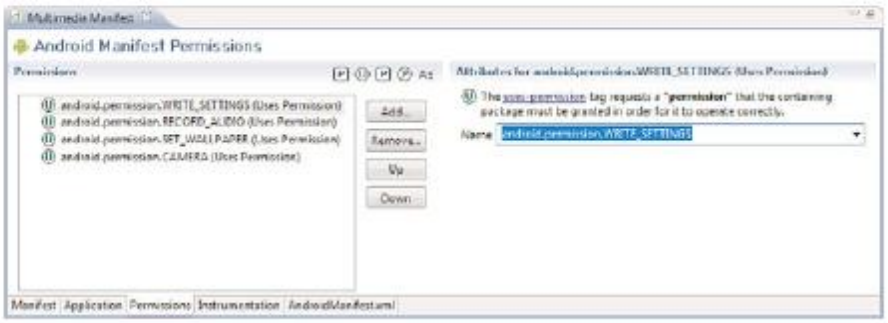
- MultimediaMenuActivity (Activity)
  - Intent Filter
    - android.intent.action.MAIN (Action)
    - android.intent.category.LAUNCHER (Category)
- AudioActivity (Activity)
- StillImageActivity (Activity)
- VideoPlayActivity (Activity)
- VideoRecordActivity (Activity)

Attributes for MultimediaMenuActivity (Activity)

The activity tag declares an android.app.Activity class that is available as part of the package's application components, implementing a part of the application's user interface.

Name*	MultimediaMenuActivity
Theme	
Label	@string/app_name
Description	

The Permissions tab of the Eclipse Manifest File resource editor.



android.permission.WRITE_SETTINGS (Uses Permission)
android.permission.RECORD_AUDIO (Uses Permission)
android.permission.SET_WALLPAPER (Uses Permission)
android.permission.CAMERA (Uses Permission)

Attributes for android.permission.WRITE_SETTINGS (Uses Permission)

The uses-permission tag requests a "permission" that the containing package must be granted in order for it to operate correctly.

Name android.permission.WRITE_SETTINGS

Editing the Manifest File Manually

The Android manifest file is a specially formatted XML file. You can edit the XML manually by clicking on the AndroidManifest.xml tab.

Android manifest files generally include a single <manifest> tag with a single <application> tag. The following is a sample AndroidManifest.xml file for an application called Multimedia:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=http://schemas.android.com/apk/res/android
      package="com.androidbook.multimedia"
      android:versionCode="1"
      android:versionName="1.0">
      <application android:icon="@drawable/icon"
              android:label="@string/app_name"
              android:debuggable="true">
              <activity android:name=".MultimediaMenuActivity"
                      android:label="@string/app_name">
                      <intent-filter>
                          <action
                            android:name="android.intent.action.MAIN"/>
                          <category
                            android:name="android.intent.category.LAUNCHER"/>
                      </intent-filter>
              </activity>
              <activity android:name="AudioActivity"></activity>
              <activity android:name="StillImageActivity"></activity>
              <activity android:name="VideoPlayActivity"></activity>
              <activity android:name="VideoRecordActivity"></activity>
      </application>
      <uses-permission android:name="android.permission.WRITE_SETTINGS"/>
      <uses-permission android:name="android.permission.RECORD_AUDIO"/>
      <uses-permission android:name="android.permission.SET_WALLPAPER"/>
      <uses-permission android:name="android.permission.CAMERA"/>
      <uses-sdk
              android:minSdkVersion="3"
              android:targetSdkVersion="8">
      </uses-sdk>
      <uses-feature android:name="android.hardware.camera" />
</manifest>
```

Here’s a summary of what this file tells us about the Multimedia application:

- The application uses the package name com.androidbook.multimedia.
- The application version name is 1.0.
- The application version code is 1.
- The application name and label are stored in the resource string called @string /app_name within the /res /values /strings.xml resource file.
- The application is debuggable on an Android device.
- The application icon is the graphic file called icon (could be a PNG, JPG, or GIF) stored within the /res/drawable directory (there are actually multiple versions for different pixel densities).
- The application has five activities (MultimediaMenuActivity, AudioActivity, Still Image Activity, Video Play Activity, and Video RecordActivity).
- MultimediaMenuActivity is the primary entry point for the application. This is the activity that starts when the application icon is pressed in the application drawer.
- The application requires the following permissions to run: the ability to record audio, the ability to set the wallpaper on the device, the ability to access the built-in camera, and the ability to write settings.
- The application works from any API level from 3 to 8; in other words, Android SDK 1.5 is the lowest supported, and the application was written to target Android 2.2.
- Finally, the application requires a camera to work properly.

Now let’s talk about some of these important configurations in detail.

Managing Your Application’s Identity

Your application’s Android manifest file defines the application properties. The package name must be defined within the Android manifest file within the <manifest> tag using the package attribute:

```
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
package="com.androidbook.multimedia"
android:versionCode="1"
android:versionName="1.0">
```

Versioning Your Application

Versioning your application appropriately is vital to maintaining your application in the field. Intelligent versioning can help reduce confusion and make product support and upgrades simpler. There are two different version attributes defined within the <manifest> tag: the version name and the version code.

The version name (android:versionName) is a user-friendly, developer-defined version attribute. This information is displayed to users when they manage applications on their devices and when they download the application from marketplaces. Developers use this version information to keep track of their application versions in the field.We discuss appropriate application versioning for mobile applications in detail in Chapter “The Mobile Software Development Process.”

The Android operating system uses the version code (android:versionCode) that is a numeric attribute to manage application upgrades.We talk more about publishing and upgrade support in Chapter “Selling Your Android Application.”

Setting the Application Name and Icon

Overall application settings are configured with the <application> tag of the Android manifest file. Here you set information such as the application icon (android:icon) and friendly name (android:label).These settings are attributes of the <application> tag.

For example, here we set the application icon to a drawable resource provided with the application package and the application label to a string resource:

```
<application android:icon="@drawable/icon"
 android:label="@string/app_name">
```

You can also set optional application settings as attributes in the <application> tag, such as the application description (android:description) and the setting to enable the application for debugging on the device (android:debuggable="true").

Enforcing Application System Requirements

In addition to configuring your application's identity, the Android manifest file is also used to specify any system requirements necessary for the application to run properly. For example, an augmented reality application might require that the handset have GPS, a compass, and a camera. Similarly, an application that relies upon the Bluetooth APIs available within the Android SDK requires a handset with an SDK version of API Level 5 or higher (Android 2.0).These types of system requirements can be defined and enforced in the Android manifest file.Then, when an application is listed on the Android Market, applications can be filtered by these types of information; the Android platform also checks these requirements when installing the application package on the system and errors out if necessary.

Some of the application system requirements that developers can configure through the Android manifest file include

- The Android SDK versions supported by the application
- The Android platform features used by the application
- The Android hardware configurations required by the application
- The screen sizes and pixel densities supported by the application
- Any external libraries that the application links to

Targeting Specific SDK Versions

Android devices run different versions of the Android platform. Often, you see old, less powerful, or even less expensive devices running older versions of the Android platform, whereas newer, more powerful devices that show up on the market often run the latest Android software.

There are now dozens of different Android devices in users' hands. Developers must decide who their target audience is for a given application. Are they trying to support the largest population of users and therefore want to support as many different versions of the platform as possible? Or are they developing a bleeding-edge game that requires the latest device hardware?

Developers can specify which versions of the Android platform an application supports within its Android manifest file using the <uses-sdk> tag. This tag has three important attributes:

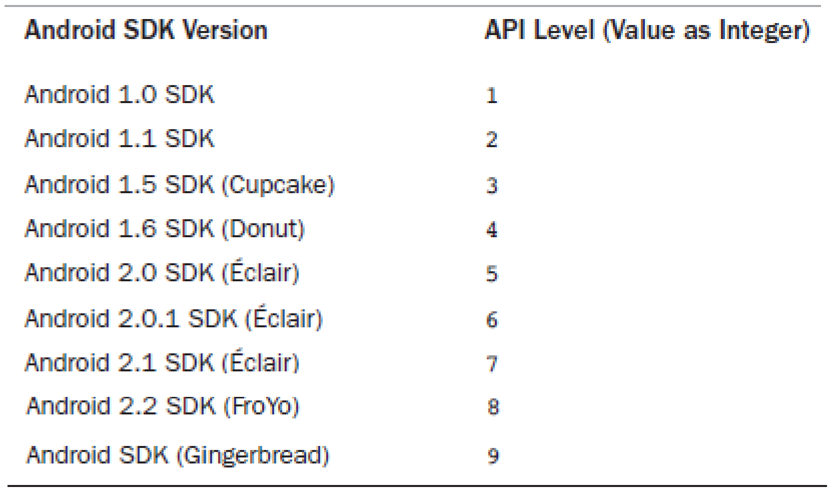
- **The minSdkVersion attribute:**This attribute specifies the lowest API level that the application supports.

- **The targetSdkVersion attribute:**This attribute specifies the optimum API level that the application supports.
- **The maxSdkVersion attribute:**This attribute specifies the highest API level that the application supports.

Each attribute of the <uses-sdk> tag is an integer that represents the API level associated with a given Android SDK. This value does not directly correspond to the SDK version. Instead, it is the revision of the API level associated with that SDK. The API level is set by the developers of the Android SDK. You need to check the SDK documentation to determine the API level value for each version.

This shows the Android SDK versions available for shipping applications.

Android SDK Versions and Their API Levels



Android SDK Version	API Level (Value as Integer)
Android 1.0 SDK	1
Android 1.1 SDK	2
Android 1.5 SDK (Cupcake)	3
Android 1.6 SDK (Donut)	4
Android 2.0 SDK (Éclair)	5
Android 2.0.1 SDK (Éclair)	6
Android 2.1 SDK (Éclair)	7
Android 2.2 SDK (FroYo)	8
Android SDK (Gingerbread)	9

Specifying the Minimum SDK Version

You should always specify the minSdkVersion attribute for your application. This value represents the lowest Android SDK version your application supports.

For example, if your application requires APIs introduced in Android SDK 1.6, you would check that SDK's documentation and find that this release is defined as API Level 4. Therefore, add the following to your Android Manifest file within the

```
<manifest> tag block:
<uses-sdk android:minSdkVersion="4" />
```

It's that simple. You should use the lowest API level possible if you want your application to be compatible with the largest number of Android handsets. However, you must ensure that your application is tested sufficiently on any non-target platforms (any API level supported below your target SDK, as described in the next section).

Specifying the Target SDK Version

You should always specify the targetSdkVersion attribute for your application. This value represents the Android SDK version your application was built for and tested against.

For example, if your application was built using the APIs that are backward-compatible to Android 1.6 (API Level 4), but targeted and tested using Android 2.2 SDK (API Level 8), then you would want to specify the targetSdkVersion attribute as 8.Therefore, add the following to your Android manifest file within the <manifest> tag block:

```
<uses-sdk android:minSdkVersion="4" android:targetSdkVersion="8" />
```

Why should you specify the target SDK version you used? Well, the Android platform has built-in functionality for backward-compatibility (to a point).Think of it like this: A specific method of a given API might have been around since API Level 1. However, the internals of that method—its behavior—might have changed slightly from SDK to SDK. By specifying the target SDK version for your application, the Android operating system attempts to match your application with the exact version of the SDK (and the behavior as you tested it within the application), even when running a different (newer) version of the platform. This means that the application should continue to behave in "the old way" despite any new changes or "improvements" to the SDK that might cause unintended consequences in your application.

Specifying the Maximum SDK Version

You will rarely want to specify the maxSdkVersion attribute for your application. This value represents the highest Android SDK version your application supports, in terms of API level. It restricts forward-compatibility of your application.

One reason you might want to set this attribute is if you want to limit who can install the application to exclude devices with the newest SDKs. For example, you might develop a free beta version of your application with plans for a paid version for the newest SDK. By setting the maxSdkVersion attribute of the manifest file for your free application, you disallow anyone with the newest SDK to install the free version of the application. The downside of this idea? If your users have phones that receive over-the-air SDK updates, your application would cease to work (and appear) on phones where it had functioned perfectly, which might “upset” your users and result in bad ratings on your market of choice.

The short answer: Use maxSdkVersion only when absolutely necessary and when you understand the risks associated with its use.

Enforcing Application Platform Requirements

Android devices have different hardware and software configurations. Some devices have built-in keyboards and others rely upon the software keyboard. Similarly, certain Android devices support the latest 3-D graphics libraries and others provide little or no graphics support. The Android manifest file has several informational tags for flagging the system features and hardware configurations supported or required by an Android application.

Specifying Supported Input Methods

The <uses-configuration> tag can be used to specify which input methods the application supports. There are different configuration attributes for five-way navigation, the hardware keyboard and keyboard types; navigation devices such as the directional pad, trackball, and wheel; and touch screen settings.

There is no “OR” support within a given attribute. If an application supports multiple input configurations, there must be multiple <uses-configuration> tags—one for each complete configuration supported.

For example, if your application requires a physical keyboard and touch screen input using a finger or a stylus, you need to define two separate <uses- configuration> tags in your manifest file, as follows:

```
<uses-configuration android:reqHardKeyboard="true"
     android:reqTouchScreen="finger" />
<uses-configuration android:reqHardKeyboard="true"
     android:reqTouchScreen="stylus" />
```

Specifying Required Device Features

Not all Android devices support every Android feature. Put another way: There are a number of APIs (and related hardware) that Android devices may optionally include. For example, not all Android devices have multi-touch ability or a camera flash.

The <uses-feature> tag can be used to specify which Android features the application requires to run properly. These settings are for informational purposes only—the Android operating system does not enforce these settings, but publication channels such as the Android Market use this information to filter the applications available to a given user.

If your application requires multiple features, you must create a <uses-feature> tag for each. For example, an application that requires both a light and proximity sensor requires two tags:

```
<uses-feature android:name="android.hardware.sensor.light" />
<uses-feature android:name="android.hardware.sensor.proximity" />
```

One common reason to use the <uses-feature> tag is for specifying the OpenGL ES versions supported by your application. By default, all applications function with OpenGL ES 1.0 (which is a required feature of all Android devices). However, if your application requires features available only in later versions of OpenGL ES, such as 2.0, then you must specify this feature in the Android manifest file. This is done using the android:glEsVersion attribute of the <uses-feature> tag. Specify the lowest version of OpenGL ES that the application requires. If the application works with 1.0 and 2.0, specify the lowest version (so that the Android Market allows more users to install your application).

For more information about the <uses-feature> tag of the Android manifest file, see the Android SDK reference.

Specifying Supported Screen Sizes

Android devices come in many shapes and sizes. Screen sizes and pixel densities vary widely across the range of Android devices available on the market today. The Android platform categorizes screen types in terms of sizes (small, normal, and large) and pixel density (low, medium, and high).These characteristics effectively cover the variety of screen types available within the Android platform.

An application can provide custom resources for specific screen sizes and pixel densities (we cover this in Chapter "Managing Application Resources").The <supportsscreen> tag can be used to specify which Android types of screens the application supports.

For example, if the application supports QVGA screens (small) and HVGA screens (normal) regardless of pixel density, the <supports-screen> tag is configured as follows:

```
<supports-screens android:smallScreens="true"
       android:normalScreens="true"
       android:largeScreens"false"
       android:anyDensity="true"/>
```

For more information about the <supports-screen> tag of the Android manifest file, see the Android SDK reference as well as the Android Dev Guide documentation on Screen Support.

Working with External Libraries

You can register any shared libraries your application links to within the Android manifest file. By default, every application is linked to the standard Android packages (such as android.app) and is aware of its own package.However, if your application links to additional packages, they must be registered within the <application> tag of the Android manifest file using the <uses-library> tag. For example

```
<uses-library android:name="com.sharedlibrary.sharedStuff" />
```

This feature is often used for linking to optional Google APIs. For more information about the <uses-library> tag of the Android manifest file, see the Android SDK reference.

Registering Activities and Other Application Components

Each Activity within the application must be defined within the Android manifest file with an <activity> tag. For example, the following XML excerpt defines an Activity class called AudioActivity:

```
<activity android:name="AudioActivity" />
```

This Activity must be defined as a class within the com.androidbook.multimedia package. That is, the package specified in the <manifest> element of the Android manifest file.

You can also enforce scope of the activity class by using the dot as a prefix in the Activity name:

```
<activity android:name=".AudioActivity" />
```

Or you can specify the complete class name:

```
<activity android:name="com.androidbook.multimedia.AudioActivity" />
```

Designating a Primary Entry Point Activity for Your Application Using an Intent Filter

An Activity class can be designated as the primary entry point by configuring an intent filter using the Android manifest tag <intent-filter> in the application's AndroidManifest.xml file with the MAIN action type and the LAUNCHER category.

The following tag of XML configures the Activity class called MultimediaMenuActivity as the primary launching point of the application:

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

Configuring Other Intent Filters

The Android operating system uses Intent filters to resolve implicit intents. That is, Intents that do not specify the Activity or Component they want launched. Intent filters can be applied to Activities, Services, and BroadcastReceivers. The filter declares that this component is open to receiving any Intent sent to the Android operating system that matches its criteria.

Intent filters are defined using the <intent-filter> tag and must contain at least one <action> tag but can also contain other information, such as <category> and <data> blocks. Here we have a sample intent filter block, which might be found within an <activity> block:

```
<intent-filter>
<action android:name="android.intent.action.VIEW" />
<category android:name="android.intent.category.BROWSABLE" />
<category android:name="android.intent.category.DEFAULT" />
<data android:scheme="geoname"/>
</intent-filter>
```

This intent filter definition uses a predefined action called VIEW, the action for viewing particular content. It is also BROWSABLE and uses a scheme of geoname so that when a Uri starts with geoname://, the activity with this intent filter launches.You can read more about this particular intent filter in Chapter "Using Location-Based Services (LBS) APIs."

Registering Services and Broadcast Receivers

All application components are defined within the Android manifest file. In addition to activities, all services and broadcast receivers must be registered within the Android Manifest file.

- Services are registered using the <service> tag.
- Broadcast Receivers are registered using the <receiver> tag.

Both Services and Broadcast Receivers use intent filters. You learn much more about services, broadcast receivers, and intent filters later in the book.

Registering Content Providers

If your application acts as a content provider, effectively exposing a shared data service for use by other applications, it must declare this capability within the Android manifest file using the <provider> tag. Configuring a content provider involves determining what subsets of data are shared and what permissions are required to access them, if any.

Working with Permissions

The Android operating system has been locked down so that applications have limited capability to adversely affect operations outside their process space. Instead, Android applications run within the bubble of their own virtual machine, with their own Linux user account (and related permissions).

Registering Permissions Your Application Requires

Android applications have no permissions by default. Instead, any permissions for shared resources or privileged access—whether it's shared data, such as the Contacts database, or access to underlying hardware, such as the built-in camera—must be explicitly registered within the Android manifest file. These permissions are granted when the application is installed.

The following XML excerpt for the preceding Android manifest file defines a ermission using the <uses-permission> tag to gain access to the built-in camera:

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

A complete list of the permissions can be found in the android.Manifest.permission class. Your application manifest should include only the permissions required to run. The user is informed what permissions each Android application requires at install time.

You might find that, in certain cases, permissions are not enforced (you can operate without the permission). In these cases, it is prudent to request the permission anyway for two reasons. First, the user is informed that the application is performing those sensitive actions, and second, that permission could be enforced in a later SDK version.

Registering Permissions Your Application Grants to Other Applications

Applications can also define their own permissions by using the <permission> tag. Permissions must be described and then applied to specific application components, such as Activities, using the android:permission attribute.

Permissions can be enforced at several points:

- When starting an Activity or Service
- When accessing data provided by a content provider
- At the function call level
- When sending or receiving broadcasts by an Intent

Permissions can have three primary protection levels: normal, dangerous, and signature. The normal protection level is a good default for fine-grained permission enforcement within the application. The dangerous protection level is used for higherrisk Activities, which might adversely affect the device. Finally, the signature protection level permits any application signed with the same certificate to use that component for controlled application inter operability.

Permissions can be broken down into categories, called permission groups, which describe or warn why specific Activities require permission. For example, permissions might be applied for Activities that expose sensitive user data such as location and personal information (android. permission- group.LOCATION and android. permission group. PERSONAL_ INFO), access underlying hardware (android.permissiongroup. HARDWARE_ CONTROLS), or perform operations that might incur fees to the user (android .permission -group.COST_MONEY). A complete list of permission groups is available within the Manifest .permission _group class.

Enforcing Content Provider Permissions at the Uri Level

You can also enforce fine-grained permissions at the Uri level using the <grant-uri- permissions> tag.

Exploring Other Manifest File Settings

We have now covered the basics of the Android manifest file, but there are many other settings configurable within the Android manifest file using different tag blocks, not to mention attributes within each tag we already discussed.

Some other features you can configure within the Android manifest file include

- Setting application-wide themes as <application> tag attributes
- Configuring instrumentation using the <instrumentation> tag
- Aliasing activities using the <activity-alias> tag
- Creating intent filters using the <intent-filter> tag
- Creating broadcast receivers using the <receiver> tag

Managing Application Resources

RKUNIVERSITY™

By : Ketan Bhimani

Managing Application Resources

The well-written application accesses its resources programmatically instead of hard coding them into the source code. This is done for a variety of reasons. Storing application resources in a single place is a more organized approach to development and makes the code more readable and maintainable. Externalizing resources such as strings makes it easier to localize applications for different languages and geographic regions.

In this chapter, you learn how Android applications store and access important resources such as strings, graphics, and other data. You also learn how to organize Android resources within the project files for localization and different device configurations.

What Are Resources?

All Android applications are composed of two things: functionality (code instructions) and data (resources).The functionality is the code that determines how your application behaves. This includes any algorithms that make the application run. Resources include text strings, images and icons, audio files, videos, and other data used by the application.

Storing Application Resources

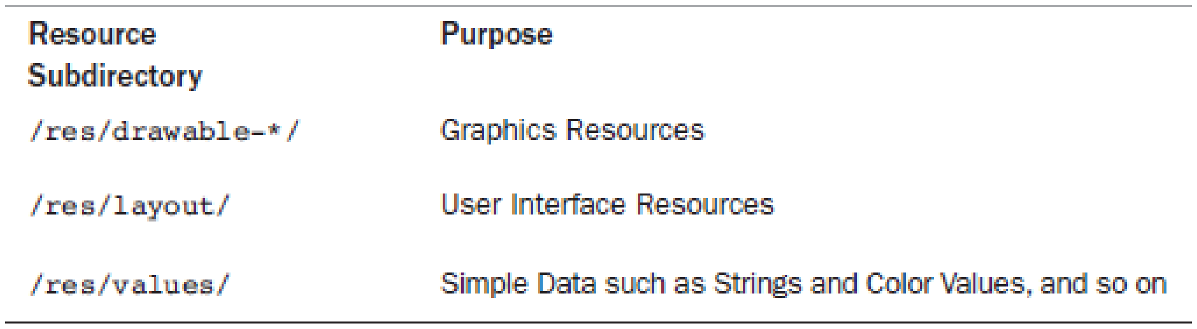
Android resource files are stored separately from the java class files in the Android project. Most common resource types are stored in XML. You can also store raw data files and graphics as resources.

Understanding the Resource Directory Hierarchy

Resources are organized in a strict directory hierarchy within the Android project. All resources must be stored under the /res project directory in specially named subdirectories that must be lowercase.

Different resource types are stored in different directories. The resource sub-directories generated when you create an Android project using the Eclipse plug-in are shown in Table.

Default Android Resource Directories



Resource Subdirectory	Purpose
`/res/drawable-*/`	Graphics Resources
`/res/layout/`	User Interface Resources
`/res/values/`	Simple Data such as Strings and Color Values, and so on

Each resource type corresponds to a specific resource subdirectory name. For example, all graphics are stored under the /res/drawable directory structure. Resources can be further organized in a variety of ways using even more specially named directory qualifiers. For example, the /res/drawable-hdpi directory stores graphics for high-density screens, the /res/drawable-ldpi directory stores graphics for low-density screens, and the /res/drawable-mdpi directory stores graphics for medium-density screens. If you had a graphic resource that was shared by all screens, you would simply store that resource in the /res/drawable directory. We talk more about resource directory qualifiers later in this chapter.

Using the Android Asset Packaging Tool

If you use the Eclipse with the Android Development Tools Plug-In, you will find that adding resources to your project is simple. The plug-in detects new resources when you add them to the appropriate project resource directory under /res automatically. These resources are compiled, resulting in the generation of the R.java file, which enables you to access your resources programmatically.

If you use a different development environment, you need to use the aapt tool command-line interface to compile your resources and package your application binaries to deploy to the phone or emulator. You can find the aapt tool in the /tools subdirectory of each specific Android SDK version.

Resource Value Types

Android applications rely on many different types of resources—such as text strings, graphics, and color schemes—for user interface design.

These resources are stored in the /res directory of your Android project in a strict (but reasonably flexible) set of directories and files. All resources filenames must be lowercase and simple (letters, numbers, and underscores only).

The resource types supported by the Android SDK and how they are stored within the project are shown in Table.

How Important Resource Types Are Stored in Android Project Resource Directories

Resource Type	Required Directory	Filename	XML Tag
Strings	`/res/values/`	`strings.xml` (suggested)	`<string>`
String Pluralization	`/res/values/`	`strings.xml` (suggested)	`<plurals>`, `<item>`
Arrays of Strings	`/res/values/`	`strings.xml` (suggested)	`<string-array>`, `<item>`
Booleans	`/res/values/`	`bools.xml` (suggested)	`<bool>`
Colors	`/res/values/`	`Colors.xml` (suggested)	`<color>`
Color State Lists	`/res/color/`	Examples include `buttonstates.xml` `indicators.xml`	`<selector>`, `<item>`
Dimensions	`/res/values/`	`Dimens.xml` (suggested)	`<dimen>`
Integers	`/res/values/`	`integers.xml` (suggested)	`<integer>`
Arrays of Integers	`/res/values/`	`integers.xml` (suggested)	`<integer-array>`, `<item>`

Resource Type	Required Directory	Filename	XML Tag
Mixed-Type Arrays	`/res/values/`	`Arrays.xml` (suggested)	`<array>`, `<item>`
Simple Drawables (Paintable)	`/res/values/`	`drawables.xml` (suggested)	`<drawable>`
Graphics	`/res/drawable/`	Examples include `icon.png logo.jpg`	Supported graphics files or drawable definition XML files such as shapes.
Tweened Animations	`/res/anim/`	Examples include `fadesequence.xml` `spinsequence.xml`	`<set>`, `<alpha>`, `<scale>`, `<translate>`, `<rotate>`
Frame-by-Frame Animations	`/res/drawable/`	Examples include `sequence1.xml` `sequence2.xml`	`<animation-list>`, `<item>`
Menus	`/res/menu/`	Examples include `mainmenu.xml` `helpmenu.xml`	`<menu>`
XML Files	`/res/xml/`	Examples include `data.xml` `data2.xml`	Defined by the developer.
Raw Files	`/res/raw/`	Examples include `jingle.mp3` `somevideo.mp4` `helptext.txt`	Defined by the developer.
Layouts	`/res/layout/`	Examples include `main.xml` `help.xml`	Varies. Must be a layout control.
Styles and Themes	`/res/values/`	`styles.xml` `themes.xml` (suggested)	`<style>`

Storing Different Resource Value Types

The aapt traverses all properly formatted files in the /res directory hierarchy and generates the class file R.java in your source code directory /src to access all variables.

Later in this chapter, we cover how to store and use each different resource type in detail, but for now, you need to understand that different types of resources are stored in different ways.

Storing Simple Resource Types Such as Strings

Simple resource value types, such as strings, colors, dimensions, and other primitives, are stored under the /res/values project directory in XML files. Each resource file under the /res/values directory should begin with the following XML header:

```
<?xml version=”1.0” encoding=”utf-8”?>
```

Next comes the root node <resources> followed by the specific resource element types such as <string> or <color>. Each resource is defined using a different element name.

Although the XML file names are arbitrary, the best practice is to store your resources in separate files to reflect their types, such as strings.xml, colors.xml, and so on. However, there’s nothing stopping the developers from creating multiple resource files for a given type, such as two separate xml files called bright_ colors .xml and muted_colors.xml, if they so choose.

Storing Graphics, Animations, Menus, and Files

In addition to simple resource types stored in the /res/values directory, you can also store numerous other types of resources, such as animation sequences, graphics, arbitrary XML files, and raw files. These types of resources are not stored in the /res/values directory, but instead stored in specially named directories according to their type. For example, you can include animation sequence definitions in the /res/anim directory. Make sure you name resource files appropriately because the resource name is derived from the filename of the specific resource. For example, a file called flag.png in the /res/drawable directory is given the name R.drawable.flag.

Understanding How Resources Are Resolved

Few applications work perfectly, no matter the environment they run in. Most require some tweaking, some special case handling. That's where alternative resources come in. You can organize Android project resources based upon more than a dozen different types of criteria, including language and region, screen characteristics, device modes (night mode, docked, and so on), input methods, and many other device differentiators.

It can be useful to think of the resources stored at the top of the resource hierarchy as *default resources* and the specialized versions of those resources as *alternative resources*. Two common reasons that developers use alternative resources are for internationalization and localization purposes and to design an application that runs smoothly on different device screens and orientations.

The Android platform has a very robust mechanism for loading the appropriate resources at runtime. An example might be helpful here. Let's presume that we have a simple application with its requisite string, graphic, and layout resources. In this application, the resources are stored in the top-level resource directories (for example,*/res/values/strings.xml*, */res/drawable/myLogo.png*, and */res/layout/main.xml*). No matter what Android device (huge hi-def screen, postage-stamp-sized screen, English or Chinese language or region, portrait or landscape orientation, and so on), you run this application on, the same resource data is loaded and used.

Back in our simple application example, we could create alternative string resources in Chinese simply by adding a second *strings.xml* file in a resource subdirectory called*/res/values-zh/strings.xml* (note the *–zh* qualifier).We could provide different logos for different screen densities by providing three versions of *myLogo.png*:

- /res/drawable-ldpi/myLogo.png (low-density screens)
- /res/drawable-mdpi/myLogo.png (medium-density screens)
- /res/drawable-hdpi/myLogo.png (high-density screens)

Finally, let's say that the application would look much better if the layout was different in portrait versus landscape modes. We could change the layout around, moving controls around, in order to achieve a more pleasant user experience, and provide two layouts:

- /res/layout-port/main.xml (layout loaded in portrait mode)
- /res/layout-land/main.xml (layout loaded in landscape mode)

With these alternative resources in place, the Android platform behaves as follows:

- If the device language setting is Chinese, the strings in /res/values-zh/strings.xml are used. In all other cases, the strings in /res/values/strings.xml are used.
- If the device screen is a low-density screen, the graphic stored in the /res/drawable-ldpi/myLogo.png resource directory is used. If it’s a medium-density screen, the mdpi drawable is used, and so on.
- If the device is in landscape mode, the layout in the /res/layout-land/main.xml is loaded. If it’s in portrait mode, the /res/layout-port/main.xml layout is loaded.

There are four important rules to remember when creating alternative resources:

1. The Android platform always loads the most specific, most appropriate resource available. If an alternative resource does not exist, the default resource is used. Therefore, know your target devices, design for the defaults, and add alternative resources judiciously.
2. Alternative resources must always be named exactly the same as the default resources. If a string is called strHelpText in the /res/values/strings.xml file, then it must be named the same in the /res/values-fr/strings.xml (French) and /res/values-zh/strings.xml (Chinese) string files. The same goes for all other types of resources, such as graphics or layout files.
3. Good application design dictates that alternative resources should always have a default counterpart so that regardless of the device, some version of the resource always loads.The only time you can get away without a default resource is when you provide every kind of alternative resource (for example, providing ldpi, mdpi, and hdpi graphics resources cover every eventuality, in theory).
4. Don’t go overboard creating alternative resources, as they add to the size of your application package and can have performance implications. Instead, try to design your default resources to be flexible and scalable. For example, a good layout design can often support both landscape and portrait modes seamlessly—if you use the right controls.

Enough about alternative resources; let's spend the rest of this chapter talking about how to create the default resources first. In Chapter "Targeting Different Device Configurations and Languages," we discuss how to use alternative resources to make your Android applications compatible with many different device configurations.

Accessing Resources Programmatically

Developers access specific application resources using the R.java class file and its subclasses, which are automatically generated when you add resources to your project (if you use Eclipse).You can refer to any resource identifier in your project by name. For example, the following string resource named strHello defined within the resource file called /res /values /strings .xml is accessed in the code as R.string.strHello

This variable is not the actual data associated with the string named hello. Instead, you use this resource identifier to retrieve the resource of that type (which happens to be string).

For example, a simple way to retrieve the string text is to call

String myString = getResources().getString(R.string.strHello);

First, you retrieve the Resources instance for your application Context (android .content. Context), which is, in this case, this because the Activity class extends Context.Then you use the Resources instance to get the appropriate kind of resource you want. You find that the Resources class (android.content.res.Resources) has helper methods for handling every kind of resource.

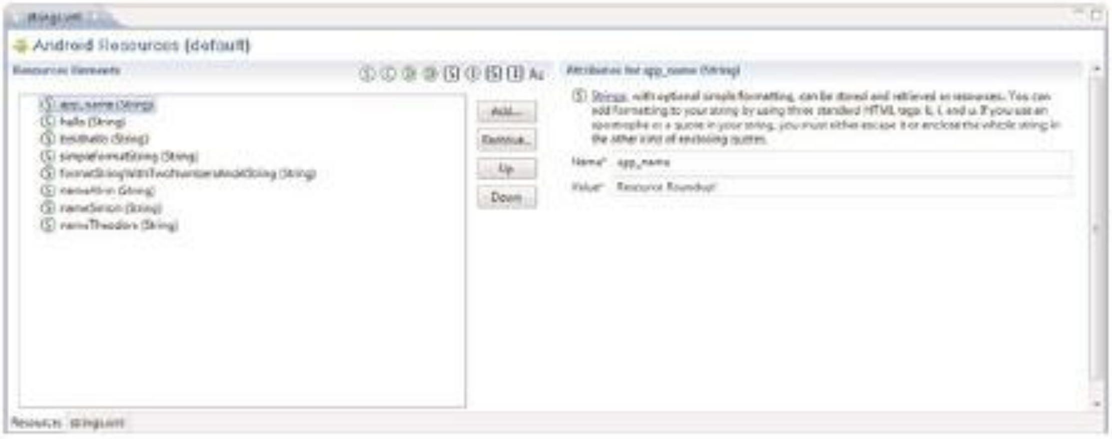
Before we go any further, we find it can be helpful to dig in and create some resources, so let’s create a simple example. Don’t worry if you don’t understand every aspect of the exercise. You can find out more about each different resource type later in this chapter.

Setting Simple Resource Values Using Eclipse

Developers can define resource types by editing resource XML files manually and using the aapt to compile them and generate the R.java file or by using Eclipse with the Android plug-in, which includes some very handy resource editors.

To illustrate how to set resources using the Eclipse plug-in, let’s look at an example. Create a new Android project and navigate to the /res/values/strings.xml file in Eclipse and double-click the file to edit it. Your strings.xml resource file opens in the right pane and should look something like Figure.

The string resource file in the Eclipse Resource Editor (Editor view).



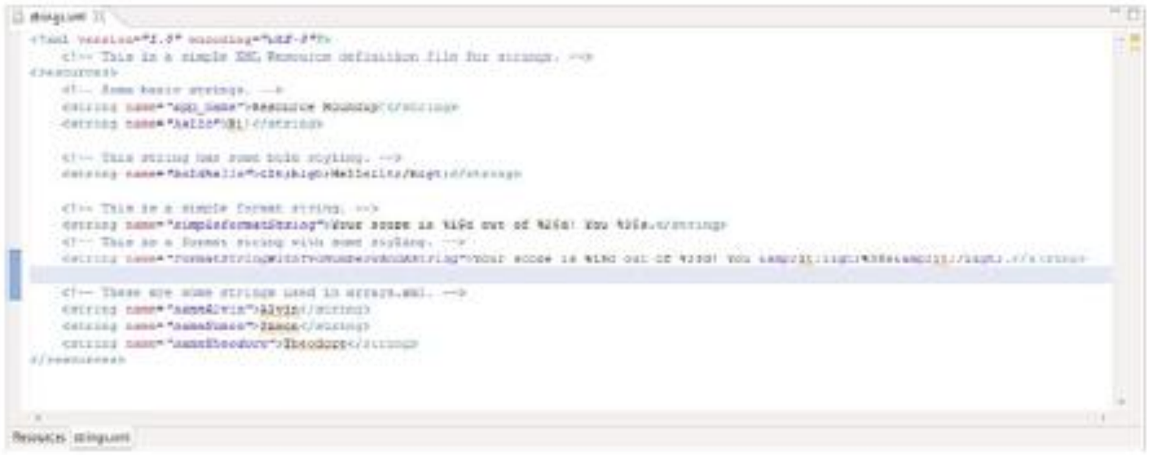
Strings, with optional simple formatting, can be stored and retrieved as resources. You can add formatting to your string by using three standard HTML tags: b, i, and u. If you use an apostrophe or a quote in your string, you must either escape it or enclose the whole string in the other kind of enclosing quotes.

Name* app_name

Value* Resource Roundup!

There are two tabs at the bottom of this pane.The Resources tab provides a friendly method to easily insert primitive resource types such as strings, colors, and dimension resources. The strings.xml tab shows the raw XML resource file you are creating. Sometimes, editing the XML file manually is much faster, especially if you add a number of new resources. Click the strings.xml tab, and your pane should look something like Figure.

The string resource files in the Eclipse Resource Editor (XML view).



```
<?xml version="1.0" encoding="utf-8"?>
    <!-- This is a simple XML Resource definition file for strings. -->
<resources>
    <!-- Some basic strings. -->
    <string name="app_name">Resource Roundup</string>
    <string name="hello">Hi!</string>

    <!-- This string has some bold styling. -->
    <string name="boldhello">[illegible]</string>

    <!-- This is a simple format string. -->
    <string name="simpleformatString">Your score is %1$d out of %2$d! You %3$s.</string>
    <!-- This is a format string with some styling. -->
    <string name="formatStringWith[illegible]">Your score is %1$d out of %2$d! You [illegible]</string>

    <!-- These are some strings used in arrays.xml. -->
    <string name="nameAlvin">Alvin</string>
    <string name="nameSimon">Simon</string>
    <string name="nameTheodore">Theodore</string>
</resources>
```

Now add some resources using the Add button on the Resources tab. Specifically, create the following resources:

- A color resource named prettyTextColor with a value of #ff0000
- A dimension resource named textPointSize with a value of 14pt
- A drawable resource named redDrawable with a value of #F00

Now you have several resources of various types in your strings.xml resource file. If you switch back to the XML view, you see that the Eclipse resource editor has added the appropriate XML elements to your file, which now should look something like this:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
     <string name=”app_name”>ResourceRoundup</string>
     <string
        name=”hello”>Hello  World,  ResourceRoundupActivity</string>
     <color name=”prettyTextColor”>#ff0000</color>
     <dimen name=”textPointSize”>14pt</dimen>
     <drawable name=”redDrawable”>#F00</drawable>
</resources>
```

Save the strings.xml resource file. The Eclipse plug-in automatically generates the R.java file in your project, with the appropriate resource IDs, which enable you to programmatically access your resources after they are compiled into the project. If you navigate to your R.java file, which is located under the /src directory in your package, it looks something like this:

```
package com.androidbook.resourceroundup;
      public final class R {
              public static final class attr {
       }
      public static final class color {
              public static final int prettyTextColor=0x7f050000;
      }
      public static final class dimen {
              public static final int textPointSize=0x7f060000;
      }
      public static final class drawable {
              public static final int icon=0x7f020000;
              public static final int redDrawable=0x7f020001;
      }
      public static final class layout {
              public static final int main=0x7f030000;
      }
      public static final class string {
              public static final int app_name=0x7f040000;
              public static final int hello=0x7f040001;
      }
}
```

Now you are free to use these resources in your code. If you navigate to your Resource Round upActivity.java source file, you can add some lines to retrieve your resources and work with them, like this:

```
import android.graphics.drawable.ColorDrawable;
...
String myString = getResources().getString(R.string.hello);
int myColor =
     getResources().getColor(R.color.prettyTextColor);
float myDimen =
     getResources().getDimension(R.dimen.textPointSize);
ColorDrawable myDraw = (ColorDrawable)getResources().
     getDrawable(R.drawable.redDrawable);
```

Some resource types, such as string arrays, are more easily added to resource files by editing the XML by hand. For example, if we go back to the strings.xml file and choose the strings.xml tab, we can add a string array to our resource listing by adding the following XML element:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
     <string name=”app_name”>Use Some Resources</string>
     <string
            name=”hello”>Hello World, UseSomeResources</string>
      <color name=”prettyTextColor”>#ff0000</color>
     <dimen name=”textPointSize”>14pt</dimen>
     <drawable name=”redDrawable”>#F00</drawable>
     <string-array name=”flavors”>
            <item>Vanilla</item>
            <item>Chocolate</item>
            <item>Strawberry</item>
     </string-array>
</resources>
```

Save the strings.xml file, and now this string array named “flavors” is available in your source file R.java, so you can use it programmatically in resourcesroundup.java like this:

```
String[] aFlavors =
getResources().getStringArray(R.array.flavors);
```

You now have a general idea how to add simple resources using the Eclipse plug-in, but there are quite a few different types of data available to add as resources. It is a common practice to store different types of resources in different files. For example, you might store the strings in /res/values/strings.xml but store the prettyTextColor color resource in /res/values/colors.xml and the textPointSize dimension resource in /res/values/dimens.xml. Reorganizing where you keep your resources in the resource directory hierarchy does not change the names of the resources, nor the code used earlier to access the resources programmatically.

Now let's have a look at how to add different types of resources to your project.

Working with Resources

In this section, we look at the specific types of resources available for Android applications, how they are defined in the project files, and how you can access this resource data programma- tically.

For each type of resource type, you learn what types of values can be stored and in what format. Some resource types (such as Strings and Colors) are well supported with the Android Plug-in Resource Editor, whereas others (such as Animation sequences) are more easily managed by editing the XML files directly.

Working with String Resources

String resources are among the simplest resource types available to the developer. String resources might show text labels on form views and for help text. The application name is also stored as a string resource, by default.

String resources are defined in XML under the /res/values project directory and compiled into the application package at build time. All strings with apostrophes or single straight quotes need to be escaped or wrapped in double straight quotes. Some examples of well-formatted string values are shown in Table.

String Resource Formatting Examples

String Resource Value	Displays As
Hello, World	Hello, World
"User's Full Name:"	User's Full Name:
User\'s Full Name:	User's Full Name:
She said, \"Hi.\"	She said, "Hi."
She\'s busy but she did say, \"Hi.\"	She's busy but she did say, "Hi."

You can edit the strings.xml file using the Resources tab, or you can edit the XML directly by clicking the file and choosing the strings.xml tab. After you save the file, the resources are automatically added to your R.java class file.

String values are appropriately tagged with the <string> tag and represent a namevalue pair. The name attribute is how you refer to the specific string programmatically, so name these resources wisely.

Here's an example of the string resource file /res/values/strings.xml:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
     <string name=”app_name”>Resource Viewer</string>
     <string name=”test_string”>Testing 1,2,3</string>
     <string name=”test_string2”>Testing 4,5,6</string>
</resources>
```

Bold, Italic,and Underlined Strings

You can also add three HTML-style attributes to string resources. These are bold, italic,and underlining. You specify the styling using the <b>, <i>, and <u> tags. For example

```
<string
name=”txt”><b>Bold</b>,<i>Italic</i>,<u>Line</u></string>
```

Using String Resources as Format Strings

You can create format strings, but you need to escape all bold, italic, and underlining tags if you do so. For example, this text shows a score and the “win” or “lose” string:

```
<string
name=”winLose”>Score: %1$d of %2$d!  You %3$s.</string>
```

If you want to include bold, italic, or underlining in this format string, you need to escape the format tags. For example, if want to italicize the “win” or “lose” string at the end, your resource would look like this:

```
<string name=”winLoseStyled”>
Score: %1$d of %2$d! You&lt;i&gt;%3$s&lt;/i&gt;.</string>
```

Using String Resources Programmatically

As shown earlier in this chapter, accessing string resources in code is straightforward. There are two primary ways in which you can access this string resource.

The following code accesses your application's string resource named hello, returning only the string. All HTML-style attributes (bold, italic, and underlining) are stripped from the string.

```
      String myStrHello =
      getResources().getString(R.string.hello);
```

You can also access the string and preserve the formatting by using this other method:

```
      CharSequence myBoldStr =
      getResources().getText(R.string.boldhello);
```

To load a format string, you need to make sure any format variables are properly escaped.One way you can do this is by using the TextUtils.htmlEncode() method:

```
import android.text.TextUtils;
...
String mySimpleWinString;
     mySimpleWinString  =
getResources().getString(R.string.winLose);
String escapedWin = TextUtils.htmlEncode("Won");
String resultText =
     String.format(mySimpleWinString, 5, 5, escapedWin);
```

The resulting text in the resultText variable is

```
 Score: 5 of 5! You Won.
```

Now if you have styling in this format string like the preceding winLoseStyled string resource, you need to take a few more steps to handle the escaped italic tags.

```
import android.text.Html;
import android.text.TextUtils;
...
String myStyledWinString;
myStyledWinString  =
     getResources().getString(R.string. winLoseStyled);
String escapedWin = TextUtils.htmlEncode(“Won”);
String resultText =
     String.format(myStyledWinString, 5, 5, escapedWin);
CharSequence  styledResults = Html.fromHtml(resultText);
```

Theresulting text in the styledResults variable is

```
Score: 5 of 5! You <i>won</i>.
```

This variable, styledResults, can then be used in user interface controls such as TextView objects, where styled text is displayed correctly.

Working with String Arrays

You can specify lists of strings in resource files. This can be a good way to store menu options and drop-down list values. String arrays are defined in XML under the /res/values project directory and compiled into the application package at build time.String arrays are appropriately tagged with the <string-array> tag and a number of <item> child tags, one for each string in the array. Here’s an example of a simple array resource file /res /values /arrays .xml:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
    <string-array name=”flavors”>
          <item>Vanilla Bean</item>
          <item>Chocolate Fudge Brownie</item>
          <item>Strawberry Cheesecake</item>
          <item>Coffee, Coffee, Buzz Buzz  Buzz</item>
          <item>Americone Dream</item>
   </string-array>
   <string-array name="soups">
          <item>Vegetable minestrone</item>
          <item>New England clam chowder</item>
          <item>Organic chicken noodle</item>
   </string-array>
</resources>
```

As shown earlier in this chapter, accessing string arrays resources is easy. The following code retrieves a string array named flavors:

```
String[] aFlavors =
     getResources().getStringArray(R.array.flavors);
```

Working with Boolean Resources

Other primitive types are supported by the Android resource hierarchy as well. Boolean resources can be used to store information about application game preferences and default values. Boolean resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

Defining Boolean Resources in XML

Boolean values are appropriately tagged with the <bool> tag and represent a name value pair. The name attribute is how you refer to the specific Boolean value programmatically, so name these resources wisely.

Here's an example of the Boolean resource file /res/values/bools.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <bool  name="bOnePlusOneEqualsTwo">true</bool>
     <bool name="bAdvancedFeaturesEnabled">false</bool>
</resources>
```

Using Boolean Resources Programmatically

To use a Boolean resource,you must load it using the Resource class.The following code accesses your application's Boolean resource named bAdvancedFeaturesEnabled.

```
boolean bAdvancedMode =
   getResources().getBoolean(R.bool.bAdvancedFeaturesEnabled);
```

Working with Integer Resources

In addition to strings and Boolean values, you can also store integers as resources.Integer resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

Defining Integer Resources in XML

Integer values are appropriately tagged with the <integer> tag and represent a name value pair. The name attribute is how you refer to the specific integer programmatically, so name these resources wisely.

Here's an example of the integer resource file /res/values/nums.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <integer name="numTimesToRepeat">25</integer>
     <integer name="startingAgeOfCharacter">3</integer>
</resources>
```

Using Integer Resources Programmatically

To use the integer resource, you must load it using the Resource class. The following code accesses your application's integer resource named numTimesToRepeat:

int repTimes =
getResources().getInteger(R.integer.numTimesToRepeat);

Working with Colors

Android applications can store RGB color values, which can then be applied to other screen elements. You can use these values to set the color of text or other elements, such as the screen background. Color resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

RGB color values always start with the hash symbol (#).The alpha value can be given for transparency control. The following color formats are supported:

- #RGB (example, #F00 is 12-bit color, red)
- #ARGB example, #8F00 is 12-bit color, red with alpha 50%)
- #RRGGBB (example, #FF00FF is 24-bit color, magenta)
- #AARRGGBB example, #80FF00FF is 24-bit color, magenta with alpha 50%)

Color values are appropriately tagged with the <color> tag and represent a name-value pair. Here’s an example of a simple color resource file /res /values /colors .xml:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
     <color name=”background_color”>#006400</color>
     <color name=”text_color”>#FFE4C4</color>
</resources>
```

The example at the beginning of the chapter accessed a color resource. Color resources are simply integers.The following code retrieves a color resource called prettyTextColor:

int myResourceColor
=getResources().getColor(R.color.prettyTextColor);

Working with Dimensions

Many user interface layout controls such as text controls and buttons are drawn to specific dimensions. These dimensions can be stored as resources. Dimension values always end with a unit of measurement tag.

Dimension values are appropriately tagged with the <dimen> tag and represent a name value pair. Dimension resources are defined in XML under the /res/values project directory and compiled into the application package at build time.

The dimension units supported are shown in Table.

Dimension Unit Measurements Supported

Unit of Measurement	Description	Resource Tag Required	Example
Pixels	Actual screen pixels	px	20px
Inches	Physical measurement	in	1in
Millimeters	Physical measurement	mm	1mm
Points	Common font measurement unit	pt	14pt
Screen density independent pixels	Pixels relative to 160dpi screen (preferable dimension for screen compatibility)	dp	1dp
Scale independent pixels	Best for scalable font display	sp	14sp

Here's an example of a simple dimension resource file /res/values/dimens.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <dimen name="FourteenPt">14pt</dimen>
     <dimen name="OneInch">1in</dimen>
     <dimen  name="TenMillimeters">10mm</dimen>
     <dimen name="TenPixels">10px</dimen>
</resources>
```

Dimension resources are simply floating point values.The following code retrieves a dimension resource called textPointSize:

```
float myDimension =
   getResources().getDimension(R.dimen.textPointSize);
```

Working with Simple Drawables

You can specify simple colored rectangles by using the drawable resource type, which can then be applied to other screen elements. These drawable types are defined in specific paint colors, much like the Color resources are defined.

Simple paintable drawable resources are defined in XML under the /res/values project directory and compiled into the application package at build time. Paintable drawable resources use the <drawable> tag and represent a name-value pair. Here's an example of a simple drawable resource file /res/values/drawables.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <drawable name="red_rect">#F00</drawable>
</resources>
```

Although it might seem a tad confusing, you can also create XML files that describe other Drawable subclasses, such as ShapeDrawable. Drawable XML definition files are stored in the

/res/drawable directory within your project along with image files. This is not the same as storing <drawable> resources, which are paintable drawables.PaintableDrawable resources are stored in the /res/values directory, as explained in the previous section.

Here's a simple ShapeDrawable described in the file /res/drawable/red_oval.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<shape
     xmlns:android="http://schemas.android.com/apk/res/android"
     android:shape="oval">
     <solid android:color="#f00"/>
</shape>
```

We talk more about graphics and drawing shapes in Chapter "Drawing and Working with Animation."

Drawable resources defined with <drawable> are simply rectangles of a given color,which is represented by the Drawable subclass ColorDrawable. The following code retrieves a ColorDrawable resource called redDrawable:

```
import android.graphics.drawable.ColorDrawable;
...
ColorDrawable myDraw = (ColorDrawable)getResources().
getDrawable(R.drawable.redDrawable);
```

Working with Images

Applications often include visual elements such as icons and graphics. Android supports several image formats that can be directly included as resources for your application. These image formats are shown in Table.

Image Formats Supported in Android

Supported Image Format	Description	Required Extension
Portable Network Graphics (PNG)	Preferred Format (Lossless)	.png
Nine-Patch Stretchable Images	Preferred Format (Lossless)	.9.png
Joint Photographic Experts Group (JPEG)	Acceptable Format (Lossy)	.jpg, .jpeg
Graphics Interchange Format (GIF)	Discouraged Format	.gif

These image formats are all well supported by popular graphics editors such as Adobe Photoshop, GIMP, and Microsoft Paint. The Nine-Patch Stretchable Graphics can be created from PNG files using the draw9patch tool included with the Android SDK under the /tools directory.

Adding image resources to your project is easy. Simply drag the image asset into the /res/drawable directory, and it is automatically included in the application package at build time.

Working with Nine-Patch Stretchable Graphics

Phone screens come in various dimensions. It can be handy to use stretchable graphics to allow a single graphic that can scale appropriately for different screen sizes and orientations or different lengths of text. This can save you or your designer a lot of time in creating graphics for many different screen sizes.

Android supports Nine-Patch Stretchable Graphics for this purpose. Nine-Patch graphics are simply PNG graphics that have patches, or areas of the image, defined to scale appropriately, instead of scaling the entire image as one unit. Often the center segment is transparent.

Nine-Patch Stretchable Graphics can be created from PNG files using the draw9patch tool included with the Tools directory of the Android SDK.

Using Image Resources Programmatically

Images resources are simply another kind of Drawable called a BitmapDrawable. Most of the time, you need only the resource ID of the image to set as an attribute on a user interface control.

For example,if I drop the graphics file flag.png into the /res /drawable directory and add an Image View control to my main layout,we can set the image to be displayed programmatically in the layout this way:

```
import android.widget.ImageView;
...
ImageView flagImageView =
(ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
```

If you want to access the BitmapDrawable object directly, you simply request that resource directly, as follows:

```
import android.graphics.drawable.BitmapDrawable;
...
BitmapDrawable bitmapFlag = (BitmapDrawable)
getResources().getDrawable(R.drawable.flag);
int iBitmapHeightInPixels =
bitmapFlag.getIntrinsicHeight();
int iBitmapWidthInPixels = bitmapFlag.getIntrinsicWidth();
```

Finally, if you work with Nine-Patch graphics, the call to getDrawable()returns a Nine Patch Drawable instead of a Bitmap Drawable object.

```
import android.graphics.drawable.NinePatchDrawable;
...
NinePatchDrawable stretchy = (NinePatchDrawable)
getResources().getDrawable(R.drawable.pyramid);
int iStretchyHeightInPixels =
stretchy.getIntrinsicHeight();
int iStretchyWidthInPixels = stretchy.getIntrinsicWidth();
```

Working with Animation

Android supports frame-by-frame animation and tweening. Frame-by-frame animation involves the display of a sequence of images in rapid succession. Tweened animation involves applying standard graphical transformations such as rotations and fades upon a single image.

The Android SDK provides some helper utilities for loading and using animation resources. These utilities are found in the android. view. animation. AnimationUtils class.

For now, let's just look at how you define animation data in terms of resources.

Defining and Using Frame-by-Frame Animation Resources

Frame-by-frame animation is often used when the content changes from frame to frame. This type of animation can be used for complex frame transitions—much like a kid's flip-book.

To define frame-by-frame resources, take the following steps:

1. Save each frame graphic as an individual drawable resource. It may help to name your graphics sequentially, in the order in which they are displayed—for example, frame1.png, frame2.png, and so on.
2. Define the animation set resource in an XML file within /res/drawable/ resource directory.
3. Load, start, and stop the animation programmatically.

Here's an example of a simple frame-by-frame animation resource file /res /drawable/ juggle .xml that defines a simple three-frame animation that takes 1.5 seconds:

```
<?xml version=”1.0” encoding=”utf-8” ?>
<animation-list
     xmlns:android=”http://schemas.android.com/apk/res/android”
            android:oneshot=”false”>
     <item
             android:drawable=”@drawable/splash1”
            android:duration=”50” />
     <item
            android:drawable=”@drawable/splash2”
            android:duration=”50” />
     <item
            android:drawable=”@drawable/splash3”
            android:duration=”50” />
</animation-list>
```

Frame-by-frame animation set resources defined with <animation-list> are represented by the Drawable subclass AnimationDrawable.The following code retrieves an Animation-Drawable resource called juggle:

```
import android.graphics.drawable.AnimationDrawable;
  ...
AnimationDrawable jugglerAnimation =
(AnimationDrawable)getResources().
getDrawable(R.drawable.juggle);
```

After you have a valid AnimationDrawable, you can assign it to a View on the screen and use the Animation methods to start and stop animation.

Defining and Using Tweened Animation Resources

Tweened animation features include scaling, fading, rotation, and translation. These actions can be applied simultaneously or sequentially and might use different interpolators. Tweened animation sequences are not tied to a specific graphic file, so you can write one sequence and then use it for a variety of different graphics. For example, you can make moon, star, and diamond graphics all pulse using a single scaling sequence, or you can make them spin using a rotate sequence.

Graphic animation sequences can be stored as specially formatted XML files in the /res/anim directory and are compiled into the application binary at build time.

Here's an example of a simple animation resource file /res/anim/spin.xml that defines a simple rotate operation—rotating the target graphic counterclockwise four times in place, taking 10 seconds to complete:

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android="http://schemas.android.com/apk/res/android"
     android:shareInterpolator="false">
<set>
     <rotate
            android:fromDegrees="0"
            android:toDegrees="-1440"
            android:pivotX="50%"
            android:pivotY="50%"
            android:duration="10000" />
</set>
</set>
```

If we go back to the example of a BitmapDrawable earlier, we can now add some animation simply by adding the following code to load the animation resource file spin.xml and set the animation in motion:

```
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;
...
ImageView flagImageView =
(ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
...
Animation an =
AnimationUtils.loadAnimation(this, R.anim.spin);
flagImageView.startAnimation(an);
```

Now you have your graphic spinning. Notice that we loaded the animation using the base class object Animation. You can also extract specific animation types using the subclasses that match: RotateAnimation, ScaleAnimation, TranslateAnimation, and AlphaAnimation.

There are a number of different interpolators you can use with your tweened animation sequences.

Working with Menus

You can also include menu resources in your project files. Like animation resources, menu resources are not tied to a specific control but can be reused in any menu control.

Each menu resource (which is a set of individual menu items) is stored as a specially formatted XML files in the /res/menu directory and are compiled into the application package at build time.

Here’s an example of a simple menu resource file /res/menu/speed.xml that defines a short menu with four items in a specific order:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
     <item
            android:id="@+id/start"
            android:title="Start!"
            android:orderInCategory="1"></item>
     <item
            android:id="@+id/stop"
            android:title="Stop!"
            android:orderInCategory="4"></item>
     <item
            android:id="@+id/accel"
            android:title="Vroom! Accelerate!"
            android:orderInCategory="2"></item>
     <item
            android:id="@+id/decel"
            android:title="Decelerate!"
            android:orderInCategory="3"></item>
     </menu>
```

You can create menus using the Eclipse plug-in, which can access the various configuration attributes for each menu item. In the previous case, we set the title (label)of each menu item and the order in which the items display. Now, you can use string resources for those titles, instead of typing in the strings. For example:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
     <item
            android:id="@+id/start"
            android:title="@string/start"
            android:orderInCategory="1"></item>
     <item
            android:id="@+id/stop"
            android:title="@string/stop"
            android:orderInCategory="2"></item>
      </menu>
```

To access the preceding menu resource called /res/menu/speed.xml, simply override the

```
method onCreateOptionsMenu() in your application:

 public boolean onCreateOptionsMenu(Menu menu) {
      getMenuInflater().inflate(R.menu.speed, menu);
     return true;
}
```

That’s it. Now if you run your application and press the menu button, you see the menu. There are a number of other XML attributes that can be assigned to menu items. For a complete list of these attributes, see the Android SDK reference for menu resources at the website. You learn a lot more about menus and menu event handling in Chapter “Exploring User Interface Screen Elements.”

Working with XML Files

You can include arbitrary XML resource files to your project. You should store these XML files in the /res/xml directory, and they are compiled into the application package at build time.

The Android SDK has a variety of packages and classes available for XML manipulation. You learn more about XML handling in Chapter “Using Android Data and Storage APIs,” Chapter “Sharing Data Between Applications with Content Providers,” and Chapter “Using Android Networking APIs.” For now, we create an XML resource file and access it through code.

First, put a simple XML file in /res/xml directory. In this case, the file my_pets.xml with the following contents can be created:

```
<?xml version=”1.0” encoding=”utf-8”?>
<pets>
     <pet name=”Bit” type=”Bunny” />
     <pet name=”Nibble” type=”Bunny” />
     <pet name=”Stack” type=”Bunny” />
     <pet name=”Queue” type=”Bunny” />
     <pet name=”Heap” type=”Bunny” />
     <pet name="Null" type="Bunny" />
     <pet name="Nigiri" type="Fish" />
     <pet name="Sashimi II" type="Fish" />
     <pet name="Kiwi" type="Lovebird" />
</pets>
```

Now you can access this XML file as a resource programmatically in the following manner:

```
     XmlResourceParser  myPets =
     getResources().getXml(R.xml.my_pets);
```

Finally, to prove this is XML, here's one way you might churn through the XML and extract the information:

```
import org.xmlpull.v1.XmlPullParserException;
import android.content.res.XmlResourceParser;
...
int eventType = -1;
while (eventType != XmlResourceParser.END_DOCUMENT) {
     if(eventType ==  XmlResourceParser.START_DOCUMENT) {
            Log.d(DEBUG_TAG, "Document  Start");
     } else if(eventType == XmlResourceParser.START_TAG)  {
            String strName = myPets.getName();
            if(strName.equals("pet")) {
            Log.d(DEBUG_TAG, "Found a PET");
            Log.d(DEBUG_TAG,"Name:"+myPets.getAttributeValue(
                 null,"name"));
            Log.d(DEBUG_TAG, "Species:"+myPets.getAttributeValue(
                 null,"type"));
     }
  }
  eventType = myPets.next();
  }
  Log.d(DEBUG_TAG,  "Document End");
```

Working with Raw Files

Your application can also include raw files as part of its resources. For example, your application might use raw files such as audio files, video files, and other file formats not supported by the Android Resource packaging tool aapt.

All raw resource files are included in the /res/raw directory and are added to your package without further processing.

The resource filename must be unique to the directory and should be descriptive because the filename (without the extension) becomes the name by which the resource is accessed.

You can access raw file resources and any resource from the /res/drawable directory (bitmap graphics files, anything not using the <resource> XML definition method). Here’s one way to open a file called the_help.txt:

```
import java.io.InputStream;
...
InputStream  iFile =
getResources().openRawResource(R.raw.the_help);
```

References to Resources

You can reference resources instead of duplicating them. For example, your application might want to reference a single string resource in multiple string arrays.

The most common use of resource references is in layout XML files, where layouts can reference any number of resources to specify attributes for layout colors, dimensions, strings, and graphics.Another common use is within style and theme resources.

Resources are referenced using the following format:

]resource_type/variable_name

Recall that earlier we had a string-array of soup names. If we want to localize the soup listing, a better way to create the array is to create individual string resources for each soup name and then store the references to those string resources in the string-array (instead of the text).

To do this,we define the string resources in the /res/strings.xml file like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <string name="app_name">Application  Name</string>
     <string name="chicken_soup">Organic Chicken Noodle</string>
     <string name="minestrone_soup">Veggie Minestrone</string>
     <string name="chowder_soup">New England LobsterChowder</string>
</resources>
```

And then we can define a localizable string-array that references the string resources by name in the /res/arrays.xml file like this:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <string-array name="soups">
            <item>@string/minestrone_soup</item>
            <item>@string/chowder_soup</item>
            <item>@string/chicken_soup</item>
     </string-array>
</resources>
```

You can also use references to make aliases to other resources. For example, you can alias the system resource for the OK string to an application resource name by including the following in your strings.xml resource file:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
     <string id="app_ok">@android:string/ok</string>
</resources>
```

You learn more about all the different system resources available later in this chapter.

Working with Layouts

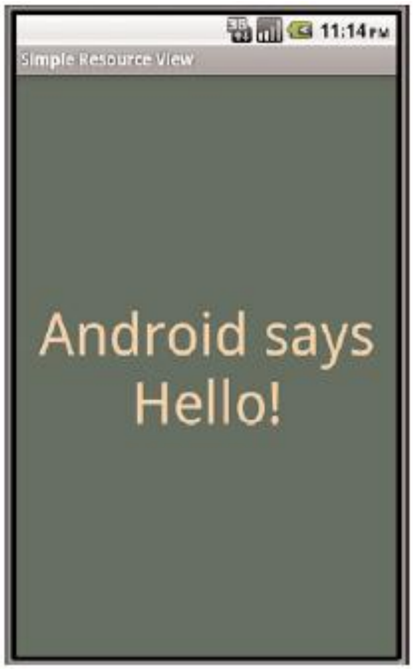
Much as web designers use HTML, user interface designers can use XML to define Android application screen elements and layout. A layout XML resource is where many different resources come together to form the definition of an Android application screen. Layout resource files are included in the /res/layout/ directory and are compiled into the application package at build time. Layout files might include many user interface controls and define the layout for an entire screen or describe custom controls used in other layouts.

Here's a simple example of a layout file (/res/layout/main.xml) that sets the screen's back ground color and displays some text in the middle of the screen.

The main.xml layout file that displays this screen references a number of other resources, including colors, strings, and dimension values, all of which were defined in the strings.xml, colors.xml, and dimens.xml resource files.The color resource for the screen background color and resources for a TextView control's color, string, and text size follows:

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout  xmlns:android
       =“http://schemas.android.com/apk/res/android”
     android:orientation=”vertical”
     android:layout_width=”fill_parent”
     android:layout_height=”fill_parent”
     android:background=”@color/background_color”>
<TextView
     android:id=”@+id/TextView01”
     android:layout_width=”fill_parent”
     android:layout_height=”fill_parent”
     android:text="@string/test_string"
     android:textColor="@color/text_color"
     android:gravity="center"
     android:textSize="@dimen/text_size"></TextView>
</LinearLayout>
```

main.xml layout file displays in the emulator.

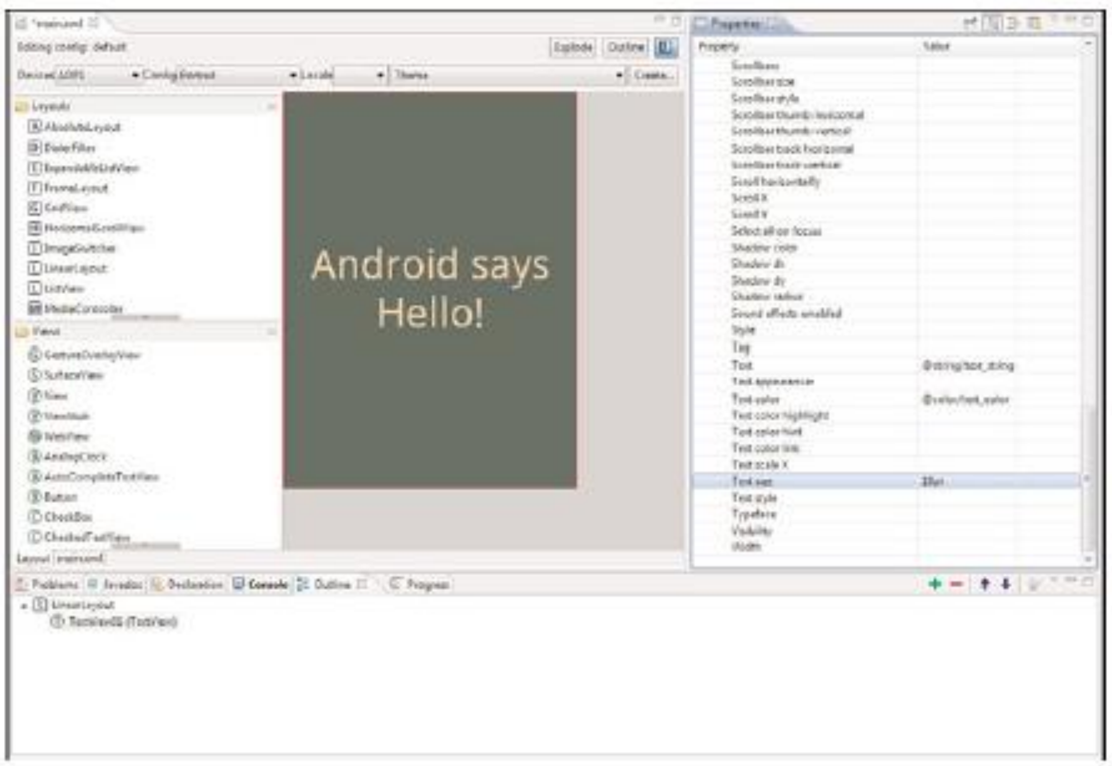


The preceding layout describes all the visual elements on a screen. In this example, a LinearLayout control is used as a container for other user interface controls—here, a single TextView that displays a line of text.

Designing Layouts in Eclipse

Layouts can be designed and previewed in Eclipse using the Resource editor functionality provided by the Android plug-in. If you click the project file /res /layout /main.xml (provided with any new Android project), you see a Layout tab, which shows you the preview of the layout, and a main.xml tab, which shows you the raw XML of the layout file.

Designing a layout file using Eclipse.



As with most user interface designers, the Android plug-in works well for your basic layout needs, enables you to create user interface controls such as TextView and Button controls easily, and enables setting the controls' properties in the Properties pane.

Now is a great time to get to know the layout resource designer. Try creating a new Android project called ParisView (available as a sample project). Navigate to the /res/layout/main.xml layout file and double-click it to open it in the resource editor. It's quite simple by default, only a black (empty) rectangle and string of text.

Below in the Resource pane of the Eclipse perspective, you notice the Outline tab. This outline is the XML hierarchy of this layout file. By default, you see a LinearLayout.

If you expand it, you see it contains one TextView control. Click on the TextView control. You see that the Properties pane of the Eclipse perspective now has all the properties available for that object. If you scroll down to the property called text, you see that it’s set to a string resource variable @string/hello.

You can use the layout designer to set and preview layout control properties. For example, you can modify the TextView property called text Size by typing 18pt (a dimension). You see the results of your change to the property immediately in the preview area.

Take a moment to switch to the main.xml tab. You notice that the properties you set are now in the XML. If you save and run your project in the emulator now, you see similar results to what you see in the designer preview.

Now go back to the Outline pane. You see a green plus and a red minus button. You can use these buttons to add and remove controls to your layout file. For example, select the LinearLayout from the Outline view, and click the green button to add a control within that container object.

Choose the ImageView object. Now you have a new control in your layout. You can’t actually see it yet because it is not fully defined.

Drag two PNG graphics files (or JPG) into your /res/drawable project directory, naming them flag.png and background.png.Now, browse the properties of your ImageView control, and set the Src property by clicking on the resource browser button labeled [...].You can browse all the Drawable resources in your project and select the flag resource you just added. You can also set this property manually by typing @drawable/flag.

Now, you see that the graphic shows up in your preview. While we're at it, select the LinearLayout object and set its background property to the background Drawable you added.

If you save the layout file and run the application in the emulator or on the phone, you see results much like you did in the resource designer preview pane.

Using Layout Resources Programmatically

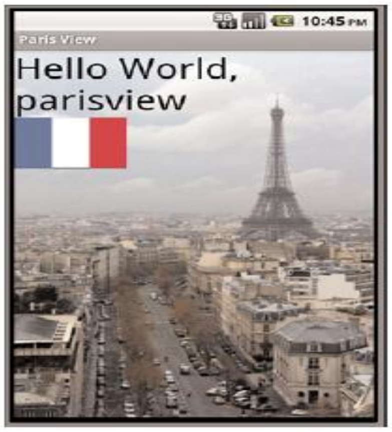
Layouts, whether they are Button or ImageView controls, are all derived from the View class. Here's how you would retrieve a TextView object named TextView01:

TextView txt = (TextView)findViewById(R.id.TextView01);

You can also access the underlying XML of a layout resource much as you would any XML file. The following code retrieves the main.xml layout file for XML parsing:

```
XmlResourceParser myMainXml =
getResources().getLayout(R.layout.main);
```

A layout with a LinearLayout, TextView, and ImageView, shown in the Android emulator.



Developers can also define custom layouts with unique attributes. We talk much more about layout files and designing Android user interfaces in Chapter "Designing User Interfaces with Layouts."

Working with Styles

Android user interface designers can group layout element attributes together in styles. Layout controls are all derived from the View base class, which has many useful attributes. Individual controls, such as Checkbox, Button, and TextView, have specialized attributes associated with their behavior.

Styles are tagged with the <style> tag and should be stored in the /res/values/ directory. Style resources are defined in XML and compiled into the application binary at build time.

Styles cannot be previewed using the Eclipse Resource designer but they are displayed correctly in the emulator and on the device.

Here’s an example of a simple style resource file /res/values/styles.xml containing two styles: one for mandatory form fields, and one for optional form fields on TextView and EditText objects:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
     <style name=”mandatory_text_field_style”>
            <item name=”android:textColor”>#000000</item>
            <item name=”android:textSize”>14pt</item>
            <item name=”android:textStyle”>bold</item>
     </style>
     <style name=”optional_text_field_style”>
            <item name=”android:textColor”>#0F0F0F</item>
            <item name=”android:textSize”>12pt</item>
            <item name=”android:textStyle”>italic</item>
     </style>
</resources>
```

Many useful style attributes are colors and dimensions. It would be more appropriate to use references to resources. Here's the styles.xml file again; this time, the color and text size fields are available in the other resource files colors.xml and dimens.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <style name="mandatory_text_field_style">
     <item name="android:textColor">@color/mand_text_color</item>
     <item name="android:textSize">@dimen/important_text</item>
     <item name="android:textStyle">bold</item>
   </style>
   <style name="optional_text_field_style">
     <item name="android:textColor">@color/opt_text_color</item>
     <item name="android:textSize">@dimen/unimportant_text</item>
     <item name="android:textStyle">italic</item>
   </style>
</resources>
```

Now, if you can create a new layout with a couple of TextView and EditText text controls, you can set each control's style attribute by referencing it as such:

```
style="@style/name_of_style"
```

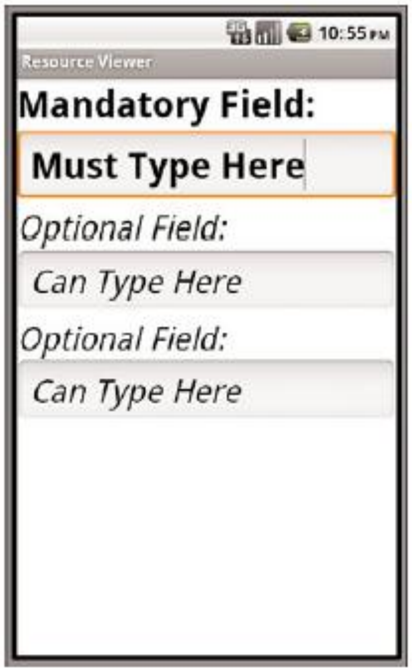
Here we have a form layout called /res/layout/form.xml that does that:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:background="@color/background_color">
   <TextView
     android:id="@+id/TextView01"
     style="@style/mandatory_text_field_style"
     android:layout_height="wrap_content"
     android:text="@string/mand_label"
     android:layout_width="wrap_content" />
   <EditText
     android:id="@+id/EditText01"
     style="@style/mandatory_text_field_style"
     android:layout_height="wrap_content"
```

```
      android:text="@string/mand_default"
      android:layout_width="fill_parent"
      android:singleLine="true" />
   <TextView
      android:id="@+id/TextView02"
      style="@style/optional_text_field_style"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/opt_label" />
   <EditText
      android:id="@+id/EditText02"
      style="@style/optional_text_field_style"
      android:layout_height="wrap_content"
      android:text="@string/opt_default"
      android:singleLine="true"
       android:layout_width="fill_parent" />
   <TextView
      android:id="@+id/TextView03"
      style="@style/optional_text_field_style"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/opt_label" />
   <EditText
      android:id="@+id/EditText03"
      style="@style/optional_text_field_style"
      android:layout_height="wrap_content"
      android:text="@string/opt_default"
      android:singleLine="true"
      android:layout_width="fill_parent" />
</LinearLayout>
```

The resulting layout has three fields, each made up of one TextView for the label and one EditText where the user can input text. The mandatory style is applied to the mandatory label and text entry. The other two fields use the optional style. The resulting layout would look something like Figure.

***A layout using two styles, one for mandatory fields and another for optional fields*.**



Using Style Resources Programmatically

Styles are applied to specific layout controls such as TextView and Button objects. Usually, you want to supply the style resource id when you call the control's constructor. For example, the style named myAppIsStyling would be referred to as R.style.myAppIsStyling.

Working with Themes

Themes are much like styles, but instead of being applied to one layout element at a time, they are applied to all elements of a given activity (which, generally speaking, means one screen).

Themes are defined in exactly the same way as styles. Themes use the <style> tag and should be stored in the /res/values directory. The only difference is that instead of applying that named style to a layout element, you define it as the theme attribute of an activity in the Android Manifest .xml file.

Referencing System Resources

You can access system resources in addition to your own resources. The android package contains all kinds of resources, which you can browse by looking in the android.R subclasses.

Here you find system resources for

- Animation sequences for fading in and out
- Arrays of email/phone types (home, work, and such)
- Standard system colors
- Dimensions for application thumbnails and icons
- Many commonly used drawable and layout types
- Error strings and standard button text
- System styles and themes

You can reference system resources the same way you use your own; set the package name to android. For example, to set the background to the system color for darker gray, you set the appropriate background color attribute to @android:color/darker_gray.

You can access system resources much like you access your application's resources. Instead of using your application resources, use the Android package's resources under the android.R class.

If we go back to our animation example, we could have used a system animation instead of defining our own. Here is the same animation example again, except it uses a system animation to fade in:

```
impor android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;
...
ImageView flagImageView =
     (ImageView)findViewById(R.id.ImageView01);
flagImageView.setImageResource(R.drawable.flag);
...
Animation an = AnimationUtils.
    loadAnimation(this, android.R.anim.fade_in);
flagImageView.startAnimation(an);
```

Exploring User Interface Screen Elements

Exploring User Interface Screen Elements

Most Android applications inevitably need some form of user interface. In this chapter, we discuss the user interface elements available within the Android Software Development Kit (SDK). Some of these elements display information to the user, whereas others gather information from the user.

You learn how to use a variety of different components and controls to build a screen and how your application can listen for various actions performed by the user. Finally, you learn how to style controls and apply themes to entire screens.

Introducing Android Views and Layouts

Before we go any further, we need to define a few terms. This gives you a better understanding of certain capabilities provided by the Android SDK before they are fully introduced. First, let's talk about the View and what it is to the Android SDK.

Introducing the Android View

The Android SDK has a Java packaged named android.view. This package contains a number of interfaces and classes related to drawing on the screen. However, when we refer to the View object, we actually refer to only one of the classes within this package: the android.view.View class.

The View class is the basic user interface building block within Android. It represents a rectangular portion of the screen. The View class serves as the base class for nearly all the user interface controls and layouts within the Android SDK.

Introducing the Android Control

The Android SDK contains a Java package named android.widget. When we refer to controls, we are typically referring to a class within this package. The Android SDK includes classes to draw most common objects, including ImageView, FrameLayout, EditText, and Button classes. As mentioned previously, all controls are typically derived from the View class.

This chapter is primarily about controls that display and collect data from the user. We cover many of these basic controls in detail.

Introducing the Android Layout

One special type of control found within the android.widget package is called a layout. A layout control is still a View object, but it doesn't actually draw anything specific on the screen. Instead, it is a parent container for organizing other controls (children). Layout controls determine how and where on the screen child controls are drawn. Each type of layout control draws its children using particular rules. For instance, the LinearLayout control draws its child controls in a single horizontal row or a single vertical column. Similarly, a TableLayout control displays each child control in tabular format (in cells within specific rows and columns).

In Chapter "Designing User Interfaces with Layouts," we organize various controls within layouts and other containers. These special View controls, which are derived from the android.view.ViewGroup class, are useful only after you understand the various display controls these containers can hold. By necessity, we use some of the layout View objects within this chapter to illustrate how to use the controls previously mentioned. However, we don’t go into the details of the various layout types available as part of the Android SDK until the next chapter.

Displaying Text to Users with TextView

One of the most basic user interface elements, or controls, in the Android SDK is the TextView control. You use it, quite simply, to draw text on the screen. You primarily use it to display fixed text strings or labels.

Frequently, the TextView control is a child control within other screen elements and controls. As with most of the user interface elements, it is derived from View and is within the android.widget package. Because it is a View, all the standard attributes such as width, height, padding, and visibility can be applied to the object. However, as a text-displaying control, you can apply many other TextView-specific attributes to control behavior and how the text is viewed in a variety of situations.

First, though, let’s see how to put some quick text up on the screen. <TextView> is the XML layout file tag used to display text on the screen. You can set the android:text property of the TextView to be either a raw text string in the layout file or a reference to a string resource.

Here are examples of both methods you can use to set the android:text attribute of a TextView. The first method sets the text attribute to a raw string; the second method uses a string resource called sample_text, which must be defined in the strings.xml resource file.

```
<TextView
     android:id="@+id/TextView01"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Some sample text here" />
<TextView
     android:id="@+id/TextView02"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/sample_text" />
```

To display this TextView on the screen, all your Activity needs to do is call the setContentView() method with the layout resource identifier in which you defined the preceding XML shown.You can change the text displayed programmatically by calling the setText() method on the TextView object. Retrieving the text is done with the< getText() method.

Now let’s talk about some of the more common attributes of TextView objects.

Configuring Layout and Sizing

The TextView control has some special attributes that dictate how the text is drawn and flows. You can, for instance, set the TextView to be a single line high and a fixed width. If, however, you put a long string of text that can’t fit, the text truncates abruptly. Luckily, there are some attributes that can handle this problem.

The width of a TextView can be controlled in terms of the ems measurement rather than in pixels. An em is a term used in typography that is defined in terms of the point size of a particular font. (For example, the measure of an em in a 12-point font is 12 points.)

This measurement provides better control over how much text is viewed, regardless of the font size.Through the ems attribute, you can set the width of the Text View.Additionally, you can use the maxEms and minEms attributes to set the maximum width and minimum width, respectively, of the TextView in terms of ems.

The height of a TextView can be set in terms of lines of text rather than pixels.Again, this is useful for controlling how much text can be viewed regardless of the font size. The lines attribute sets the number of lines that the TextView can display.You can also use maxLines and minLines to control the maximum height and minimum height, respectively, that the Textview displays.

Here is an example that combines these two types of sizing attributes. This TextView is two lines of text high and 12 ems of text wide. The layout width and height are specified to the size of the TextView and are required attributes in the XML schema:

```
<TextView
     android:id="@+id/TextView04"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:lines="2"
     android:ems="12"
     android:text="@string/autolink_test" />
```

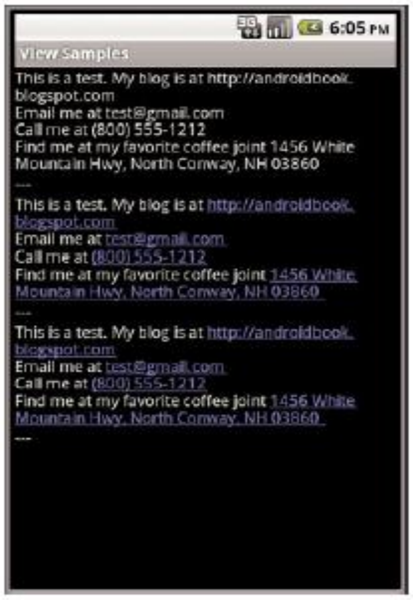
Instead of having the text only truncate at the end, as happens in the preceding example, we can enable the ellipsize attribute to replace the last couple characters with an ellipsis (...) so the user knows that not all text is displayed.

Creating Contextual Links in Text

If your text contains references to email addresses, web pages, phone numbers, or even street addresses, you might want to consider using the attribute autoLink. The autoLink attribute has four values that you can use in combination with each other. When enabled, these autoLink attribute values create standard web-style links to the application that can act on that data type. For instance, setting the attribute to web automatically finds and links any URLs to web pages.

Your text can contain the following values for the autoLink attribute:

- **none:** Disables all linking.
- **web:** Enables linking of URLs to web pages.
- **email:** Enables linking of email addresses to the mail client with the recipient filled.
- **phone:** Enables linking of phone numbers to the dialer application with the phone number filled out, ready to be dialed.
- **map:** Enables linking of street addresses to the map application to show the location.
- **all:** Enables all types of linking.



```
This is a test. My blog is at http://androidbook.
blogspot.com
Email me at test@gmail.com
Call me at (800) 555-1212
Find me at my favorite coffee joint 1456 White
Mountain Hwy, North Conway, NH 03860
---
This is a test. My blog is at http://androidbook.
blogspot.com
Email me at test@gmail.com
Call me at (800) 555-1212
Find me at my favorite coffee joint 1456 White
Mountain Hwy, North Conway, NH 03860
---
This is a test. My blog is at http://androidbook.
blogspot.com
Email me at test@gmail.com
Call me at (800) 555-1212
Find me at my favorite coffee joint 1456 White
Mountain Hwy, North Conway, NH 03860
---
```

Turning on the autoLink feature relies on the detection of the various types within the Android SDK. In some cases, the linking might not be correct or might be misleading.

Three TextViews: Simple, AutoLink All (not clickable), and AutoLink All (clickable).

Here is an example that links email and

web pages, which, in our opinion, are the most reliable and predictable:

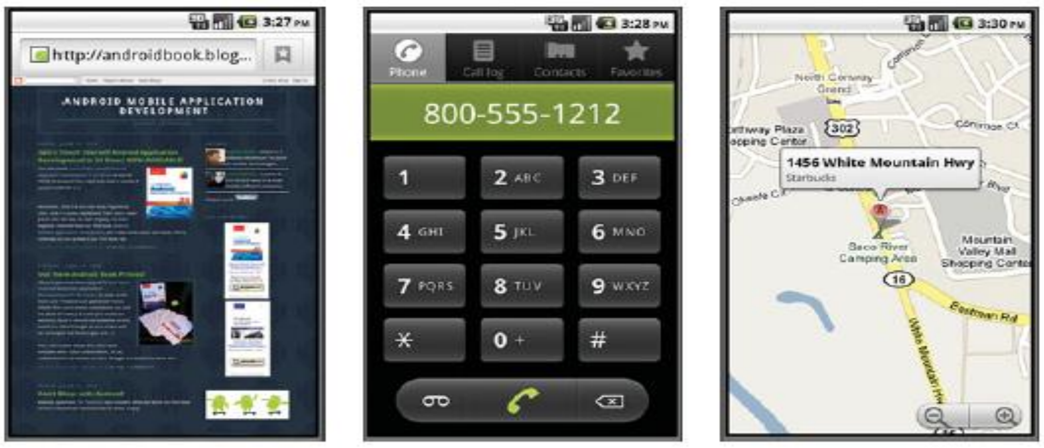
```
<TextView
     android:id="@+id/TextView02"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/autolink_test"
     android:autoLink="web|email" />
```

There are two helper values for this attribute, as well. You can set it to none to make sure no type of data is linked. You can also set it to all to have all known types linked. Figure illustrates what happens when you click on these links. The default for a TextView is not to link any types. If you want the user to see the various data types highlighted but you don't want the user to click on them, you can set the linksClickable attribute to false.

Retrieving Data from Users

The Android SDK provides a number of controls for retrieving data from users. One of the most common types of data that applications often need to collect from users is text. Two frequently used controls to handle this type of job are EditText controls and Spinner controls.

Clickable AutoLinks: URL launches browser, phone number launches dialer, and street address launches Google Maps.



Retrieving Text Input Using EditText Controls

The Android SDK provides a convenient control called EditText to handle text input from a user. The EditText class is derived from TextView. In fact, most of its functionality is contained within TextView but enabled when created as an EditText.The EditText object has a number of useful features enabled by default, many of which are shown in Figure.

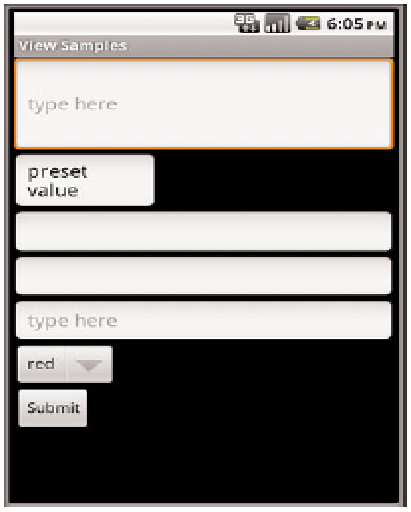
First, though, let’s see how to define an EditText control in an XML layout file:

```
<EditText
     android:id=”@+id/EditText01”
     android:layout_height=”wrap_content”
     android:hint=”type here”
     android:lines=”4”
     android:layout_width=”fill_parent” />
```

This layout code shows a basic EditText element. There are a couple of interesting things to note. First, the hint attribute puts some text in the edit box that goes away when the user starts entering text. Essentially, this gives a hint to the user as to what should go there. Next is the lines attribute, which defines how many lines tall the input box is. If this is not set, the entry field grows as the user enters text. However, setting a size allows the user to scroll within a fixed sized to edit the text. This also applies to the width of the entry.

By default, the user can perform a long press to bring up a context menu. This provides to the user some basic copy, cut, and paste operations as well as the ability to change the input method and add a word to the user's dictionary of frequently used words.You do not need to provide any additional code for this useful behavior to benefit your users. You can also highlight a portion of the text from code, too. A call to setSelection() does this, and a call to selectAll() highlights the entire text entry field.

Various styles of EditText controls and Spinner and Button controls.

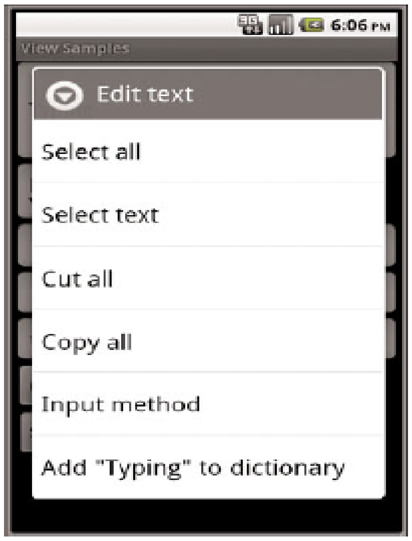


The EditText object is essentially an editable TextView.This means that you can read text from it in the same way as you did with TextView: by using the getText() method. You can also set initial text to draw in the text entry area using the setText() method. This is useful when a user edits a form that already has data. Finally, you can set the editable attribute to false, so the user cannot edit the text in the field but can still copy text out of it using a long press.

Helping the User with Auto Completion

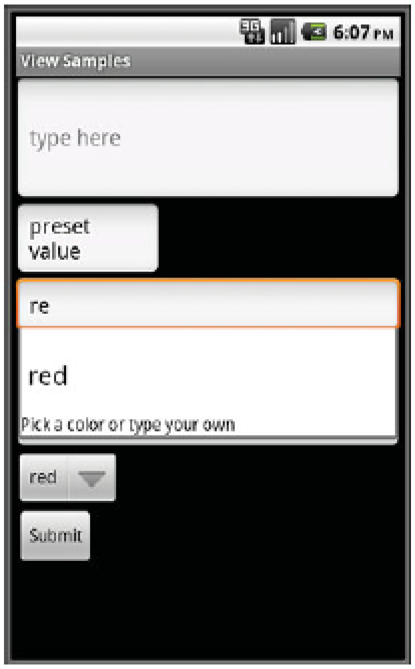
In addition to providing a basic text editor with the EditText control, the Android SDK also provides a way to help the user with entering commonly used data into forms. This functionality is provided through the auto-complete feature.

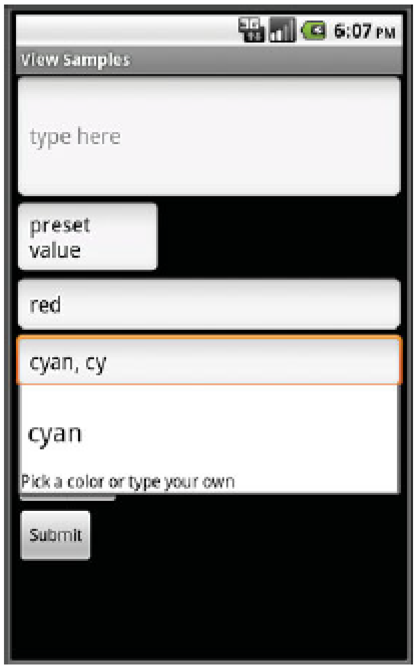
There are two forms of auto-complete. One is the more standard style of filling in the entire text entry based on what the user types. If the user begins typing a string that matches a word in a developer-provided list, the user can choose to complete the word with just a tap. This is done through the AutoCompleteTextView control. The second method allows the user to enter a list of items, each of which has autocomplete functionality. These items must be separated in some way by providing a Tokenizer to the MultiAutoComplete TextView object that handles this method. A common Tokenizer implementation is provided for comma-separated lists and is used by specifying the MultiAuto CompleteText View.CommaTokenizer object. This can be helpful for lists of specifying common tags and the like.



A long press on EditText controls typically launches a Context menu for Select, Cut, and Paste.

Using AutoCompleteTextView (left) and MultiAutoCompleteTextView (right).





Both of the auto-complete text editors use an adapter to get the list of text that they use to provide completions to the user. This example shows how to provide an AutoComplete TextView for the user that can help them type some of the basic colors from an array in the code:

```
final String[] COLORS = {
     "red", "green", "orange", "blue", "purple",
     "black", "yellow", "cyan", "magenta" };
ArrayAdapter<String> adapter =
     new ArrayAdapter<String>(this,
            android.R.layout.simple_dropdown_item_1line,
            COLORS);
AutoCompleteTextView text = (AutoCompleteTextView)
```

```
     findViewById(R.id.AutoCompleteTextView01);
text.setAdapter(adapter);
```

In this example, when the user starts typing in the field, if he starts with one of the letters in the COLORS array, a drop-down list shows all the available completions. Note that this does not limit what the user can enter. The user is still free to enter any text (such as puce).The adapter controls the look of the drop-down list. In this case, we use a built-in layout made for such things. Here is the layout resource definition for this

AutoCompleteTextView control:

```
<AutoCompleteTextView
     android:id="@+id/AutoCompleteTextView01"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:completionHint="Pick a color or type your  own"
     android:completionThreshold="1" />
```

There are a couple more things to notice here. First, you can choose when the completion drop-down list shows by filling in a value for the completionThreshold attribute. In this case, we set it to a single character, so it displays immediately if there is a match. The default value is two characters of typing before it displays auto-completion options. Second, you can set some text in the completionHint attribute. This displays at the bottom of the drop-down list to help users. Finally, the drop-down list for completions is sized to the TextView. This means that it should be wide enough to show the completions and the text for the completionHint attribute.

The MultiAutoCompleteTextView is essentially the same as the regular auto-complete, except that you must assign a Tokenizer to it so that the control knows where each autocompletion should begin. The following is an example that uses the same adapter as the previous example but includes a Tokenizer for a list of user color responses, each separated by a comma:

```
MultiAutoCompleteTextView mtext =
   (MultiAutoCompleteTextView)
findViewById(R.id.MultiAutoCompleteTextView01);
  mtext.setAdapter(adapter);
  mtext.setTokenizer(new MultiAutoCompleteTextView.CommaTokenizer());
```

As you can see, the only change is setting the Tokenizer. Here we use the built-in comma Tokenizer provided by the Android SDK. In this case, whenever a user chooses a color from the list, the name of the color is completed, and a comma is automatically added so that the user can immediately start typing in the next color. As before, this does not limit what the user can enter. If the user enters "maroon" and places a comma after it, the auto-completion starts again as the user types another color, regardless of the fact that it didn't help the user type in the color maroon. You can create your own Tokenizer by implementing the Multi Auto Complete Text View.Tokenizer interface. You can do this if you'd prefer entries separated by a semicolon or some other more complex separators.

Constraining User Input with Input Filters

There are often times when you don't want the user to type just anything. Validating input after the user has entered something is one way to do this. However, a better way to avoid wasting the user's time is to filter the input. The EditText control provides a way to set an InputFilter that does only this.

The Android SDK provides some InputFilter objects for use. There are InputFilter objects that enforce such rules as allowing only uppercase text and limiting the length of the text entered. You can create custom filters by implementing the InputFilter interface, which contains the single method called filter(). Here is an example of an EditText control with two built-in filters that might be appropriate for a two-letter state abbreviation:

```
final EditText text_filtered =
     (EditText) findViewById(R.id.input_filtered);
text_filtered.setFilters(new InputFilter[] {
     new InputFilter.AllCaps(),
     new InputFilter.LengthFilter(2)
});
```

The setFilters() method call takes an array of InputFilter objects. This is useful for combining multiple filters, as shown. In this case, we convert all input to uppercase. Additionally, we set the maximum length to two characters long. The EditText control looks the same as any other, but if you try to type lowercase, the text is converted to uppercase, and the string is limited to two characters. This does not mean that all possible inputs are valid, but it does help users to not concern themselves with making the input too long or bother with the case of the input. This also helps your application by guaranteeing that any text from this input is a length of two. It does not constrain the input to only letters, though. Input filters can also be defined in XML.

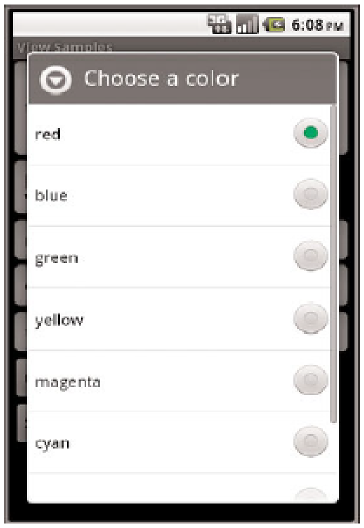
Giving Users Input Choices Using Spinner Controls

Sometimes you want to limit the choices available for users to type. For instance, if users are going to enter the name of a state, you might as well limit them to only the valid states because this is a known set. Although you could do this by letting them type something and then blocking invalid entries, you can also provide similar functionality with a Spinner control. As with the auto-complete method, the possible choices for a spinner can come from an Adapter. You can also set the available choices in the layout definition by using the entries attribute with an array resource (specifically a string-array that is referenced as something such as @array/state-list).The Spinner control isn't actually an EditText, although it is frequently used in a similar fashion. Here is an example of the XML layout definition for a Spinner control for choosing a color:

```
<Spinner
     android:id="@+id/Spinner01"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:entries="@array/colors"
     android:prompt="@string/spin_prompt" />
```

This places a Spinner control on the screen.When the user selects it, a pop-up shows the prompt text followed by a list of the possible choices. This list allows only a single item to be selected at a time, and when one is selected, the pop-up goes away.

There are a couple of things to notice here. First, the entries attribute is set to the values that shows by assigning it to an array resource, referred to here as @array/colors.



Filtering choices with a spinner control.

Second, the prompt attribute is defined to a string resource. Unlike some other string attributes, this one is required to be a string resource. The prompt displays when the popup comes up and can be used to tell the user what kinds of values that can be selected from.

Because the Spinner control is not a TextView, but a list of TextView objects, you can't directly request the selected text from it. Instead, you have to retrieve the selected View and extract the text directly:

```
final Spinner spin = (Spinner) findViewById(R.id.Spinner01);
TextView text_sel = (TextView)spin. getSelectedView();
String selected_text = text_sel.getText();
```

As it turns out, you can request the currently selected View object, which happens to be a TextView in this case. This enables us to retrieve the text and use it directly. Alternatively, we could have called the getSelectedItem() or getSelectedItemId() methods to deal with other forms of selection.

Using Buttons, Check Boxes, and Radio Groups

Another common user interface element is the button. In this section, you learn about different kinds of buttons provided by the Android SDK. These include the basic Button, ToggleButton, CheckBox, and RadioButton. You can find examples of each button type in Figure.

A basic Button is often used to perform some sort of action, such as submitting a form or confirming a selection. A basic Button control can contain a text or image label.

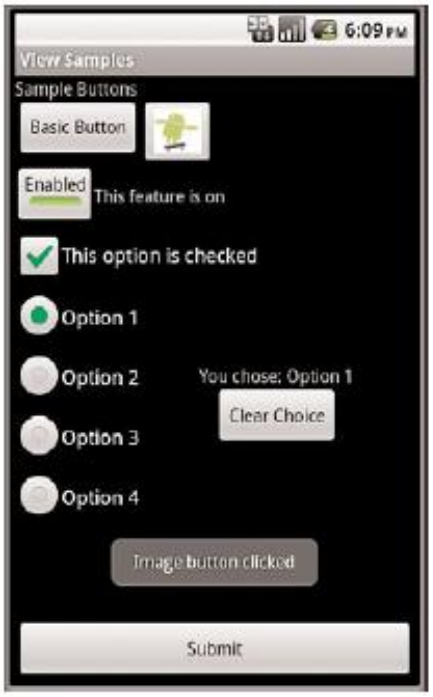
A CheckBox is a button with two states—checked or unchecked. You often use CheckBox controls to turn a feature on or off or to pick multiple items from a list.

A ToggleButton is similar to a CheckBox, but you use it to visually show the state. The default behavior of a toggle is like that of a power on/off button.

A RadioButton provides selection of an item. Grouping RadioButton controls together in a container called a RadioGroup enables the developer to enforce that only one RadioButton is selected at a time.

Using Basic Buttons

The android.widget.Button class provides a basic button implementation in the Android SDK. Within the XML layout resources, buttons are specified using the Button element. The primary attribute for a basic button is the text field. This is the label that appears on the middle of the button's face. You often use basic Button controls for buttons with text such as "Ok,""Cancel," or "Submit."



Various types of button controls.

The following XML layout resource file shows a typical Button control definition:

```
<Button
android:id="@+id/basic_button"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Basic Button" />
```

A button won't do anything, other than animate, without some code to handle the click event. Here is an example of some code that handles a click for a basic button and displays a Toast message on the screen:

```
setContentView(R.layout.buttons);
final Button basic_button = (Button)findViewById(R.id.basic_button);
basic_button.setOnClickListener(new View.OnClickListener() {
     public void onClick(View v) {
         Toast.makeText(ButtonsActivity.this,
            "Button clicked", Toast.LENGTH_SHORT).show();
     }
});
```

To handle the click event for when a button is pressed, we first get a reference to the Button by its resource identifier. Next, the setOnClickListener() method is called. It requires a valid instance of the class View.OnClickListener. A simple way to provide this is to define the instance right in the method call. This requires implementing the onClick() method. Within the onClick() method, you are free to carry out whatever actions you need. Here, we simply display a message to the users telling them that the button was, in fact, clicked.

A button with its primary label as an image is an ImageButton. An ImageButton is, for most purposes, almost exactly like a basic button. Click actions are handled in the same way. The primary difference is that you can set its src attribute to be an image. Here is an example of an ImageButton definition in an XML layout resource file:

```
<ImageButton
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/image_button"
android:src="@drawable/droid" />
```

In this case, a small drawable resource is referenced.

Using Check Boxes and Toggle Buttons

The check box button is often used in lists of items where the user can select multiple items. The Android check box contains a text attribute that appears to the side of the check box. This is used in a similar way to the label of a basic button. In fact, it's basically a TextView next to the button.

Here's an XML layout resource definition for a CheckBox control:

```
<CheckBox
android:id="@+id/checkbox"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:text="Check me?" />
```

You can see how this CheckBox is displayed in Figure.

The following example shows how to check for the state of the button programmatically and change the text label to reflect the change:

```
final CheckBox check_button = (CheckBox)findViewById(R.id.checkbox);
check_button.setOnClickListener(new  View.OnClickListener() {
   public void onClick (View v) {
      TextView tv =  (TextView)findViewById(R.id.checkbox);
      tv.setText(check_button.isChecked()  ?
         "This option is checked" :
         "This option is not checked");
      }
});
```

This is similar to the basic button. A check box automatically shows the check as enabled or disabled. This enables us to deal with behavior in our application rather than worrying about how the button should behave. The layout shows that the text starts out one way but, after the user presses the button, the text changes to one of two different things depending on the checked state. As the code shows, the CheckBox is also a TextView.

A Toggle Button is similar to a check box in behavior but is usually used to show or alter the on or off state of something. Like the CheckBox, it has a state (checked or not). Also like the check box, the act of changing what displays on the button is handled for us. Unlike the CheckBox, it does not show text next to it. Instead, it has two text fields. The first attribute is textOn, which is the text that displays on the button when its checked state is on. The second attribute is textOff, which is the text that displays on the button when its checked state is off. The default text for these is “ON” and “OFF,” respectively.

The following layout code shows a definition for a toggle button that shows “Enabled” or “Disabled” based on the state of the button:

```
<ToggleButton
 android:id=”@+id/toggle_button”
 android:layout_width=”wrap_content”
 android:layout_height=”wrap_content”
 android:text=”Toggle”
 android:textOff=”Disabled”
 android:textOn=”Enabled” />
```

This type of button does not actually display the value for the text attribute, even though it’s a valid attribute to set. Here, the only purpose it serves is to demonstrate that it doesn’t display. You can see what this ToggleButton looks like in Figure.

Using RadioGroups and RadioButtons

You often use radio buttons when a user should be allowed to only select one item from a small group of items. For instance, a question asking for gender can give three options: male, female, and unspecified. Only one of these options should be checked at a time. The RadioButton objects are similar to CheckBox objects. They have a text label next to them, set via the text attribute, and they have a state (checked or unchecked). However, you can group RadioButton objects inside a RadioGroup that handles enforcing their combined states so that only one RadioButton can be checked at a time. If the user selects a RadioButton that is already checked, it does not become unchecked. However, you can provide the user with an action to clear the state of the entire RadioGroup so that none of the buttons are checked.

Here we have an XML layout resource with a RadioGroup containing four RadioButton objects.The RadioButton objects have text labels,"Option 1," and so on.The XML layout resource definition is shown here:

```
<RadioGroup
     android:id="@+id/RadioGroup01"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content">
     <RadioButton
            android:id="@+id/RadioButton01"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Option 1"></RadioButton>
     <RadioButton
            android:id="@+id/RadioButton02"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Option 2"></RadioButton>
     <RadioButton
            android:id="@+id/RadioButton03"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Option 3"></RadioButton>
     <RadioButton
            android:id=”@+id/RadioButton04”
            android:layout_width=”wrap_content”
            android:layout_height=”wrap_content”
            android:text=”Option 4”></RadioButton>
</RadioGroup>
```

You handle actions on these RadioButton objects through the RadioGroup object.The following example shows registering for clicks on the RadioButton objects within the RadioGroup:

```
final RadioGroup group = (RadioGroup)findViewById(R.id.RadioGroup01);
final TextView tv = (TextView)findViewById(R.id.TextView01);
group.setOnCheckedChangeListener(new
     RadioGroup.OnCheckedChangeListener()
     {
            public void onCheckedChanged(
            RadioGroup group, int checkedId) {
            if (checkedId != -1) {
            RadioButton rb = (RadioButton)findViewById(checkedId);
            if (rb != null) {
            tv.setText(“You chose: “ + rb.getText());
            }
            } else {
            tv.setText(“Choose 1”);
     }
}
});
```

As this layout example demonstrates, there is nothing special that you need to do to make the RadioGroup and internal RadioButton objects work properly.The preceding code illustrates how to register to receive a notification whenever the RadioButton selection changes.

The code demonstrates that the notification contains the resource identifier for the specific RadioButton that the user chose, as assigned in the layout file. To do something interesting with this, you need to provide a mapping between this resource identifier (or the text label) to the corresponding functionality in your code. In the example, we query for the button that was selected, get its text, and assign its text to another TextView control that we have on the screen.

As mentioned, the entire RadioGroup can be cleared so that none of the RadioButton objects are selected. The following example demonstrates how to do this in response to a button click outside of the RadioGroup:

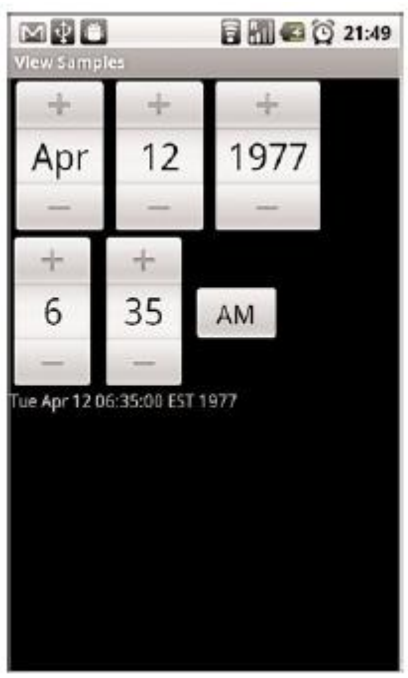
```
final Button clear_choice = (Button) findViewById(R.id.Button01);
clear_choice.setOnClickListener(new View.OnClickListener() {
   public void onClick(View v) {
      RadioGroup group = (RadioGroup)findViewById(R.id.RadioGroup01);
      if (group != null) {
             group.clearCheck();
      }
   }
}
```

The action of calling the clearCheck() method triggers a call to the on Checked Changed Listener()callback method.This is why we have to make sure that the resource identifier we received is valid. Right after a call to the clearCheck() method, it is not a valid identifier but instead is set to the value -1 to indicate that no Radio Button is currently checked.

Getting Dates and Times from Users

The Android SDK provides a couple controls for getting date and time input from the user.The first is the DatePicker control. It can be used to get a month, day, and year from the user.

Date and time controls.



```
Tue Apr 12 06:35:00 EST 1977
```

The basic XML layout resource definition for a DatePicker follows:

```
<DatePicker
android:id="@+id/DatePicker01"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

As you can see from this example, there aren't any attributes specific to the DatePicker control.As with many of the other controls, your code can register to receive a method call when the date changes.You do this by implementing the onDateChanged() method. However, this isn't done the usual way.

```
final DatePicker date = (DatePicker)findViewById(R.id.DatePicker01);
date.init(date.getYear(), date.getMonth(), date.getDayOfMonth(),
  new DatePicker.OnDateChangedListener()  {
     public void onDateChanged(DatePicker view, int year,
        int monthOfYear, int dayOfMonth)  {
          Date dt = new Date(year-1900,
            monthOfYear, dayOfMonth, time.getCurrentHour(),
            time.getCurrentMinute());
            text.setText(dt.toString());
     }
});
```

The preceding code sets the DatePicker.OnDateChangedListener by a call to the DatePicker.init() method. A DatePicker control is initialized with the current date.A Text View is set with the date value that the user entered into the DatePicker control. The value of 1900 is subtracted from the year parameter to make it compatible with the java.util.Date class.

A TimePicker control is similar to the DatePicker control. It also doesn't have any unique attributes. However, to register for a method call when the values change, you call the more traditional method of

```
TimePicker.setOnTimeChangedListener().
time.setOnTimeChangedListener(new TimePicker.OnTimeChangedListener()
{
     public void onTimeChanged(TimePicker view,
       int hourOfDay, int minute) {
       Date dt = new Date(date.getYear()-1900, date.getMonth(),
       date.getDayOfMonth(), hourOfDay, minute);
       text.setText(dt.toString());
     }
});
```

As in the previous example, this code also sets a TextView to a string displaying the time value that the user entered. When you use the DatePicker control and the TimePicker control together, the user can set a full date and time.

Using Indicators to Display Data to Users

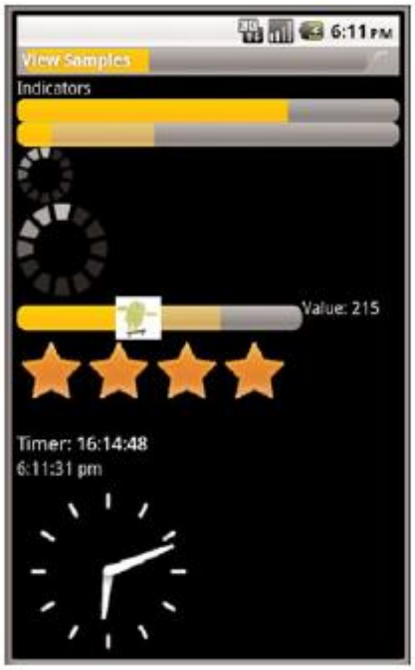
The Android SDK provides a number of controls that can be used to visually show some form of information to the user. These indicator controls include progress bars, clocks, and other similar controls.

Indicating Progress with ProgressBar

Applications commonly perform actions that can take a while. A good practice during this time is to show the user some sort of progress indicator that informs the user that the application is off "doing something."Applications can also show how far a user is through some operation, such as a playing a song or watching a video. The Android SDK provides several types of progress bars.

The standard progress bar is a circular indicator that only animates. It does not show how complete an action is. It can, however, show that something is taking place. This is useful when an action is indeterminate in length. There are three sizes of this type of progress indicator.

Various types of progress and rating indicators.



The second type is a horizontal progress bar that shows the completeness of an action. (For example, you can see how much of a file is downloading.) This horizontal progress bar can also have a secondary progress indicator on it. This can be used, for instance, to show the completion of a downloading media file while that file plays.

This is an XML layout resource definition for a basic indeterminate progress bar:

```
<ProgressBar
android:id="@+id/progress_bar"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

The default style is for a medium-size circular progress indicator; not a “bar” at all. The other two styles for indeterminate progress bar are progressBarStyleLarge and progressBarStyleSmall. This style animates automatically. The next sample shows the layout definition for a horizontal progress indicator:

```
<ProgressBar
     android:id="@+id/progress_bar"
     style="?android:attr/progressBarStyleHorizontal"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:max="100" />
```

We have also set the attribute for max in this sample to 100.This can help mimic a percentage progress bar. That is, setting the progress to 75 shows the indicator at 75 percent complete.

We can set the indicator progress status programmatically as follows:

```
mProgress = (ProgressBar) findViewById(R.id.progress_bar);
mProgress.setProgress(75);
```

You can also put these progress bars in your application's title bar. This can save screen real estate. This can also make it easy to turn on and off an indeterminate progress indicator without changing the look of the screen. Indeterminate progress indicators are commonly used to display progress on pages where items need to be loaded before the page can finish drawing. This is often employed on web browser screens. The following code demonstrates how to place this type of indeterminate progress indicator on your Activity screen:

```
requestWindowFeature(Window.FEATURE_INDETERMINATE_PROGRESS);
requestWindowFeature(Window.FEATURE_PROGRESS);
setContentView(R.layout.indicators);
setProgressBarIndeterminateVisibility(true);
setProgressBarVisibility(true);
setProgress(5000);
```

To use the indeterminate indicator on your Activity objects title bar, you need to request the feature

Window.FEATURE_INDETERMINATE_PROGRESS, as previously shown. This shows a small circular indicator in the right side of the title bar. For a horizontal progress bar style that shows behind the title, you need to enable the Window.FEATURE_PROGRESS. These features must be enabled before your application calls the setContentView() method, as shown in the preceding example.

You need to know about a couple of important default behaviors. First, the indicators are visible by default. Calling the visibility methods shown in the preceding example can set their visibility on or off. Second, the horizontal progress bar defaults to a maximum progress value of 10,000. In the preceding example, we set it to 5,000, which is equivalent to 50 percent. When the value reaches the maximum value, the indicators fade away so that they aren't visible. This happens for both indicators.

Adjusting Progress with SeekBar

You have seen how to display progress to the user. What if, however, you want to give the user some ability to move the indicator, for example, to set the current cursor position in a playing media file or to tweak a volume setting? You accomplish this by using the SeekBar control provided by the Android SDK. It's like the regular horizontal progress bar, but includes a thumb, or selector, that can be dragged by the user. A default thumb selector is provided, but you can use any drawable item as a thumb. In Figure, we replaced the default thumb with a little Android graphic.

Here we have an example of an XML layout resource definition for a simple SeekBar:

```
<SeekBar
android:id="@+id/seekbar1"
android:layout_height="wrap_content"
android:layout_width="240px"
android:max="500" />
```

With this sample SeekBar, the user can drag the thumb to any value between 0 and 500. Although this is shown visually, it might be useful to show the user what exact value the user is selecting. To do this, you can provide an implementation of the onProgressChanged() method, as shown here:

```
SeekBar seek = (SeekBar) findViewById(R.id.seekbar1);
seek.setOnSeekBarChangeListener(
     new SeekBar.OnSeekBarChangeListener()  {
            public void onProgressChanged(
            SeekBar seekBar, int progress,boolean fromTouch) {
                   ((TextView)findViewById(R.id.seek_text))
                     .setText("Value: "+progress);
            seekBar.setSecondaryProgress(
              (progress+seekBar.getMax())/2);
            }
});
```

There are two interesting things to notice in this example. The first is that the fromTouch parameter tells the code if the change came from the user input or if, instead, it came from a programmatic change as demonstrated with the regular ProgressBar controls. The second interesting thing is that the SeekBar still enables you to set a secondary progress value. In this example, we set the secondary indicator to be halfway between the user's selected value and the maximum value of the progress bar. You might use this feature to show the progress of a video and the buffer stream.

Displaying Rating Data with RatingBar

Although the SeekBar is useful for allowing a user to set a value, such as the volume, the RatingBar has a more specific purpose: showing ratings or getting a rating from a user. By default, this progress bar uses the star paradigm with five stars by default. A user can drag across this horizontal to set a rating. A program can set the value, as well. However, the secondary indicator cannot be used because it is used internally by this particular control.

Here’s an example of an XML layout resource definition for a RatingBar with four stars:

```
<RatingBar
     android:id=”@+id/ratebar1”
     android:layout_width=”wrap_content”
     android:layout_height=”wrap_content”
     android:numStars=”4”
     android:stepSize=”0.25” />
```

This layout definition for a RatingBar demonstrates setting both the number of stars and the increment between each rating value. Here, users can choose any rating value between 0 and 4.0, but only in increments of 0.25, the stepSize value. For instance, users can set a value of 2.25.This is visualized to the users, by default, with the stars partially filled. Figure illustrates how the RatingBar behaves.

Although the value is indicated to the user visually, you might still want to show a numeric representation of this value to the user. You can do this by implementing the onRatingChanged() method of the RatingBar.OnRatingBarChangeListener class.

```
RatingBar rate = (RatingBar) findViewById(R.id.ratebar1);
rate.setOnRatingBarChangeListener(new
    RatingBar.OnRatingBarChangeListener() {
     public void onRatingChanged(RatingBar ratingBar,
       float rating, boolean  fromTouch) {
       ((TextView)findViewById(R.id.rating_text))
       .setText("Rating: "+ rating);
    }
});
```

The preceding example shows how to register the listener. When the user selects a rating using the control, a TextView is set to the numeric rating the user entered. One interesting thing to note is that, unlike the SeekBar, the implementation of the onRatingChange() method is called after the change is complete, usually when the user lifts a finger. That is, while the user is dragging across the stars to make a rating, this method isn't called. It is called when the user stops pressing the control.

Showing Time Passage with the Chronometer

Sometimes you want to show time passing instead of incremental progress. In this case, you can use the Chronometer control as a timer. This might be useful if it's the user who is taking time doing some task or in a game where some action needs to be timed. The Chronometer control can be formatted with text, as shown in this XML layout resource definition:

```
<Chronometer
android:id="@+id/Chronometer01"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:format="Timer: %s" />
```

You can use the Chronometer object's format attribute to put text around the time that displays. A Chronometer won't show the passage of time until its start() method is called. To stop it, simply call its stop() method. Finally, you can change the time from which the timer is counting. That is, you can set it to count from a particular time in the past instead of from the time it's started. You call the setBase() method to do this.

In this next example code, the timer is retrieved from the View by its resource identifier. We then check its base value and set it to 0. Finally, we start the timer counting up from there.

```
final Chronometer timer =
   (Chronometer)findViewById(R.id.Chronometer01);
long base = timer.getBase();
Log.d(ViewsMenu.debugTag,  "base = "+ base);
timer.setBase(0);
timer.start();
```

Displaying the Time

Displaying the time in an application is often not necessary because Android devices have a status bar to display the current time. However, there are two clock controls available to display this information: the DigitalClock and AnalogClock controls.

Using the DigitalClock

The DigitalClock control is a compact text display of the current time in standard numeric format based on the users' settings. It is a TextView, so anything you can do with a TextView you can do with this control, except change its text. You can change the color and style of the text, for example.

By default, the DigitalClock control shows the seconds and automatically updates as each second ticks by. Here is an example of an XML layout resource definition for a DigitalClock control:

```
<DigitalClock
android:id="@+id/DigitalClock01"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

Using the AnalogClock

The AnalogClock control is a dial-based clock with a basic clock face with two hands, one for the minute and one for the hour. It updates automatically as each minute passes. The image of the clock scales appropriately with the size of its View. Here is an example of an XML layout resource definition for an AnalogClock control:

```
<AnalogClock
android:id=”@+id/AnalogClock01”
android:layout_width=”wrap_content”
android:layout_height=”wrap_content” />
```

The AnalogClock control’s clock face is simple. However you can set its minute and hour hands. You can also set the clock face to specific drawable resources, if you want to jazz it up. Neither of these clock controls accepts a different time or a static time to display. They can show only the current time in the current time zone of the device, so they are not particularly useful.

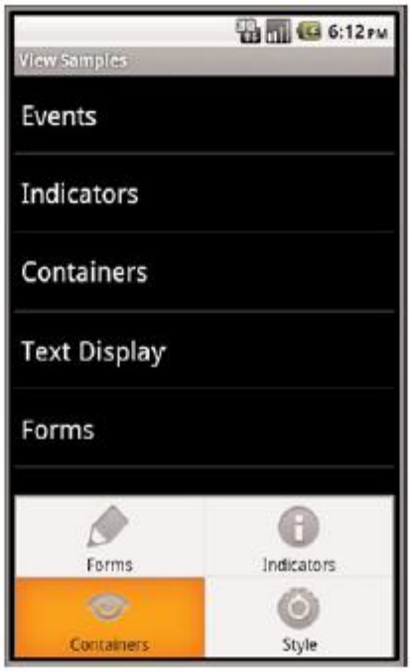
Providing Users with Options and Context Menus

You need to be aware of two special application menus for use within your Android applications: the options menu and the context menu.

Enabling the Options Menu

The Android SDK provides a method for users to bring up a menu by pressing the menu key from within the application. You can use options menus within your application to bring up help, to navigate, to provide additional controls, or to configure options. The OptionsMenu control can contain icons, submenus, and keyboard shortcuts.

An options menu.



For an options menu to show when a user presses the Menu button on their device, you need to override the implementation of onCreateOptionsMenu() in your Activity. Here is a sample implementation that gives the user three menu items to choose from:

```
public boolean onCreateOptionsMenu(
 android.view.Menu menu)
 {
  super.onCreateOptionsMenu(menu);
  menu.add("Forms")
 .setIcon(android.R.drawable.ic_menu_edit)
 .setIntent(new Intent(this,
    FormsActivity.class));
  menu.add("Indicators")
 .setIntent(new Intent(this,
    IndicatorsActivity.class))
 .setIcon(android.R.drawable.ic_menu_info_details);
  menu.add("Containers")
 .setIcon(android.R.drawable.ic_menu_view)
 .setIntent(new Intent(this,  ContainersActivity.class));
  return true;
}
```

For each of the items that are added, we also set a built-in icon resource and assign an Intent to each item. We give the item title with a regular text string, for clarity. You can use a resource identifier, as well. For this example, there is no other handling or code needed. When one of these menu items is selected, the Activity described by the Intent starts.

This type of options menu can be useful for navigating to important parts of an application, such as the help page, from anywhere within your application. Another great use for an options menu is to allow configuration options for a given screen. The user can configure these options in the form of checkable menu items. The initial menu that appears when the user presses the menu button does not support checkable menu items. Instead, you must place these menu items on a SubMenu control, which is a type of Menu that can be configured within a menu. SubMenu objects support checkable items but do not support icons or other SubMenu items. Building on the preceding example, the following is code for programmatically adding a SubMenu control to the previous Menu:

```
SubMenu style_choice = menu.addSubMenu("Style")
 .setIcon(android.R.drawable.ic_menu_preferences);
style_choice.add(style_group, light_id, 1, "Light")
 .setChecked(isLight);
style_choice.add(style_group, dark_id, 2, "Dark")
 .setChecked(!isLight);
style_choice.setGroupCheckable(style_group, true, true);
```

This code would be inserted before the return statement in the implementation of the onCreateOptionsMenu()method. It adds a single menu item with an icon to the previous menu, called "Style."When the "Style" option is clicked, a pop-up menu with the two items of the SubMenu control is displayed. These items are grouped together and the checkable icon, by default, looks like the radio button icon. The checked state is assigned during creation time.

To handle the event when a menu option item is selected, we also implement the onOptionsItemSelected() method, as shown here:

```
public boolean onOptionsItemSelected(MenuItem item) {
   if (item.getItemId() == light_id) {
       item.setChecked(true);
       isLight = true;
       return true;
   } else if (item.getItemId() == dark_id) {
       item.setChecked(true);
       isLight = false;
       return true;
   }
   return super.onOptionsItemSelected(item);
}
```

This method must call the super class's onOptionsItemSelected() method for basic behavior to work. The actual MenuItem object is passed in, and we can use that to retrieve the identifier that we previously assigned to see which one was selected and perform an appropriate action. Here, we switch the values and return. By default, a Menu control goes away when any item is selected, including checkable items. This means it's useful for quick settings but not as useful for extensive settings where the user might want to change more than one item at a time.

As you add more menu items to your options menu, you might notice that a "More" item automatically appears. This happens whenever more than six items are visible. If the user selects this, the full menu appears. The full, expanded menu doesn't show menu icons and although checkable items are possible, you should not use them here. Additionally, the full title of an item doesn't display. The initial menu, also known as the icon menu, shows only a portion of the title for each item. You can assign each item a condensedTitle attribute, which shows instead of a truncated version of the regular title. For example, instead of the title Instant Message, you can set the condensedTitle attribute to "IM."

Enabling the ContextMenu

The ContextMenu is a subtype of Menu that you can configure to display when a long press is performed on a View. As the name implies, the ContextMenu provides for contextual menus to display to the user for performing additional actions on selected items.

ContextMenu objects are slightly more complex than OptionsMenu objects. You need to implement the onCreateContextMenu() method of your Activity for one to display. However, before that is called, you must call the registerForContextMenu() method and pass in the View for which you want to have a context menu. This means each View on your screen can have a different context menu, which is appropriate as the menus are designed to be highly contextual.

Here we have an example of a Chronometer timer, which responds to a long click with a context menu:

```
registerForContextMenu(timer);
```

After the call to the registerForContextMenu() method has been executed, the user can then long click on the View to open the context menu. Each time this happens, your Activity gets a call to the onCreateContextMenu() method, and your code creates the menu each time the user performs the long click.

The following is an example of a context menu for the Chronometer control, as previously used:

```
public void onCreateContextMenu(
  ContextMenu menu, View v, ContextMenuInfo menuInfo) {
    super.onCreateContextMenu(menu, v, menuInfo);
   if (v.getId() == R.id.Chronometer01) {
     getMenuInflater().inflate(R.menu.timer_context, menu);
     menu.setHeaderIcon(android.R.drawable.ic_media_play)
     .setHeaderTitle("Timer controls");
   }
}
```

Recall that any View control can register to trigger a call to the onCreateContextMenu() method when the user performs a long press. That means we have to check which View control it was for which the user tried to get a context menu. Next, we inflate the appropriate menu from a menu resource that we defined with XML. Because we can't define header information in the menu resource file, we set a stock Android SDK resource to it and add a title. Here is the menu resource that is inflated:

```
<menu
   xmlns:android="http://schemas.android.com/apk/res/android">
   <item android:id="@+id/start_timer" android:title="Start" />
   <item android:id="@+id/stop_timer" android:title="Stop" />
   <item android:id="@+id/reset_timer" android:title="Reset" />
</menu>
```

This defines three menu items. If this weren't a context menu, we could have assigned icons. However, context menus do not support icons, submenus, or shortcuts.

Now we need to handle the ContextMenu clicks by implementing the onContext Item Selected() method in our Activity. Here's an example:

```
public boolean onContextItemSelected(MenuItem item) {
  super.onContextItemSelected(item);
  boolean result = false;
  Chronometer timer = (Chronometer)findViewById(R.id.Chronometer01);
  switch (item.getItemId()){
      case R.id.stop_timer:
      timer.stop();
      result = true;
      break;
      case R.id.start_timer:
      timer.start();
      result = true;
      break;
      case R.id.reset_timer:
      timer.setBase(SystemClock.elapsedRealtime());
      result = true;
      break;
    }
    return result;
}
```

Because we have only one context menu in this example, we find the Chronometer view for use in this method. This method is called regardless of which context menu the selected item is on, though, so you should take care to have unique resource identifiers or keep track of which menu is shown. This can be accomplished because the context menu is created each time it's shown.

Handling User Events

You've seen how to do basic event handling in some of the previous control examples. For instance, you know how to handle when a user clicks on a button. There are a number of other events generated by various actions the user might take. This section briefly introduces you to some of these events. First, though, we need to talk about the input states within Android.

Listening for Touch Mode Changes

The Android screen can be in one of two states. The state determines how the focus on View controls is handled. When touch mode is on, typically only objects such as EditText get focus when selected. Other objects, because they can be selected directly by the user tapping on the screen, won't take focus but instead trigger their action, if any. When not in touch mode, however, the user can change focus between even more object types. These include buttons and other views that normally need only a click to trigger their action. In this case, the user uses the arrow keys, trackball, or wheel to navigate between items and select them with the Enter or select keys.

Knowing what mode the screen is in is useful if you want to handle certain events. If, for instance, your application relies on the focus or lack of focus on a particular control, your application might need to know if the phone is in touch mode because the focus behavior is likely different.

Your application can register to find out when the touch mode changes by using the add On Touch Mode Change Listener() method within the android.view.ViewTreeObserver class. Your application needs to implement the View Tree Observer. On Touch Mode Change Listener class to listen for these events. Here is a sample implementation:

```
View all = findViewById(R.id.events_screen);
ViewTreeObserver vto = all.getViewTreeObserver();
vto.addOnTouchModeChangeListener(
    new ViewTreeObserver.OnTouchModeChangeListener() {
       public void onTouchModeChanged(
       boolean isInTouchMode) {
          events.setText("Touch mode: " + isInTouchMode);
       }
});
```

In this example, the top-level View in the layout is retrieved. A ViewTreeObserver listens to a View and all its child View objects. Using the top-level View of the layout means the ViewTreeObserver listens to events within the entire layout. An implementation of the onTouchModeChanged() method provides the ViewTreeObserver with a method to call when the touch mode changes. It merely passes in which mode the View is now in.

In this example, the mode is written to a TextView named events. We use this same TextView in further event handling examples to visually show on the screen which events our application has been told about. The ViewTreeObserver can enable applications to listen to a few other events on an entire screen.

By running this sample code, we can demonstrate the touch mode changing to true immediately when the user taps on the touch screen. Conversely, when the user chooses to use any other input method, the application reports that touch mode is false immediately after the input event, such as a key being pressed or the trackball or scroll wheel moving.

Listening for Events on the Entire Screen

You saw in the last section how your application can watch for changes to the touch mode state for the screen using the ViewTreeObserver class. The ViewTreeObserver also provides three other events that can be watched for on a full screen or an entire View and all of its children. These are

- **PreDraw:** Get notified before the View and its children are drawn
- **GlobalLayout:** Get notified when the layout of the View and its children might change, including visibility changes
- **GlobalFocusChange:** Get notified when the focus within the View and its children changes

Your application might want to perform some actions before the screen is drawn. You can do this by calling the method addOnPreDrawListener() with an implementation of the ViewTreeObserver.OnPreDrawListener class interface.

Similarly, your application can find out when the layout or visibility of a View has changed. This might be useful if your application is dynamically changing the display contents of a view and you want to check to see if a View still fits on the screen. Your application needs to provide an implementation of the View Tree Observer. OnGlobalLayoutListener class interface to the addGlobalLayoutListener() method of the ViewTreeObserver object.

Finally, your application can register to find out when the focus changes between a View control and any of its child View controls. Your application might want to do this to monitor how a user moves about on the screen. When in touch mode, though, there might be fewer focus changes than when the touch mode is not set. In this case, your application needs to provide an implementation of the ViewTreeObserver.OnGlobalFocusChangeListener class interface to the addGlobalFocusChangeListener() method. Here is a sample implementation of this:

```
vto.addOnGlobalFocusChangeListener(new
   ViewTreeObserver.OnGlobalFocusChangeListener()  {
        public void onGlobalFocusChanged(
        View oldFocus, View newFocus) {
           if (oldFocus != null && newFocus != null) {
                 events.setText("Focus \nfrom: " +
                 oldFocus.toString() + " \nto: " +
                 newFocus.toString());
           }
}});
```

This example uses the same ViewTreeObserver, vto, and TextView events as in the previous example. This shows that both the currently focused View and the previously focused View pass to the listener. From here, your application can perform needed actions.

If your application merely wants to check values after the user has modified a particular View, though, you might need to only register to listen for focus changes of that particular View. This is discussed later in this chapter.

Listening for Long Clicks

In a previous section discussing the ContextMenu control, you learned that you can add a context menu to a View that is activated when the user performs a long click on that view. A long click is typically when a user presses on the touch screen and holds his finger there until an action is performed. However, a long press event can also be triggered if the user navigates there with a non-touch method, such as via a keyboard or trackball, and then holds the Enter or Select key for a while. This action is also often called a press-and hold action.

Although the context menu is a great typical use case for the long-click event, you can listen for the long-click event and perform any action you want. However, this is the same event that triggers the context menu. If you've already added a context menu to a View, you might not want to listen for the long-click event as other actions or side effects might confuse the user or even prevent the context menu from showing. As always with good user interface design, try to be consistent for usability sake.

Your application can listen to the long-click event on any View. The following example demonstrates how to listen for a long-click event on a Button control:

```
Button long_press = (Button)findViewById(R.id.long_press);
long_press.setOnLongClickListener(new View.OnLongClickListener() {
public boolean onLongClick(View  v) {
events.setText("Long click: " + v.toString());
return true;
}
});
```

First, the Button object is requested by providing its identifier. Then the setOn LongClick Listener() method is called with our implementation of the View. OnLong Click Listener class interface. The View that the user long-clicked on is passed in to the onLongClick() event handler. Here again we use the same TextView as before to display text saying that a long click occurred.

Listening for Focus Changes

We already discussed focus changes for listening for them on an entire screen. All View objects, though, can also trigger a call to listeners when their particular focus state changes. You do this by providing an implementation of the View.OnFocusChangeListener class to the setOnFocusChangeListener() method.The following is an example of how to listen for focus change events with an EditText control:

```
TextView focus = (TextView)findViewById(R.id.text_focus_change);
focus.setOnFocusChangeListener(new View.OnFocusChangeListener() {
public void onFocusChange(View v, boolean hasFocus) {
     if (hasFocus) {
     if (mSaveText != null) {
            ((TextView)v).setText(mSaveText);
            }
     } else {
  mSaveText = ((TextView)v).getText().toString();
 }
}
```

In this implementation, we also use a private member variable of type String for mSaveText. After retrieving the EditText view as a

TextView, we do one of two things. If the user moves focus away from the control, we store off the text in mSaveText and set the text to empty. If the user changes focus to the control, though, we restore this text. This has the amusing effect of hiding the text the user entered when the control is not active. This can be useful on a form on which a user needs to make multiple, lengthy text entries but you want to provide the user with an easy way to see which one they edit. It is also useful for demonstrating a purpose for the focus listeners on a text entry. Other uses might include validating text a user enters after a user navigates away or prefilling the text entry the first time they navigate to it with something else entered.

Working with Dialogs

An Activity can use dialogs to organize information and react to user-driven events. For example, an activity might display a dialog informing the user of a problem or ask the user to confirm an action such as deleting a data record. Using dialogs for simple tasks helps keep the number of application activities manageable.

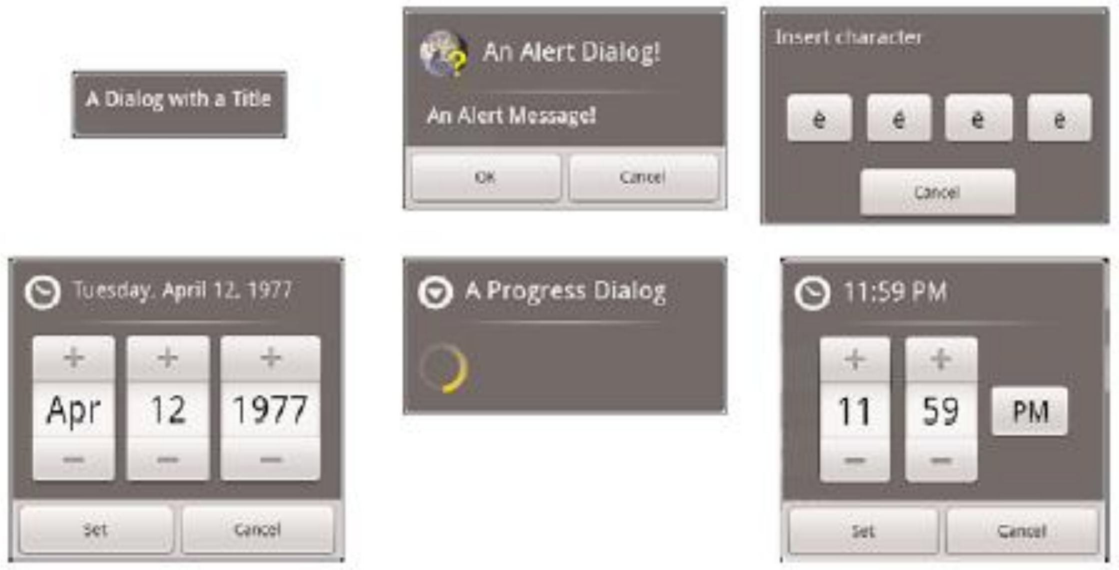
Exploring the Different Types of Dialogs

There are a number of different dialog types available within the Android SDK. Each has a special function that most users should be somewhat familiar with. The dialog types available include

- **Dialog:** The basic class for all Dialog types. A basic Dialog is shown in the top left of Figure.
- **AlertDialog:** A Dialog with one, two, or three Button controls. An AlertDialog is shown in the top center of Figure.

- **CharacterPickerDialog:** A Dialog for choosing an accented character associated with a base character.A CharacterPickerDialog is shown in the top right of Figure.
- **DatePickerDialog:** A Dialog with a DatePicker control. A DatePickerDialog is shown in the bottom left of Figure.
- **ProgressDialog:** A Dialog with a determinate or indeterminate ProgressBar control. An indeterminate ProgressDialog is shown in the bottom center of Figure.
- **TimePickerDialog:** A Dialog with a TimePicker control. A TimePickerDialog is shown in the bottom right of Figure.

The different dialog types available in Android.



If none of the existing Dialog types is adequate, you can also create custom Dialog windows, with your specific layout requirements.

Tracing the Lifecycle of a Dialog

Each Dialog must be defined within the Activity in which it is used. A Dialog may be launched once, or used repeatedly. Understanding how an Activity manages the Dialog lifecycle is important to implementing a Dialog correctly. Let’s look at the key methods that an Activity must use to manage a Dialog:

- The showDialog() method is used to display a Dialog.
- The dismissDialog() method is used to stop showing a Dialog. The Dialog is kept around in the Activity’s Dialog pool. If the Dialog is shown again using showDialog(), the cached version is displayed once more.
- The removeDialog() method is used to remove a Dialog from the Activity objects Dialog pool.The Dialog is no longer kept around for future use. If you call showDialog() again, the Dialog must be re-created.

Adding the Dialog to an Activity involves several steps:

1. Define a unique identifier for the Dialog within the Activity.
2. Implement the onCreateDialog() method of the Activity to return a Dialog of the appropriate type, when supplied the unique identifier.
3. Implement the onPrepareDialog() method of the Activity to initialize the Dialog as appropriate.
4. Launch the Dialog using the showDialog() method with the unique identifier.

Defining a Dialog

A Dialog used by an Activity must be defined in advance. Each Dialog has a special identifier (an integer).When the showDialog() method is called, you pass in this identifier. At this point, the onCreateDialog() method is called and must return a Dialog of the appropriate type.

It is up to the developer to override the onCreateDialog() method of the Activity and return the appropriate Dialog for a given identifier. If an Activity has multiple Dialog windows, the onCreateDialog() method generally contains a switch statement to return the appropriate Dialog based on the incoming parameter—the Dialog identifier.

Initializing a Dialog

Because a Dialog is often kept around by the Activity in its Dialog pool, it might be important to re-initialize a Dialog each time it is shown, instead of just when it is created the first time. For this purpose, you can override the onPrepareDialog() method of the Activity.

Although the onCreateDialog() method may only be called once for initial Dialog creation, the onPrepareDialog() method is called each time the showDialog() method is called, giving the Activity a chance to modify the Dialog before it is shown to the user.

Launching a Dialog

You can display any Dialog defined within an Activity by calling its showDialog() method and passing it a valid Dialog identifier—one that will be recognized by the< onCreateDialog() method.

Dismissing a Dialog

Most types of dialogs have automatic dismissal circumstances. However, if you want to force a Dialog to be dismissed, simply call the dismissDialog() method and pass in the Dialog identifier.

Removing a Dialog from Use

Dismissing a Dialog does not destroy it. If the Dialog is shown again, its cached contents are redisplayed. If you want to force an Activity to remove a Dialog from its pool and not use it again, you can call the removeDialog() method, passing in the valid Dialog identifier.

Working with Custom Dialogs

When the dialog types do not suit your purpose exactly, you can create a custom Dialog. One easy way to create a custom Dialog is to begin with an AlertDialog and use an AlertDialog.Builder class to override its default layout. In order to create a custom Dialog this way, the following steps must be performed:

1. Design a custom layout resource to display in the AlertDialog.
2. Define the custom Dialog identifier in the Activity.
3. Update the Activity's onCreateDialog() method to build and return the appropriate custom AlertDialog.You should use a LayoutInflater to inflate the custom layout resource for the Dialog.
4. Launch the Dialog using the showDialog() method.

Working with Styles

A style is a group of common View attribute values. You can apply the style to individual View controls. Styles can include such settings as the font to draw with or the color of text. The specific attributes depend on the View drawn. In essence, though, each style attribute can change the look and feel of the particular object drawn.

In the previous examples of this chapter, you have seen how XML layout resource files can contain many references to attributes that control the look of TextView objects. You can use a style to define your application's standard TextView attributes once and then reference to the style either in an XML layout file or programmatically from within Java. we see how you can use one style to indicate mandatory form fields and another to indicate optional fields. Styles are typically defined within the resource file res/values/styles.xml. The XML file consists of a resources tag with any number of style tags, which contain an item tag for each attribute and its value that is applied with the style.

The following is an example with two different styles:

```
<?xml version=”1.0” encoding=”utf-8”?>
<resources>
     <style name=”padded_small”>
        <item name=”android:padding”>2px</item>
        <item name=”android:textSize”>8px</item>
     </style>
     <style name=”padded_large”>
        <item name=”android:padding”>4px</item>
        <item name=”android:textSize”>16px</item>
     </style>
</resources>
```

When applied, this style sets the padding to two pixels and the textSize to eight pixels. The following is an example of how it is applied to a TextView from within a layout resource file:

```
<TextView
     style=”@style/padded_small”
     android:layout_width=”fill_parent”
     android:layout_height=”wrap_content”
     android:text=”Small Padded” />
```

Styles support inheritance; therefore, styles can also reference another style as a parent. This way, they pick up the attributes of the parent style. The following is an example of how you might use this:

```
<style name=”red_padded”>
     <item name=”android:textColor”>#F00</item>
     <item name=”android:padding”>3px</item>
</style>
<style name=”padded_normal” parent=”red_padded”>
     <item name=”android:textSize”>12px</item>
</style>
<style name=”padded_italics” parent=”red_padded”>
```

```
     <item name="android:textSize">14px</item>
     <item name="android:textStyle">italic</item>
</style>
```

Here you find two common attributes in a single style and a reference to them from the other two styles that have different attributes. You can reference any style as a parent style; however, you can set only one style as the style attribute of a View. Applying the padded_italics style that is already defined makes the text 14 pixels in size, italic, red, and padded. The following is an example of applying this style:

```
<TextView
     style="@style/padded_italics"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="Italic w/parent color" />
```

As you can see from this example, applying a style with a parent is no different than applying a regular style. In fact, a regular style can be used for applying to Views and used as a parent in a different style.

```
<style name="padded_xlarge">
     <item name="android:padding">10px</item>
     <item name="android:textSize">100px</item>
</style>
<style name="green_glow" parent="padded_xlarge">
     <item name="android:shadowColor">#0F0</item>
     <item name="android:shadowDx">0</item>
     <item name="android:shadowDy">0</item>
     <item name="android:shadowRadius">10</item>
</style>
```

Here the padded_xlarge style is set as the parent for the green_glow style. All six attributes are then applied to any view that this style is set to.

Working with Themes

A theme is a collection of one or more styles (as defined in the resources) but instead of applying the style to a specific control, the style is applied to all View objects within a specified Activity. Applying a theme to a set of View objects all at once simplifies making the user interface look consistent and can be a great way to define color schemes and other common control attribute settings.

An Android theme is essentially a style that is applied to an entire screen. You can specify the theme programmatically by calling the Activity method setTheme() with the style resource identifier. Each attribute of the style is applied to each View within that Activity, as applicable. Styles and attributes defined in the layout files explicitly override those in the theme.

For instance, consider the following style:

```
<style name=”right”>
     <item name=”android:gravity”>right</item>
</style>
```

You can apply this as a theme to the whole screen, which causes any view displayed within that Activity to have its gravity attribute to be right-justified. Applying this theme is as simple as making the method call to the setTheme() method from within the Activity, as shown here:

```
setTheme(R.style.right);
```

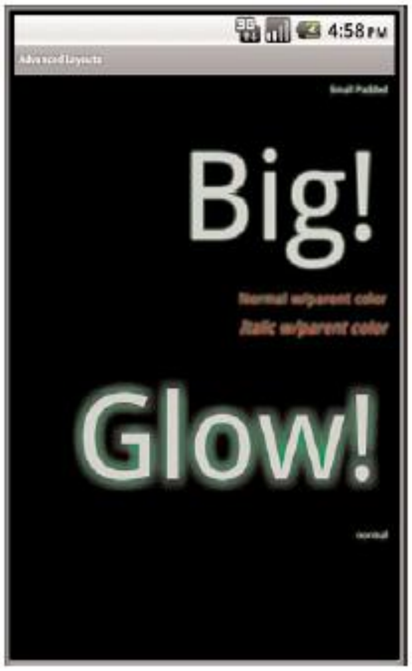
You can also apply themes to specific Activity instances by specifying them as an attribute within the <activity> element in the AndroidManifest.xml file, as follows:

```
<activity android:name=".myactivityname"
   android:label="@string/app_name"
   android:theme="@style/myAppIsStyling">
```

Unlike applying a style in an XML layout file, multiple themes can be applied to a screen. This gives you flexibility in defining style attributes in advance while applying different configurations of the attributes based on what might be displayed on the screen. This is demonstrated in the following code:

```
      setTheme(R.style.right);
      setTheme(R.style.green_glow);
      setContentView(R.layout.style_samples);
```

In this example, both the right style and the green_glow style are applied as a theme to the entire screen. You can see the results of green glow and right-aligned gravity, applied to a variety of TextView controls on a screen, as shown in Figure. Finally, we set the layout to the Activity. You must do this after setting the themes. That is, you must apply all themes before calling the method setContentView() or the inflate() method so that the themes' attributes can take effect.



Packaging styles for glowing text, padding, and alignment into a theme.

A combination of well-designed and thought-out themes and styles can make the look of your application consistent and easy to maintain. Android comes with a number of built-in themes that can be a good starting point. These include such themes as Theme_Black, Theme_Light, and Theme_NoTitleBar_Fullscreen, as defined in the android .R .style class. They are all variations on the system theme, Theme, which builtin apps use.

Designing User Interfaces with Layouts

RKUniversity™

By : Ketan Bhimani

Designing User Interfaces with Layouts

In this chapter, we discuss how to design user interfaces for Android applications. Here we focus on the various layout controls you can use to organize screen elements in different ways. We also cover some of the more complex View objects we call container views. These are View objects that can contain other View objects and controls.

Creating User Interfaces in Android

Application user interfaces can be simple or complex, involving many different screens or only a few. Layouts and user interface controls can be defined as application resources or created programmatically at runtime.

Creating Layouts Using XML Resources

As discussed in previous chapters, Android provides a simple way to create layout files in XML as resources provided in the /res/layout project directory. This is the most common and convenient way to build Android user interfaces and is especially useful for defining static screen elements and control properties that you know in advance, and to set default attributes that you can modify programmatically at runtime.

You can configure almost any ViewGroup or View (or View subclass) attribute using the XML layout resource files. This method greatly simplifies the user interface design process, moving much of the static creation and layout of user interface controls, and basic definition of control attributes, to the XML, instead of littering the code. Developers reserve the ability to alter these layouts programmatically as necessary, but they can set all the defaults in the XML template.

You’ll recognize the following as a simple layout file with a LinearLayout and a single TextView control. This is the default layout file provided with any new Android project in Eclipse, referred to as /res/layout/main.xml:

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

This block of XML shows a basic layout with a single TextView. The first line, which you might recognize from most XML files, is required. Because it’s common across all the files, we do not show it in any other examples.

Next, we have the LinearLayout element. LinearLayout is a ViewGroup that shows each child View either in a single column or in a single row. When applied to a full screen, it merely means that each child View is drawn under the previous View if the orientation is set to vertical or to the right of the previous View if orientation is set to horizontal.

Finally, there is a single child View—in this case, a TextView. A TextView is a control, which is also a View. A TextView draws text on the screen. In this case, it draws the text defined in the "@string/hello" string resource.

Creating only an XML file, though, won't actually draw anything on the screen. A particular layout is usually associated with a particular Activity. In your default Android project, there is only one activity, which sets the main.xml layout by default. To associate the main.xml layout with the activity, use the method call setContentView() with the identifier of the main.xml layout. The ID of the layout matches the XML filename without the extension. In this case, the preceding example came from main.xml, so the identifier of this layout is simply main:

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

The term *layout* is also used to refer to a set of ViewGroup classes such as LinearLayout, FrameLayout, TableLayout, and RelativeLayout. These layout classes are used to organize View controls. We talk more about these classes later in this chapter. Therefore, you could have one or more *layouts* (such as a Linear Layout with two child controls— a TextView and an ImageView) defined within a *layout resource file*, such as /res/layout/myScreen.xml.

Creating Layouts Programmatically

You can create user interface components such as layouts at runtime programmatically, but for organization and maintainability, it's best that you leave this for the odd case rather than the norm. The main reason is because the creation of layouts programmatically is onerous and difficult to maintain, whereas the XML resource method is visual, more organized, and could be done by a separate designer with no Java skills.

The following example shows how to programmatically have an Activity instantiate a LinearLayout view and place two TextView objects within it. No resources what so ever are used; actions are done at runtime instead.

```
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      TextView text1 = new TextView(this);
      text1.setText(“Hi there!”);
      TextView text2 = new TextView(this);
      text2.setText(“I’m second. I need to wrap.”);
      text2.setTextSize((float) 60);
      LinearLayout ll = new LinearLayout(this);
      ll.setOrientation(LinearLayout.VERTICAL);
      ll.addView(text1);
      ll.addView(text2);
      setContentView(ll);
}
```

The onCreate() method is called when the Activity is created. The first thing this method does is some normal Activity housekeeping by calling the constructor for the base class.

Next, two TextView controls are instantiated. The Text property of each TextView is set using the setText() method.All TextView attributes, such as TextSize, are set by making method calls on the TextView object. These actions perform the same function that you have in the past by setting the properties Text and TextSize using the Eclipse layout resource designer, except these properties are set at runtime instead of defined in the layout files compiled into your application package.

To display the TextView objects appropriately, we need to encapsulate them within a container of some sort (a layout). In this case, we use a LinearLayout with the orientation set to VERTICAL so that the second TextView begins beneath the first, each aligned to the left of the screen. The two TextView controls are added to the LinearLayout in the order we want them to display.

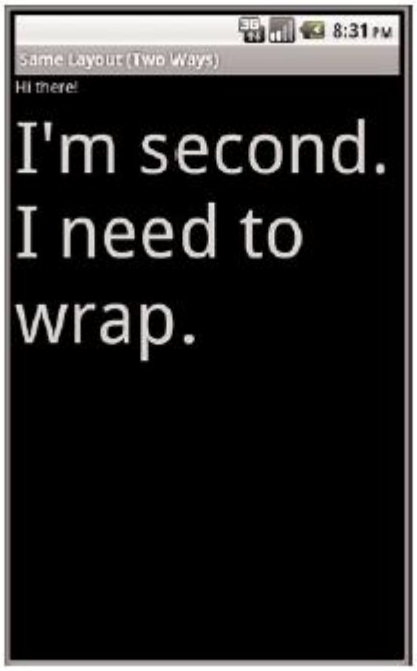
Finally, we call the setContentView() method, part of your Activity class, to draw the LinearLayout and its contents on the screen.

As you can see, the code can rapidly grow in size as you add more View controls and you need more attributes for each View. Here is that same layout, now in an XML layout file:

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=
           “http://schemas.android.com/apk/res/android”
     android:orientation=”vertical”
     android:layout_width=”fill_parent”
     android:layout_height=”fill_parent”>
     <TextView
            android:id=”@+id/TextView1”
            android:layout_width=”fill_parent”
            android:layout_height=”wrap_content”
            android:text=”Hi There!”/>
     <TextView
            android:id=”@+id/TextView2”
            android:layout_width=”fill_parent”
            android:layout_height=”wrap_content”
            android:textSize=”60px”
            android:text=”I’m second. I need to wrap.” />
</LinearLayout>
```

You might notice that this isn’t a literal translation of the code example from the previous section, although the output is identical, as shown in Figure.

Two different methods to create a screen have the same result.



Hi there!

I'm second. I need to wrap.

First, in the XML layout files, layout_width and layout_height are required attributes. Next, you see that each TextView object has a unique id property assigned so that it can be accessed programmatically at runtime. Finally, the textSize property needs to have its units defined. The XML attribute takes a dimension type instead of a float.

The end result differs only slightly from the programmatic method. However, it's far easier to read and maintain. Now you need only one line of code to display this< layout view. Again, if the layout resource is stored in the /res/layout/resource_based_layout.xml file, that is setContentView(R.layout.resource_based_layout);

Organizing Your User Interface

In Chapter "Exploring User Interface Screen Elements," we talk about how the class View is the building block for user interfaces in Android. All user interface controls, such as Button, Spinner, and EditText, derive from the View class.

Now we talk about a special kind of View called a ViewGroup. The classes derived from ViewGroup enable developers to display View objects (including all the user interface controls on the screen in an organized fashion.

Understanding View versus ViewGroup

Like other View objects, ViewGroup controls represent a rectangle of screen space. What makes ViewGroup different from your typical control is that ViewGroup objects contain other View objects. A View that contains other View objects is called a *parent view*. The parent View contains View objects called *child views*, or *children*.

You add child View objects to a ViewGroup programmatically using the method addView().In XML,you add child objects to a ViewGroup by defining the child View control as a child node in the XML (within the parent XML element, as we've seen various times using the LinearLayout ViewGroup).

ViewGroup subclasses are broken down into two categories:

- Layout classes
- View container controls

The Android SDK also provides the Hierarchy Viewer tool to help visualize the layouts you design, as discussed later in this chapter.

Using ViewGroup Subclasses for Layout Design

Many important subclasses of ViewGroup used for screen design end with the word "Layout;" for example, LinearLayout, RelativeLayout, TableLayout, and FrameLayout. You can use each of these layout classes to position groups of View objects (controls) on the screen in different ways. For example, we've been using the

LinearLayout to arrange various TextView and EditText controls on the screen in a single vertical column. We could have used an AbsoluteLayout to specify the exact x/y coordinate locations of each control on the screen instead, but this is not easily portable across many screen resolutions. Users do not generally interact with the Layout objects directly. Instead, they interact with the View objects they contain.

Using ViewGroup Subclasses as View Containers

The second category of ViewGroup subclasses is the indirect subclasses. These special View objects act as View containers like Layout objects do, but they also provide some kind of functionality that enables users to interact with them like normal controls. Unfortunately, these classes are not known by any handy names; instead they are named for the kind of functionality they provide.

Some classes that fall into this category include Gallery, GridView, ImageSwitcher, ScrollView, TabHost, and ListView. It can be helpful to consider these objects as different kinds of View browsers. A ListView displays each View as a list item, and the user can browse between the individual View objects using vertical scrolling capability. A Gallery is a horizontal scrolling list of View objects with a center "current" item; the user can browse the View objects in the Gallery by scrolling left and right. A TabHost is a more complex View container, where each Tab can contain a View, and the user chooses the tab by name to see the View contents.

Using the Hierarchy Viewer Tool

In addition to the Eclipse layout resource designer provided with the Android plug-in, the Android Software Development Kit (SDK) provides a user interface tool called the Hierarchy Viewer. You can find the Hierarchy Viewer in the Android SDK subdirectory called /tools.

The Hierarchy Viewer is a visual tool that enables you to inspect your Android application's View objects and their parent-child relationships. You can drill down on specific View objects and inspect individual View properties at runtime. You can even save screenshots of the current application state on the emulator or the device, although this feature is somewhat unreliable.

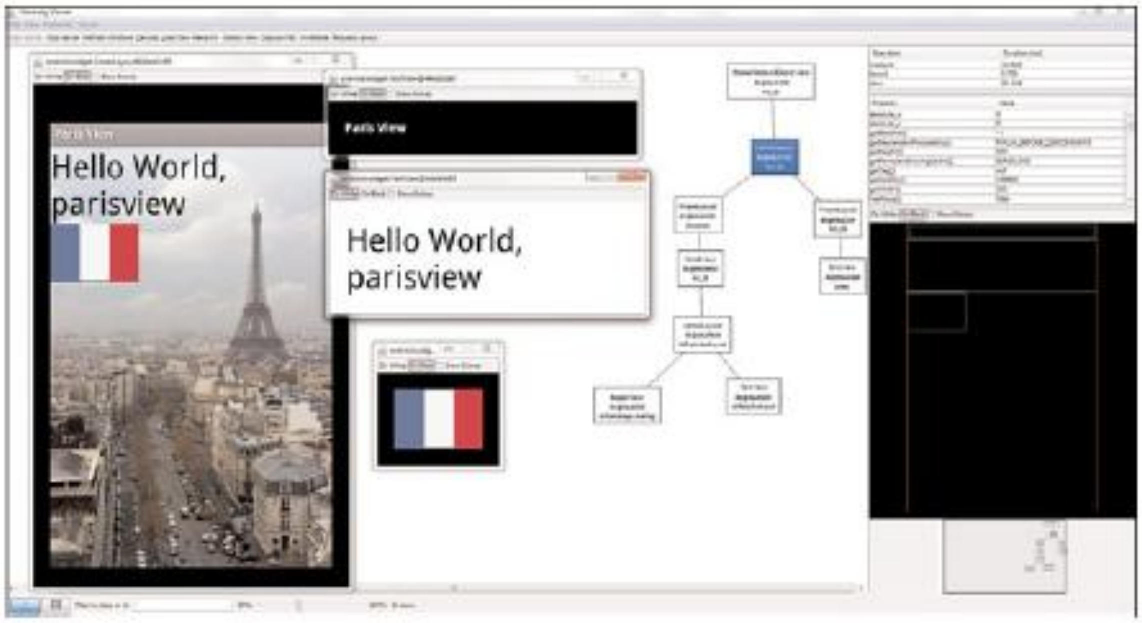
Do the following to launch the HierarchyViewer with your application in the emulator:

1. Launch your Android application in the emulator.
2. Navigate to the Android SDK /tools directory and launch the Hierarchy Viewer.
3. Choose your emulator instance from the Device listing.
4. Select the application you want to view from the windows available. For example, to load an application from this book, choose one such as the ParisView project from Chapter *Managing Application Resources.*
5. Click Load View Hierarchy button on the menu bar.

By default, the Hierarchy Viewer loads the Layout View of your application. This includes the parent-child view relationships shown as a Tree View. In addition, a property pane shows the various properties for each View node in the tree when they are selected. A wire-frame model of the View objects on the screen is shown and a red box highlights the currently selected view, which correlates to the same location on the screen.

Figure shows the Hierarchy Viewer loaded with the ParisView project from Chapter 6, which was a one-screen application with a single LinearLayout with a TextView and an ImageView child control within it, all encapsulated within a ScrollView control(for scrolling ability). The bulk of the application is shown in the right sub-tree, starting with LinearLayout with the identifier ParisViewLayout. The other sub-tree is the Application title bar. A simple double-click on each child node opens that View object individually in its own window.

The ParisView application, shown in the Hierarchy Viewer tool (Layout View).



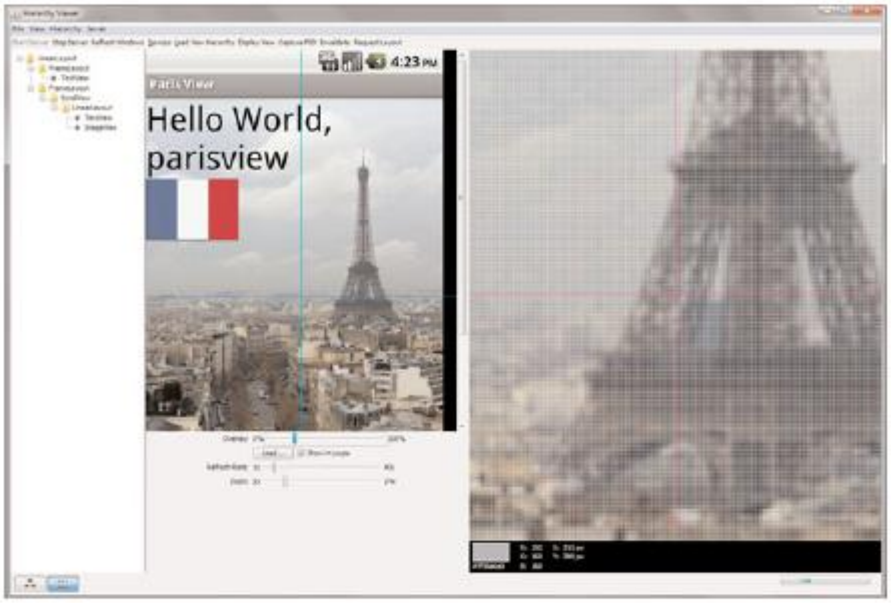
Each View can be separately displayed in its own window by selecting the appropriate View in the tree and choosing the Display View button on the menu bar. In Figure, you can also see that Display View is enabled on each of the child nodes: the ImageView with the flag, the TextView with the text, as well as the LinearLayout parent node (which includes its children), and lastly the application title bar.

You can use the Pixel Perfect view to closely inspect your application using a loupe. You can also load PNG mockup files to overlay your user interface and adjust your application's look.You can access the Pixel Perfect view by clicking the button with the nine pixels on it at the bottom left of the Hierarchy Viewer. Click the button with the three boxes depicting the Layout view to return. The Hierarchy Viewer tool is invaluable for debugging drawing issues related to View controls. If you wonder why something isn't drawing or if a View is even available, try launching the Hierarchy Viewer and checking that problem View objects' properties.

You can use the Hierarchy Viewer tool to interact and debug your application user interface. Specifically, developers can use the Invalidate and Request Layout buttons on the menu bar that correspond to View.invalidate() and View.requestLayout() functions of the UI thread. These functions initiate View objects and draw or redraw them as necessary upon events.

Finally, you can also use the Hierarchy Viewer to deconstruct how other applications (especially sample applications) have handled their layout and displays. This can be helpful if you'd like to re-create a layout similar to another application, especially if it uses stock View types. However, you can also run across View types not provided in the SDK, and you need to implement those custom classes for yourself.

The ParisView application, shown in the Hierarchy Viewer tool (Pixel Perfect View).



Using Built-In Layout Classes

We talked a lot about the LinearLayout layout, but there are several other types of layouts. Each layout has a different purpose and order in which it displays its child View controls on the screen. Layouts are derived from android.view.ViewGroup. The types of layouts built-in to the Android SDK framework include

- FrameLayout
- LinearLayout
- RelativeLayout
- TableLayout

All layouts, regardless of their type, have basic layout attributes. Layout attributes apply to any child View within that layout. You can set layout attributes at runtime programmatically, but ideally you set them in the XML layout files using the following syntax:

```
android:layout_attribute_name="value"
```

There are several layout attributes that all ViewGroup objects share. These include size attributes and margin attributes. You can find basic layout attributes in the ViewGroup.LayoutParams class. The margin attributes enable each child View within a layout to have padding on each side. Find these attributes in the View Group .Margin Layout Params classes. There are also a number of ViewGroup attributes for handling child View drawing bounds and animation settings.

Some of the important attributes shared by all ViewGroup subtypes are shown in Table.

Important ViewGroup Attributes

Attribute Name	Applies To	Description	Value
`android: layout_height`	Parent view Child view	Height of the view. Required attribute for child view controls in layouts.	Specific dimension value, `fill_parent`, or `wrap_content`. The `match_parent` option is available in API Level 8+.
`android: layout_width`	Parent view Child view	Width of the view. Required attribute for child view controls in layouts.	Specific dimension value, `fill_parent`, or `wrap_content`. The `match_parent` option is available in API Level 8+.
`android: layout_margin`	Child view	Extra space on all sides of the view.	Specific dimension value.

Here's an XML layout resource example of a LinearLayout set to the size of the screen, containing one TextView that is set to its full height and the width of the LinearLayout (and therefore the screen):

```
<LinearLayout xmlns:android=
            "http://schemas.android.com/apk/res/android"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent">
     <TextView
            android:id="@+id/TextView01"
            android:layout_height="fill_parent"
            android:layout_width="fill_parent" />
     </LinearLayout>
```

Here is an example of a Button object with some margins set via XML used in a layout resource file:

```
<Button
  android:id="@+id/Button01"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Press Me"
  android:layout_marginRight="20px"
  android:layout_marginTop="60px" />
```

Remember that layout elements can cover any rectangular space on the screen; it doesn't need to be the entire screen. Layouts can be nested within one another. This provides great flexibility when developers need to organize screen elements. It is common to start with a FrameLayout or LinearLayout (as you've seen in many of the previous chapter examples) as the parent layout for the entire screen and then organize individual screen elements inside the parent layout using whichever layout type is most appropriate.

Now let's talk about each of the common layout types individually and how they differ from one another.

Using FrameLayout

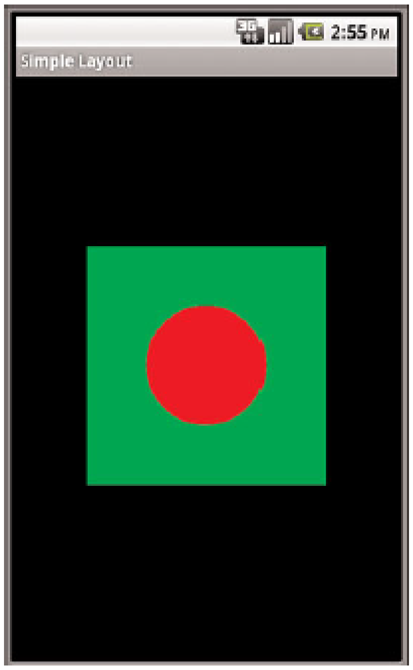
A FrameLayout view is designed to display a stack of child View items. You can add multiple views to this layout, but each View is drawn from the top-left corner of the layout. You can use this to show multiple images within the same region, as shown in Figure, and the layout is sized to the largest child View in the stack.

You can find the layout attributes available for FrameLayout child View objects in android.control.FrameLayout.LayoutParams.Table describes some of the important attributes specific to FrameLayout views.

Important FrameLayout View Attributes

Attribute Name	Applies To	Description	Value
`android:foreground`	Parent view	Drawable to draw over the content.	Drawable resource.
`android:foreground-Gravity`	Parent view	Gravity of foreground drawable.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.
`android:measureAll-Children`	Parent view	Restrict size of layout to all child views or just the child views set to `VISIBLE` (and not those set to `INVISIBLE`).	True or false.
`android:layout_gravity`	Child view	A gravity constant that describes how to place the child `View` within the parent.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.

An example of FrameLayout usage.



Here's an example of an XML layout resource with a FrameLayout and two child View objects, both ImageView objects. The green rectangle is drawn first and the red oval is drawn on top of it. The green rectangle is larger, so it defines the bounds of the FrameLayout:

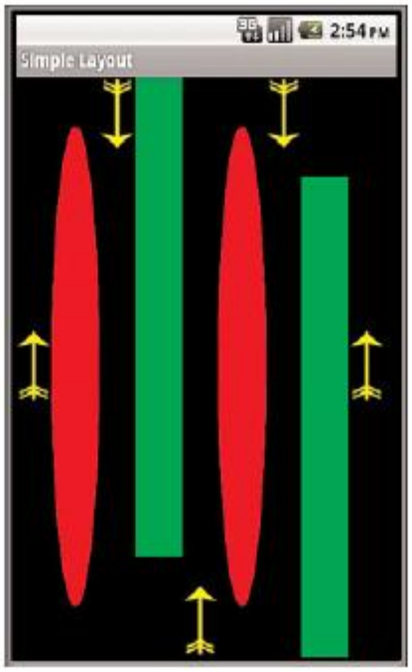
```
<FrameLayout xmlns:android=
     android:id="@+id/FrameLayout01"
      android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_gravity="center">
<ImageView
     android:id="@+id/ImageView01"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:src="@drawable/green_rect"
     android:minHeight="200px"
     android:minWidth="200px" />
<ImageView
     android:id="@+id/ImageView02"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:src="@drawable/red_oval"
     android:minHeight="100px"
     android:minWidth="100px"
     android:layout_gravity="center" />
</FrameLayout>
```

Using **LinearLayout**

A LinearLayout view organizes its child View objects in a single row, shown in Figure, or column, depending on whether its orientation attribute is set to horizontal or vertical. This is a very handy layout method for creating forms.

An example of LinearLayout (horizontal orientation).

You can find the layout attributes available for LinearLayout child View objects in android. control. Linear Layout. Layout Params. Table describes some of the important attributes specific to Linear Layout views.



Important LinearLayout View Attributes

Attribute Name	Applies To	Description	Value
`android: orientation`	Parent view	Layout is a single row (horizontal) or single column (vertical).	Horizontal or vertical.
`android: gravity`	Parent view	Gravity of child views within layout.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_ vertical`, `fill_vertical`, `center_horizontal`, `fill_ horizontal`, `center`, and `fill`.

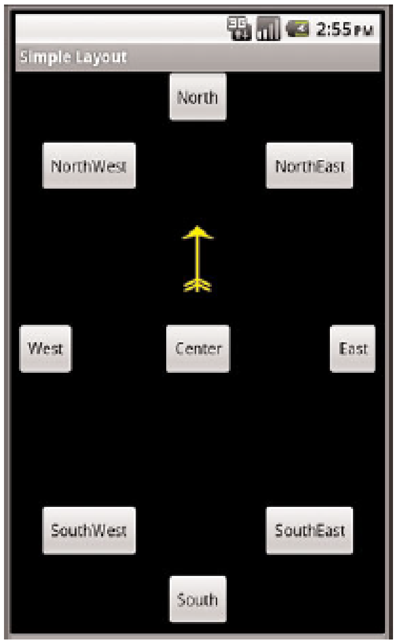
`android:layout_gravity`	Child view	The gravity for a specific child view. Used for positioning of views.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.
`android:layout_weight`	Child view	The weight for a specific child view. Used to provide ratio of screen space used within the parent control.	The sum of values across all child views in a parent view must equal 1. For example, one child control might have a value of `.3` and another have a value of `.7`.

Using RelativeLayout

The RelativeLayout view enables you to specify where the child view controls are in relation to each other. For instance, you can set a child View to be positioned "above" or "below" or "to the left of " or "to the right of " another View, referred to by its unique identifier. You can also align child View objects relative to one another or the parent layout edges. Combining RelativeLayout attributes can simplify creating interesting user interfaces without resorting to multiple layout groups to achieve a desired effect. Figure shows how each of the button controls is relative to each other.

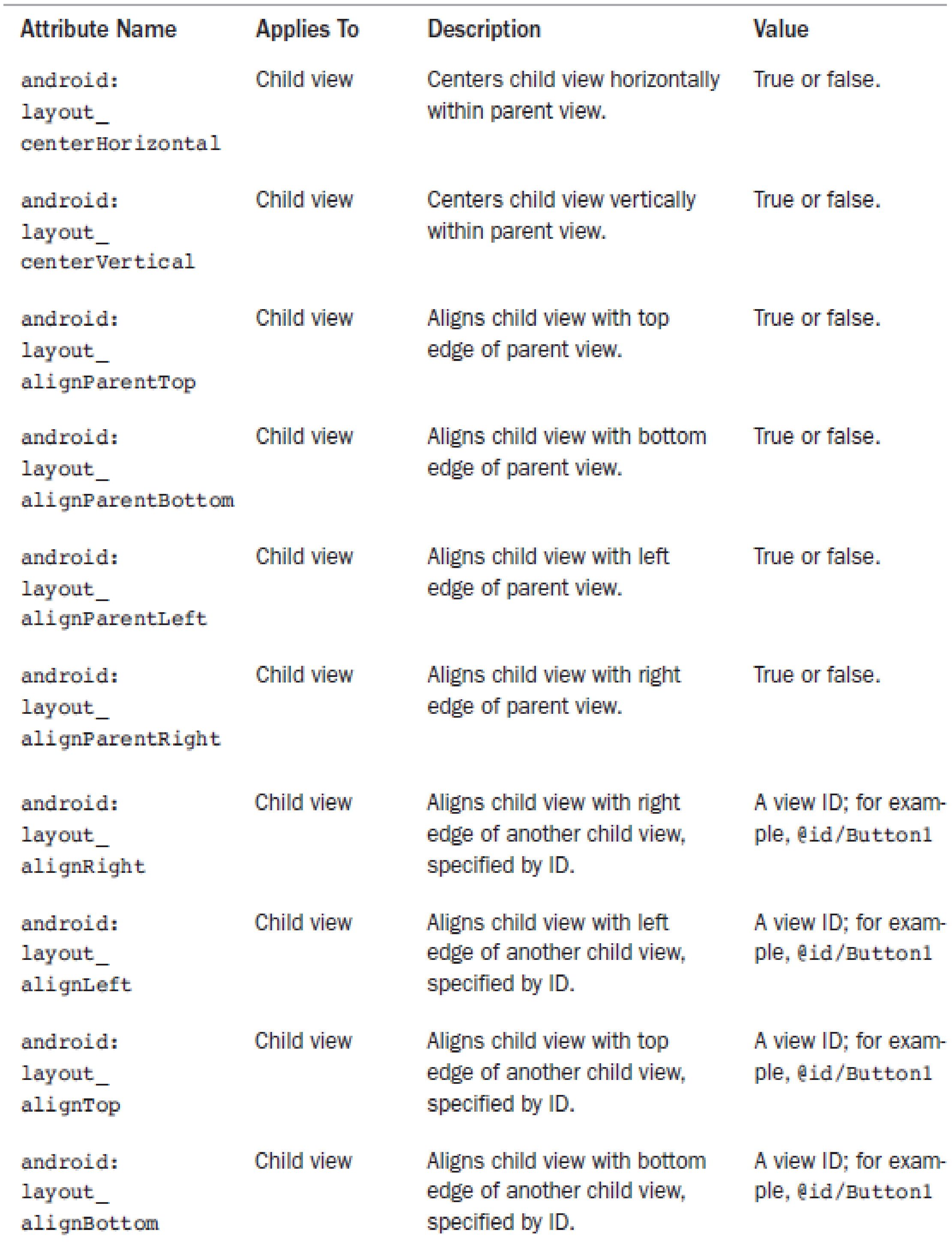
You can find the layout attributes available for RelativeLayout child View objects in android. control. Relative Layout. Layout Params. Table describes some of the important attributes specific to RelativeLayout views.

An example of RelativeLayout usage.



Important RelativeLayout View Attributes

Attribute Name	Applies To	Description	Value
`android: gravity`	Parent view	Gravity of child views within layout.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.
`android: layout_ centerInParent`	Child view	Centers child view horizon-tally and vertically within parent view.	True or false.



Attribute Name	Applies To	Description	Value
`android: layout_ centerHorizontal`	Child view	Centers child view horizontally within parent view.	True or false.
`android: layout_ centerVertical`	Child view	Centers child view vertically within parent view.	True or false.
`android: layout_ alignParentTop`	Child view	Aligns child view with top edge of parent view.	True or false.
`android: layout_ alignParentBottom`	Child view	Aligns child view with bottom edge of parent view.	True or false.
`android: layout_ alignParentLeft`	Child view	Aligns child view with left edge of parent view.	True or false.
`android: layout_ alignParentRight`	Child view	Aligns child view with right edge of parent view.	True or false.
`android: layout_ alignRight`	Child view	Aligns child view with right edge of another child view, specified by ID.	A view ID; for exam-ple, `@id/Button1`
`android: layout_ alignLeft`	Child view	Aligns child view with left edge of another child view, specified by ID.	A view ID; for exam-ple, `@id/Button1`
`android: layout_ alignTop`	Child view	Aligns child view with top edge of another child view, specified by ID.	A view ID; for exam-ple, `@id/Button1`
`android: layout_ alignBottom`	Child view	Aligns child view with bottom edge of another child view, specified by ID.	A view ID; for exam-ple, `@id/Button1`

Attribute Name	Applies To	Description	Value
`android:layout_above`	Child view	Positions bottom edge of child view above another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_below`	Child view	Positions top edge of child view below another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_toLeftOf`	Child view	Positions right edge of child view to the left of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_toRightOf`	Child view	Positions left edge of child view to the right of another child view, specified by ID.	A view ID; for example, `@id/Button1`

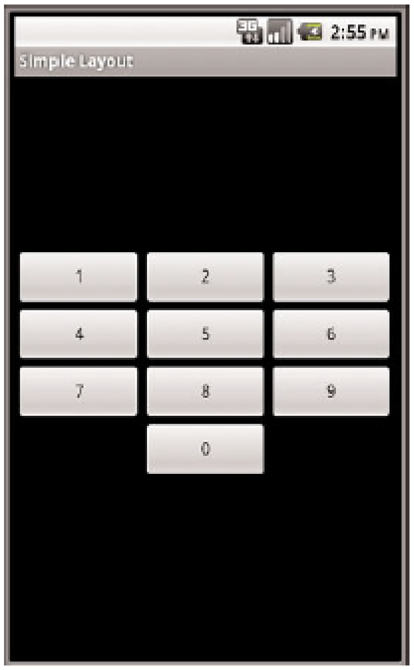
Here’s an example of an XML layout resource with a RelativeLayout and two child View objects, a Button object aligned relative to its parent, and an ImageView aligned and positioned relative to the Button (and the parent):

```
<?xml version=”1.0” encoding=”utf-8”?>
<RelativeLayout xmlns:android=
            “http://schemas.android.com/apk/res/android”
     android:id=”@+id/RelativeLayout01”
     android:layout_height=”fill_parent”
     android:layout_width=”fill_parent”>
     <Button
            android:id=”@+id/ButtonCenter”
            android:text=”Center”
            android:layout_width=”wrap_content”
            android:layout_height=”wrap_content”
            android:layout_centerInParent=”true” />
     <ImageView
            android:id=”@+id/ImageView01”
            android:layout_width=”wrap_content”
            android:layout_height=”wrap_content”
            android:layout_above=”@id/ButtonCenter”
            android:layout_centerHorizontal=”true”
            android:src=”@drawable/arrow” />
</RelativeLayout>
```

Using TableLayout

A TableLayout view organizes children into rows, as shown in Figure. You add individual View objects within each row of the table using a TableRow layout View (which is basically a horizontally oriented LinearLayout) for each row of the table. Each column of the TableRow can contain one View (or layout with child View objects). You place View items added to a TableRow in columns in the order they are added. You can specify the column number (zero-based) to skip columns as necessary (the bottom row shown in Figure demonstrates this); otherwise, the View object is put in the next column to the right. Columns scale to the size of the largest View of that column. You can also include normal View objects instead of TableRow elements, if you want the View to take up an entire row.

An example of TableLayout usage.



You can find the layout attributes available for TableLayout child View objects in android. control. Table Layout. Layout Params. You can find the layout attributes available for TableRow child View objects in android. control. TableRow. Layout Params. Table describes some of the important attributes specific to Table Layout View objects.

Important TableLayout and TableRow View Attributes

Attribute Name	Applies To	Description	Value
`android: collapseColumns`	`TableLayout`	A comma-delimited list of column indices to collapse (0-based)	String or string resource. For example, "`0,1,3,5`"
`android: shrinkColumns`	`TableLayout`	A comma-delimited list of column indices to shrink (0-based)	String or string resource. Use "`*`" for all columns. For example, "`0,1,3,5`"
`andriod: stretchColumns`	`TableLayout`	A comma-delimited list of column indices to stretch (0-based)	String or string resource. Use "`*`" for all columns. For example, "`0,1,3,5`"
`android: layout_column`	`TableRow` child view	Index of column this child view should be displayed in (0-based)	Integer or integer resource. For example, `1`
`android: layout_span`	`TableRow` child view	Number of columns this child view should span across	Integer or integer resource greater than or equal to 1. For example, `3`

Here's an example of an XML layout resource with a TableLayout with two rows (two TableRow child objects). The TableLayout is set to stretch the columns to the size of the screen width. The first TableRow has three columns; each cell has a Button object. The second TableRow puts only one Button view into the second column explicitly:

```
<TableLayout xmlns:android=
            "http://schemas.android.com/apk/res/android"
     android:id="@+id/TableLayout01"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent"
     android:stretchColumns="*">

     <TableRow
            android:id="@+id/TableRow01">
     <Button
            android:id=”@+id/ButtonLeft”
            android:text=”Left Door” />
     <Button
            android:id=”@+id/ButtonMiddle”
            android:text=”Middle Door” />
     <Button
            android:id=”@+id/ButtonRight”
            android:text=”Right Door” />
     </TableRow>
<TableRow
     android:id=”@+id/TableRow02”>
<Button
     android:id=”@+id/ButtonBack”
     android:text=”Go Back”
     android:layout_column=”1” />
</TableRow>
</TableLayout>
```

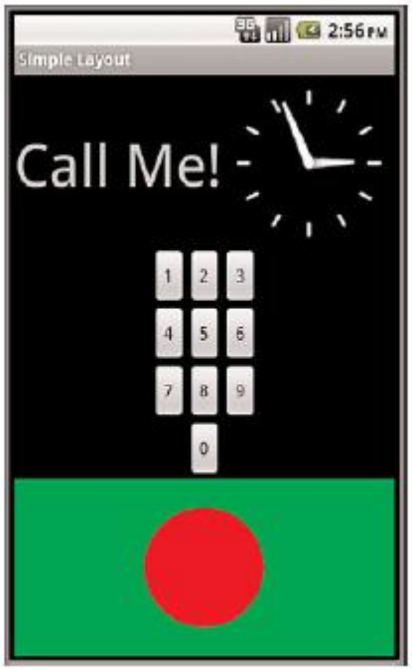
Using Multiple Layouts on a Screen

Combining different layout methods on a single screen can create complex layouts. Remember that because a layout contains View objects and is, itself, a View, it can contain other layouts. Figure demonstrates a combination of layout views used in conjunction to create a more complex and interesting screen.

Using Built-In View Container Classes

Layouts are not the only controls that can contain other View objects. Although layouts are useful for positioning other View objects on the screen, they aren’t interactive. Now let’s talk about the other kind of ViewGroup: the containers. These View objects encapsulate other, simpler View types and give the user some interactive ability to browse the child View objects in a standard fashion. Much like layouts, these controls each have a special, well-defined purpose.

An example of multiple layouts used together.



The types of ViewGroup containers built-in to the Android SDK framework include

- Lists, grids, and galleries
- Switchers with ViewFlipper, ImageSwitcher, and TextSwitcher
- Tabs with TabHost and TabControl
- Scrolling with ScrollView and HorizontalScrollView
- Hiding and showing content with the SlidingDrawer

Using Data-Driven Containers

Some of the View container controls are designed for displaying repetitive View objects in a particular way. Examples of this type of View container control include ListView, GridView, and GalleryView:

- **ListView**: Contains a vertically scrolling, horizontally filled list of View objects, each of which typically contains a row of data; the user can choose an item to perform some action upon.
- **GridView:** Contains a grid of View objects, with a specific number of columns; this container is often used with image icons; the user can choose an item to perform some action upon.
- **GalleryView:** Contains a horizontally scrolling list of View objects, also often used with image icons; the user can select an item to perform some action upon.

These containers are all types of AdapterView controls. An AdapterView control contains a set of child View controls to display data from some data source. An Adapter generates these child

View controls from a data source. As this is an important part of all these container controls, we talk about the Adapter objects first.

In this section, you learn how to bind data to View objects using Adapter objects. In the Android SDK, an Adapter reads data from some data source and provides a View object based on some rules, depending on the type of Adapter used. This View is used to populate the child View objects of a particular AdapterView.

The most common Adapter classes are the CursorAdapter and the ArrayAdapter. The CursorAdapter gathers data from a Cursor, whereas the ArrayAdapter gathers data from an array. A CursorAdapter is a good choice to use when using data from a database. The ArrayAdapter is a good choice to use when there is only a single column of data or when the data comes from a resource array.

There are some common elements to know about Adapter objects. When creating an Adapter, you provide a layout identifier. This layout is the template for filling in each row of data. The template you create contains identifiers for particular controls that the Adapter assigns data to. A simple layout can contain as little as a single TextView control. When making an Adapter, refer to both the layout resource and the identifier of the TextView control. The Android SDK provides some common layout resources for use in your application.

Using the ArrayAdapter

An ArrayAdapter binds each element of the array to a single View object within the layout resource. Here is an example of creating an ArrayAdapter:

```
private String[] items = {“Item 1”, “Item 2”, “Item 3” };
ArrayAdapter adapt =  new ArrayAdapter<String>(this,
R.layout.textview, items);
```

In this example, we have a String array called items. This is the array used by the ArrayAdapter as the source data. We also use a layout resource, which is the View that is repeated for each item in the array. This is defined as follows:

```
<TextView xmlns:android=
     “http://schemas.android.com/apk/res/android”
     android:layout_width=”fill_parent”
     android:layout_height=”wrap_content”
     android:textSize=”20px” />
```

This layout resource contains only a single TextView. However, you can use a more complex layout with the constructors that also take the resource identifier of a TextView within the layout. Each child View within the AdapterView that uses this Adapter gets one TextView instance with one of the strings from the String array.

If you have an array resource defined, you can also directly set the entries attribute for an AdapterView to the resource identifier of the array to automatically provide the Array Adapter.

Using the CursorAdapter

A CursorAdapter binds one or more columns of data to one or more View objects within the layout resource provided. This is best shown with an example. The following< example demonstrates creating a CursorAdapter by querying the Contacts content provider. The CursorAdapter requires the use of a Cursor.

```
Cursor names = managedQuery(
      Contacts.Phones.CONTENT_URI, null, null, null, null);
startManagingCursor(names);
ListAdapter adapter = new SimpleCursorAdapter(
      this, R.layout.two_text, names, new String[] {
             Contacts.Phones.NAME,
             Contacts.Phones.NUMBER
             }, new int[] {
             R.id.scratch_text1,
             R.id.scratch_text2
 });
```

In this example, we present a couple of new concepts. First, you need to know that the Cursor must contain a field named _id. In this case, we know that the Contacts content provider does have this field. This field is used later when you handle the user selecting a particular item.

We make a call to managedQuery() to get the Cursor. Then, we instantiate a Simple Cursor Adapter as a ListAdapter. Our layout, R.layout.two _text, has two TextView objects in it, which are used in the last parameter. SimpleCursorAdapter enables us to match up columns in the database with particular controls in our layout. For each row returned from the query, we get one instance of the layout within our AdapterView.

Binding Data to the AdapterView

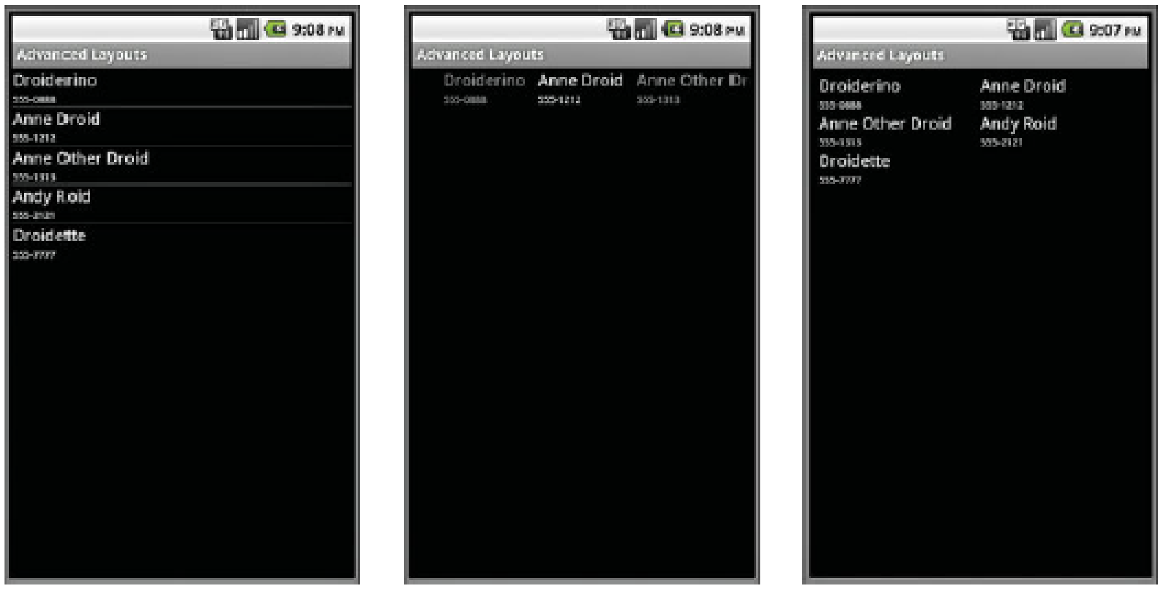
Now that you have an Adapter object, you can apply this to one of the AdapterView controls. Any of them works. Although the Gallery technically takes a Spinner Adapter,the instantiation of

SimpleCursorAdapter also returns a SpinnerAdapter. Here is an example of this with a ListView, continuing on from the previous sample code:

```
((ListView)findViewById(R.id.list)).setAdapter(adapter);
```

The call to the setAdapter() method of the AdapterView, a ListView in this case, should come after your call to setContentView(). This is all that is required to bind data to your AdapterView. Figure shows the same data in a ListView, Gallery, and GridView.

ListView, Gallery, and GridView: same data, same list item, different layout views.



Handling Selection Events

You often use AdapterView controls to present data from which the user should select. All three of the discussed controls—ListView, GridView, and Gallery—enable your application to monitor for click events in the same way. You need to call setOnItemClickListener() on your AdapterView and pass in an implementation of the Adapter

View.OnItemClickListener class. Here is an example implementation of this class:

```
av.setOnItemClickListener(
     new AdapterView.OnItemClickListener() {
     public void onItemClick(
     AdapterView<?> parent, View view,
     int position, long id) {
     Toast.makeText(Scratch.this, "Clicked _id="+id,
     Toast.LENGTH_SHORT).show();
     }
});
```

In the preceding example, av is our AdapterView. The implementation of the onItemClick() method is where all the interesting work happens. The parent parameter is the AdapterView where the item was clicked. This is useful if your screen has more than one AdapterView on it. The View parameter is the specific View within the item that was clicked. The position is the zero-based position within the list of items that the user selects. Finally, the id parameter is the value of the _id column for the particular item that the user selects. This is useful for querying for further information about that particular row of data that the item represents.

Your application can also listen for long-click events on particular items. Additionally, your application can listen for selected items. Although the parameters are the same, your application receives a call as the highlighted item changes. This can be in response to the user scrolling with the arrow keys and not selecting an item for action.

Using the ListActivity

The ListView control is commonly used for full-screen menus or lists of items from which a user selects. As such, you might consider using ListActivity as the base class for such screens. Using the ListActivity can simplify these types of screens.

First, to handle item events, you now need to provide an implementation in your List Activity. For instance, the equivalent of onListItem Click Listener is to implement the onListItemClick() method within your ListActivity.

Second, to assign an Adapter, you need a call to the setListAdapter() method. You do this after the call to the setContentView() method. However, this hints at some of the limitations of using ListActivity.

To use ListActivity, the layout that is set with the setContentView() method must contain a ListView with the identifier set to android:list; this cannot be changed. Second, you can also have a View with an identifier set to android:empty to have a View display when no data is returned from the Adapter. Finally, this works only with ListView controls, so it has limited use. However, when it does work for your application, it can save on some coding.

Organizing Screens with Tabs

The Android SDK has a flexible way to provide a tab interface to the user. Much like ListView and ListActivity, there are two ways to create tabbing on the Android platform. You can either use the TabActivity, which simplifies a screen with tabs, or you can create your own tabbed screens from scratch. Both methods rely on the TabHost control.

Using TabActivity

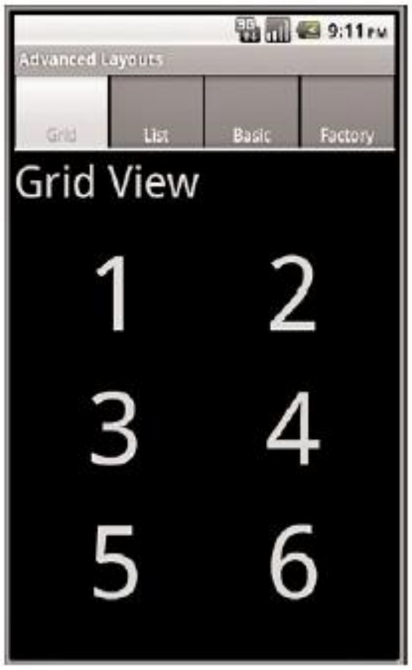
A screen layout with tabs consists of a TabActivity and a TabHost. The TabHost consists of TabSpecs, a nested class of TabHost, which contains the tab information including the tab title and the contents of the tab. The contents of the tab can either be a predefined View, an Activity launched through an Intent object, or the View can be created with a factory provided by an implementation of TabContentFactory.

Tabs aren't as complex as they might sound at first. Each tab is effectively a container for a View. That View can come from any View that is ready to be shown, such as an XML layout file. Alternatively, that View can come from launching an Activity. The following example demonstrates each of these methods using View objects and Activity objects created in the previous examples of this chapter:

```
public class TabLayout extends TabActivity implements
     android.control.TabHost.TabContentFactory {
     protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       TabHost tabHost = getTabHost();
       LayoutInflater.from(this).inflate(
         R.layout.example_layout,tabHost.getTabContentView(), true);
       tabHost.addTab(tabHost.newTabSpec("tab1").setIndicator
         ("Grid").setContent(new Intent(this, GridLayout.class)));
       tabHost.addTab(tabHost.newTabSpec("tab2").setIndicator
         ("List").setContent(new Intent(this, List.class)));
       tabHost.addTab(tabHost.newTabSpec("tab3").setIndicator
         ("Basic").setContent(R.id.two_texts));
       tabHost.addTab(tabHost.newTabSpec("tab4").setIndicator
         ("Factory").setContent(this));
     }
     public View createTabContent(String tag) {
       if (tag.compareTo("tab4") == 0) {
            TextView tv = new TextView(this);
            Date now = new Date();
            tv.setText("I'm from a factory. Created: "
               +  now.toString());
            tv.setTextSize((float) 24);
            return (tv);
     } else {
            return  null;
}}}
```

This example creates a tabbed layout view with four tabs on it, as shown in Figure. The first tab is from the recent GridView sample. The second tab is from the ListView sample before that. The third tab is the basic layout with two TextView objects, fully defined in an XML layout file as previously demonstrated. Finally, the fourth tab is created with a factory.

Four tabs displayed.



The first action is to get the TabHost instance. This is the object that enables us to add Intent objects and View identifiers for drawing the screen. A TabActivity provides a method to retrieve the current TabHost object.

The next action is only loosely related to tab views. The LayoutInflater is used to turn the XML definition of a View into the actual View objects. This would normally happen when calling setContentView(), but we're not doing that. The use of the LayoutInflater is required for referencing the View objects by identifier, as is done for the third tab.

The code finishes up by adding each of the four tabs to the TabHost in the order that they will be presented. This is accomplished by multiple calls to the addTab() method of TabHost. The first two calls are essentially the same. Each one creates a new Intent with the name of an Activity that launches within the tab. These are the same Activity classes used previously for full-screen display. If the Activity isn't designed for full-screen use, this should work seamlessly.

Next, on the third tab, a layout View is added using its identifier. In the preceding call to the LayoutInflater, the layout file also contains an identifier matching the one used here at the top level of a LinearLayout definition. This is the same one used previously to show a basic LinearLayout example. Again, there was no need to change anything in this view for it to display correctly in a tab.

Next, a tab referencing the content as the TabActivity class is added. This is possible because the class itself also implements TabHost.TabContentFactory, which requires implementing the createTabContent() method. The view is created the first time the user selects the tab, so no other information is needed here. The tag that creates this tab must be kept track of, though, as that's how the tabs are identified to the TabHost.

Finally, the method createTabContent() is implemented for use with the fourth tab. The first task here is to check the tag to see if it's the one kept track of for the fourth tab. When that is confirmed, an instance of the TextView object is created and a text string assigned to it, which contains the current time. The size of the text is set to 24 pixels. The time stamp used in this string can be used to demonstrate when the view is created and that it's not re-created by simply changing tabs.

The flexibility of tabs that Android provides is great for adding navigation to an application that has a bunch of views already defined. Few changes, if any, need to be made to existing View and Activity objects for them to work within the context of a TabHost.

Implementing Tabbing Without TabActivity

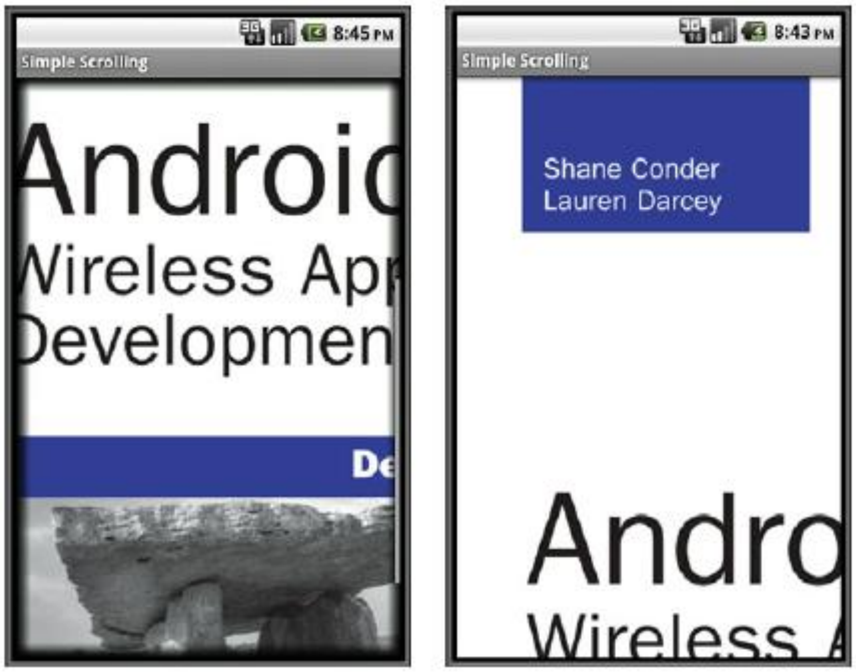
It is possible to design tabbed layouts without using the TabActivity class. However, this requires a bit of knowledge about the underpinnings of the TabHost and TabWidget controls. To define a set of tabs within an XML layout file without using TabActivity, begin with a TabHost (for example, TabHost1). Inside the TabHost, include a vertically oriented LinearLayout that must contain a TabWidget (which must have the id @android:id/tabs) and a FrameLayout (which must have id @android :id /tabcontent). The contents of each tab are then defined within the FrameLayout.

After you've defined the TabHost properly in XML, you must load and initialize it using the TabHost setup() method on your activity's onCreate() method. First, you need to create a TabSpec for each tab, setting the tab indicator using the setIndicator() method and the tab content using the setContent() method. Next, add each tab using the addTab() method of the TabHost. Finally, you should set the default tab of the TabHost, using a method such as setCurrentTabByTag().

Adding Scrolling Support

One of the easiest ways to provide vertical scrolling for a screen is by using the ScrollView (vertical scrolling) and HorizontalScrollView (horizontal scrolling) controls. Either control can be used as a wrapper container, causing all child View controls to have one continuous scrollbar. The ScrollView and HorizontalScrollView controls can have only one child, though, so it's customary to have that child be a layout, such as a LinearLayout, which then contains all the "real" child controls to be scrolled through. Figure shows a screen with and without a ScrollView control.

A screen with (right) and without (left) a ScrollView control.

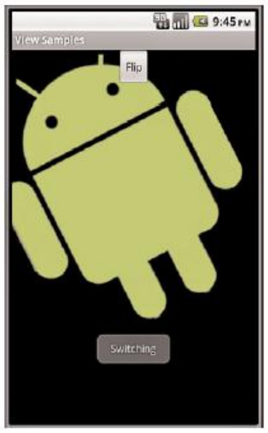


Exploring Other View Containers

Many other user interface controls are available within the Android SDK. Some of these controls are listed here:

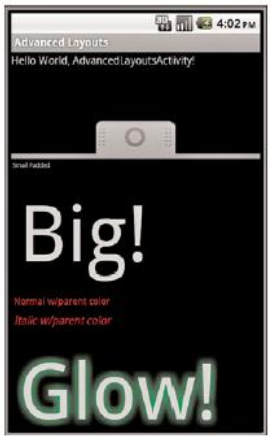
- **Switchers:** A ViewSwitcher control contains only two child View objects and only one of those is shown at a time. It switches between the two, animating as it does so. Primarily, the ImageSwitcher, shown in Figure, and TextSwitcher objects are used. Each one provides a way to set a new child View, either a Drawable resource or a text string, and then animates from what is displayed to the new contents.

ImageSwitcher while in the middle of switching between two Drawable resources.



- **SlidingDrawer:** Another View container is the SlidingDrawer control. This control includes two parts: a handle and a container view. The user drags the handle open and the internal contents are shown; then the user can drag the handle shut and the content disappears. The SlidingDrawer can be used horizontally or vertically and is always used from within a layout representing the larger screen. This makes the SlidingDrawer especially useful for application configurations such as game controls. A user can pull out the drawer, pause the game, change some features, and then close the SlidingDrawer to resume the game. Figure shows how the typical SlidingDrawer looks when pulled open.

SlidingDrawer sliding open to show contents.



Drawing and Working with Animation

Drawing and Working with Animation

This chapter talks about the drawing and animation features built into Android, including creating custom View classes and working with Canvas and Paint to draw shapes and text. We also talk about animating objects on the screen in a variety of ways.

Drawing on the Screen

In Chapter "Exploring User Interface Screen Elements," and Chapter "Designing User Interfaces with Layouts," we talk about layouts and the various View classes available in Android to make screen design simple and efficient. Now we must think at a slightly lower level and talk about drawing objects on the screen. With Android, we can display images such as PNG and JPG graphics, as well as text and primitive shapes to the screen. We can paint these items with various colors, styles, or gradients and modify them using standard image transforms. We can even animate objects to give the illusion of motion.

Working with Canvases and Paints

To draw to the screen, you need a valid Canvas object. Typically we get a valid Canvas object by extending the View class for our own purposes and implementing the

For example, here's a simple View subclass called ViewWithRedDot. We override the onDraw() method to dictate what the View looks like; in this case, it draws a red circle on a black background.

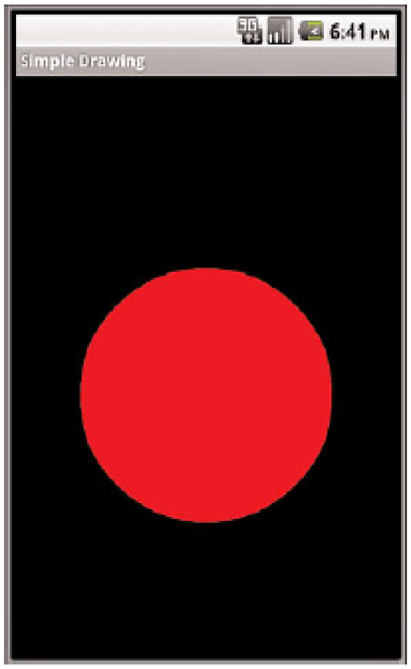
```
private static class ViewWithRedDot extends View {
       public ViewWithRedDot(Context context) {
             super(context);
      }
      @Override
       protected void onDraw(Canvas canvas) {
              canvas.drawColor(Color.BLACK);
             Paint circlePaint = new Paint();
             circlePaintsetColor(Color.RED);
             canvas.drawCircle(canvas.getWidth()/2,
             canvas.getHeight()/2,
             canvas.getWidth()/3, circlePaint);
      }
 }
```

We can then use this View like any other layout. For example, we might override the onCreate() method in our Activity with the following:

```
setContentView(new ViewWithRedDot(this));
```

The resulting screen looks something like Figure. ***The ViewWithRedDot view draws a red circle on a black canvas background.***

Understanding the Canvas



The Canvas (android.graphics.Canvas) object holds the draw calls, in order, for a rectangle of space. There are methods available for drawing images, text, shapes, and support for clipping regions.

The dimensions of the Canvas are bound by the container view. You can retrieve the size of the Canvas using the getHeight() and getWidth() methods.

Understanding the Paint

In Android, the Paint (android.graphics.Paint) object stores far more than a color. The Paint class encapsulates the style and complex color and rendering information, which can be applied to a drawable like a graphic, shape, or piece of text in a given Typeface.

Working with Paint Color

You can set the color of the Paint using the setColor() method. Standard colors are predefined within the android.graphics.Color class. For example, the following code sets the paint color to red:

```
Paint redPaint = new Paint();
redPaint.setColor(Color.RED);
```

Working with Paint Antialiasing

Antialiasing makes many graphics—whether they are shapes or typefaces—look smoother on the screen. This property is set within the Paint of an object.

For example, the following code instantiates a Paint object with antialiasing enabled:

```
    Paint aliasedPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
```

Working with Paint Styles

Paint style controls how an object is filled with color. For example, the following code instantiates a Paint object and sets the Style to STROKE, which signifies that the object should be painted as a line drawing and not filled (the default):

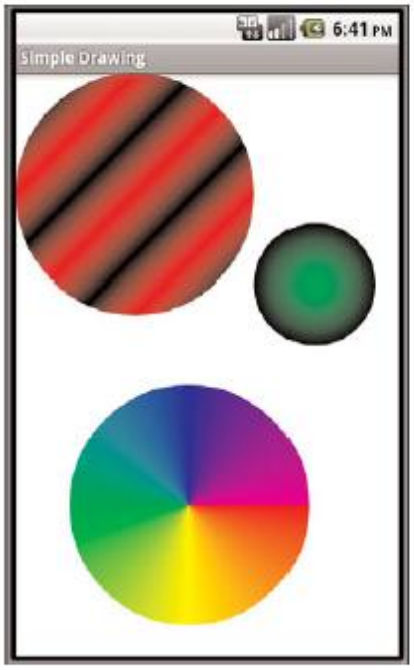
```
Paint linePaint = new Paint();
linePaint.setStyle(Paint.Style.STROKE);
```

Working with Paint Gradients

You can create a gradient of colors using one of the gradient subclasses.The different gradient classes, including LinearGradient, RadialGradient, and SweepGradient, are available under the superclass android.graphics.Shader.

All gradients need at least two colors—a start color and an end color—but might contain any number of colors in an array. The different types of gradients are differentiated by the direction in which the gradient "flows." Gradients can be set to mirror and repeat as necessary.

You can set the Paint gradient using the setShader() method. ***An example of a LinearGradient (top), a RadialGradient (right), and a SweepGradient (bottom).***



Working with Linear Gradients

A linear gradient is one that changes colors along a single straight line. The top-left circle in Figure is a linear gradient between black and red, which is mirrored.

You can achieve this by creating a LinearGradient and setting the Paint method setShader() before drawing on a Canvas, as follows:

```
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.LinearGradient;
import android.graphics.Paint;
import android.graphics.Shader;
...
Paint circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
LinearGradient linGrad = new LinearGradient(0, 0, 25, 25,
Color.RED, Color.BLACK,
Shader.TileMode.MIRROR);
circlePaint.setShader(linGrad);
canvas.drawCircle(100, 100, 100, circlePaint);
```

Working with Radial Gradients

A radial gradient is one that changes colors starting at a single point and radiating outward in a circle. The smaller circle on the right in Figure is a radial gradient between green and black.

You can achieve this by creating a RadialGradient and setting the Paint method setShader() before drawing on a Canvas, as follows:

```
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.RadialGradient;
import android.graphics.Paint;
import android.graphics.Shader;
...
Paint circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
RadialGradient radGrad = new RadialGradient(250,
175, 50, Color.GREEN, Color.BLACK,
Shader.TileMode.MIRROR);
circlePaint.setShader(radGrad);
canvas.drawCircle(250, 175, 50, circlePaint);
```

Working with Sweep Gradients

A sweep gradient is one that changes colors using slices of a pie. This type of gradient is often used for a color *chooser*. The large circle at the bottom of Figure is a sweep gradient between red, yellow, green, blue, and magenta.

You can achieve this by creating a SweepGradient and setting the Paint method setShader() before drawing on a Canvas, as follows:

```
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.SweepGradient;
import android.graphics.Paint;
import android.graphics.Shader;
...
Paint circlePaint = new Paint(Paint.ANTI_ALIAS_FLAG);
SweepGradient sweepGrad = new
SweepGradient(canvas.getWidth()-175,
canvas.getHeight()-175,
new int[] { Color.RED, Color.YELLOW, Color.GREEN,
Color.BLUE, Color.MAGENTA }, null);
circlePaint.setShader(sweepGrad);
canvas.drawCircle(canvas.getWidth()-175,
canvas.getHeight()-175, 100,
circlePaint);
```

Working with Paint Utilities for Drawing Text

The Paint class includes a number of utilities and features for rendering text to the screen in different typefaces and styles. Now is a great time to start drawing some text to the screen.

Working with Text

Android provides several default font typefaces and styles. Applications can also use custom fonts by including font files as application assets and loading them using the AssetManager,much as one would use resources.

Using Default Fonts and Typefaces

By default,Android uses the Sans Serif typeface, but Monospace and Serif typefaces are also available. The following code excerpt draws some antialiased text in the default typeface (Sans Serif) to a Canvas:

```
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.graphics.Typeface;
...
Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
Typeface mType;
mPaint.setTextSize(16);
mPaint.setTypeface(null);
canvas.drawText("Default  Typeface", 20, 20, mPaint);
```

You can instead load a different typeface, such as Monotype:

```
Typeface mType = Typeface.create(Typeface.MONOSPACE,
Typeface.NORMAL);
```

Perhaps you would prefer *italic* text, in which case you can simply set the style of the typeface and the font family:

```
Typeface mType = Typeface.create(Typeface.SERIF,
Typeface.ITALIC);
```

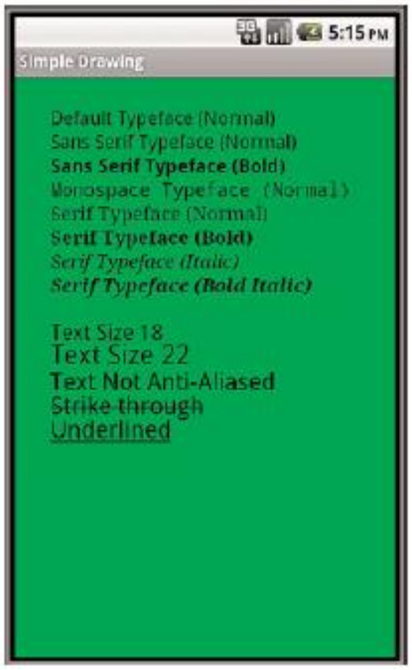
You can set certain properties of a typeface such as antialiasing, underlining, and strikethrough using the setFlags() method of the Paint object:

```
mPaint.setFlags(Paint.UNDERLINE_TEXT_FLAG);
```

Figure shows some of the Typeface families and styles available by default on Android.

Some typefaces and typeface styles available on Android.

Using Custom Typefaces



5:15 PM

Simple Drawing

Default Typeface (Normal)
Sans Serif Typeface (Normal)
Sans Serif Typeface (Bold)
`Monospace Typeface (Normal)`
Serif Typeface (Normal)
Serif Typeface (Bold)
Serif Typeface (Italic)
Serif Typeface (Bold Italic)

Text Size 18
Text Size 22
Text Not Anti-Aliased
~~Strike through~~
Underlined

You can easily use custom typefaces with your application by including the font file as an application asset and loading it on demand. Fonts might be used for a custom look-and feel, for implementing language symbols that are not supported natively, or for custom symbols.

For example, you might want to use a handy chess font to implement a simple, scalable chess game. A chess font includes every symbol needed to implement a chessboard, including the board and the pieces. Hans Bodlaender has kindly provided a free chess font called *Chess Utrecht*. Using the *Chess Utrecht* font, the letter Q draws a black queen on a white square, whereas a q draws a white queen on a white square, and so on.

To use a custom font, such as *Chess Utrecht*, simply download the font from the website and copy the chess1.ttf file from your hard drive to the project directory /assets/fonts/chess1.ttf. Now you can load the Typeface object programmatically much as you would any resource:

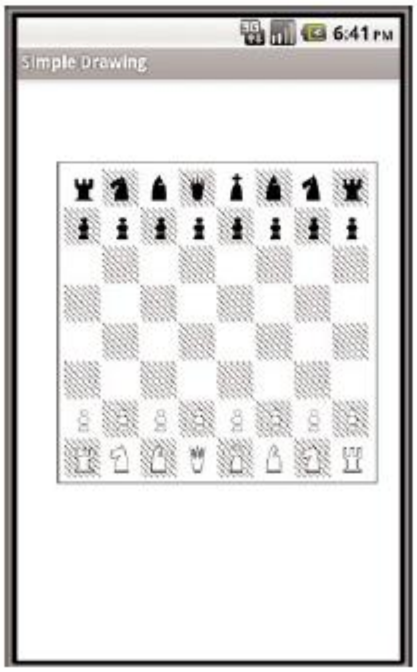
```
import android.graphics.Typeface;
import android.graphics.Color;
import android.graphics.Paint;
...
Paint mPaint = new Paint(Paint.ANTI_ALIAS_FLAG);
Typeface mType =
Typeface.createFromAsset(getContext().getAssets(),
    "fonts/chess1.ttf");
```

You can then use the *Chess Utrecht* typeface to "draw" a chessboard using the appropriate character sequences.

Using the Chess Utrecht font to draw a chessboard

Measuring Text Screen Requirements

You can measure how large text with a given Paint is and how big of a rectangle you need to encompass it using the measureText() and getTextBounds() methods.



Working with Bitmaps

You can find lots of goodies for working with graphics such as bitmaps (including NinePatch) in the android.graphics package. The core class for bitmaps is android.graphics.Bitmap.

Drawing Bitmap Graphics on a Canvas

You can draw bitmaps onto a valid Canvas, such as within the onDraw() method of a View, using one of the drawBitmap()

methods. For example, the following code loads a Bitmap resource and draws it on a canvas:

```
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
...
Bitmap pic = BitmapFactory.decodeResource(getResources(),
R.drawable.bluejay);
canvas.drawBitmap(pic, 0, 0, null);
```

Scaling Bitmap Graphics

Perhaps you want to scale your graphic to a smaller size. In this case, you can use the createScaledBitmap() method, like this:

```
    Bitmap sm = Bitmap.createScaledBitmap(pic, 50, 75, false);
```

You can preserve the aspect ratio of the Bitmap by checking the getWidth() and getHeight() methods and scaling appropriately.

Transforming Bitmaps Using Matrixes

You can use the helpful Matrix class to perform transformations on a Bitmap graphic. Use the Matrix class to perform tasks such as mirroring and rotating graphics, among other actions.

The following code uses the createBitmap() method to generate a new Bitmap that is a mirror of an existing Bitmap called pic:

```
import android.graphics.Bitmap;
import android.graphics.Matrix;
...
Matrix mirrorMatrix = new Matrix();
mirrorMatrix.preScale(-1, 1);
Bitmap mirrorPic = Bitmap.createBitmap(pic, 0, 0,
pic.getWidth(), pic.getHeight(), mirrorMatrix, false);
```

You can perform a 30-degree rotation in addition to mirroring by using this Matrix instead:

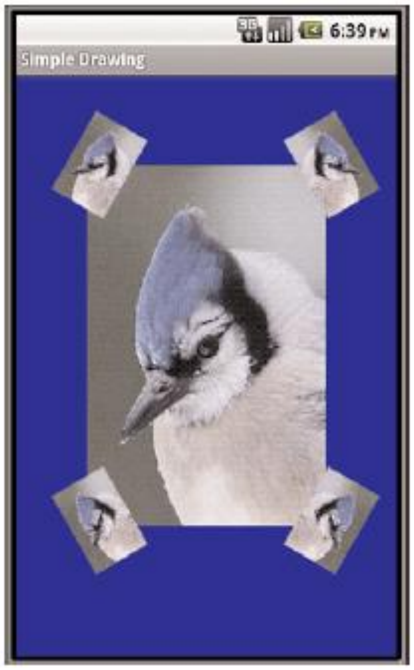
```
Matrix mirrorAndTilt30 = new Matrix();
mirrorAndTilt30.preRotate(30);
mirrorAndTilt30.preScale(-1, 1);
```

You can see the results of different combinations of tilt and mirror Matrix transforms in Figure. When you're no longer using a Bitmap, you can free its memory using the recycle() method:

```
    pic.recycle();
```

A single-source bitmap: scaled, tilted, and mirrored using Android Bitmap classes.

There are a variety of other Bitmap effects and utilities available as part of the Android SDK, but they are numerous and beyond the scope of this book. See the android.graphics package for more details.



Working with Shapes

You can define and draw primitive shapes such as rectangles and ovals using the ShapeDrawable class in conjunction with a variety of specialized Shape classes. You can define Paintable drawables as

XML resource files, but more often, especially with more complex shapes, this is done programmatically.

Defining Shape Drawables as XML Resources

In Chapter"Managing Application Resources," we show you how to define primitive shapes such as rectangles using specially formatted XML files within the /res /drawable/ resource directory. The following resource file called /res/drawable/green_rect.xml describes a simple, green rectangle shape drawable:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
            android:shape="rectangle">
     <solid android:color="#0f0"/>
</shape>
```

You can then load the shape resource and set it as the Drawable as follows:

```
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageResource(R.drawable.green_rect);
```

You should note that many Paint properties can be set via XML as part of the Shape definition. For example, the following Oval shape is defined with a linear gradient (red to white) and stroke style information:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
     android:shape="oval">
<solid android:color="#f00"/>
<gradient android:startColor="#f00" android:endColor="#fff"
     android:angle="180"/>
<stroke android:width="3dp" android:color="#00f"
     android:dashWidth="5dp" android:dashGap="3dp"/>
</shape>
```

Defining Shape Drawables Programmatically

You can also define these ShapeDrawable instances programmatically. The different shapes are available as classes within the android.graphics.drawable.shapes package. For example, you can programmatically define the aforementioned green rectangle as follows:

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RectShape;
...
ShapeDrawable rect = new ShapeDrawable(new RectShape());
rect.getPaint().setColor(Color.GREEN);
```

You can then set the Drawable for the ImageView directly:

```
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rect);
```

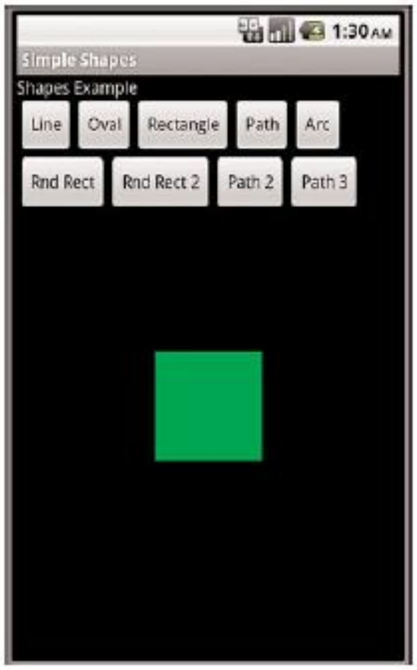
The resulting green rectangle is shown in Figure.

Drawing Different Shapes

Some of the different shapes available within the android.graphics.drawable.shapes package include

- Rectangles (and squares)
- Rectangles with rounded corners
- Ovals (and circles)
- Arcs and lines
- Other shapes defined as paths

A green rectangle.



You can create and use these shapes as Drawable resources directly within ImageView views, or you can find corresponding methods for creating these primitive shapes within a Canvas.

Drawing Rectangles and Squares

Drawing rectangles and squares (rectangles with equal height/width values) is simply a matter of creating a ShapeDrawable from a RectShape object.The RectShape object has no dimensions but is bound by the container object—in this case, the ShapeDrawable. You can set some basic properties of the ShapeDrawable, such as the Paint color and the default size.

For example, here we create a magenta-colored rectangle that is 100-pixels long and 2- pixels wide, which looks like a straight, horizontal line. We then set the shape as the drawable for an ImageView so the shape can be displayed:

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RectShape;
...
ShapeDrawable rect = new ShapeDrawable(new RectShape());
rect.setIntrinsicHeight(2);
rect.setIntrinsicWidth(100);
rect.getPaint().setColor(Color.MAGENTA);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rect);
```

Drawing Rectangles with Rounded Corners

You can create rectangles with rounded corners, which can be nice for making custom buttons. Simply create a ShapeDrawable from a RoundRectShape object. The Round Rect Shape requires an array of eight float values, which signify the radii of the rounded corners. For example, the following creates a simple cyan-colored, rounded-corner rectangle:

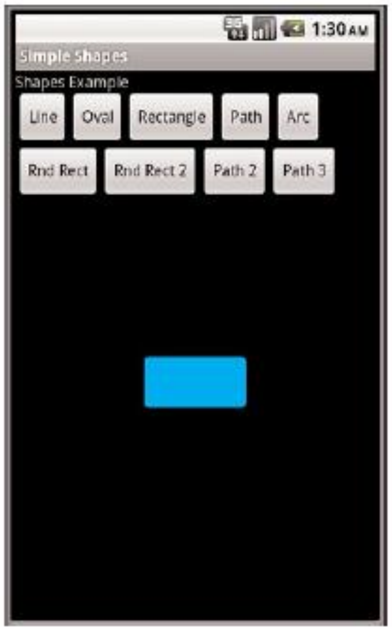
```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RoundRectShape;
...
ShapeDrawable rndrect = new ShapeDrawable(
new RoundRectShape( new float[] { 5, 5, 5, 5, 5,  5, 5, 5 },
null, null));
rndrect.setIntrinsicHeight(50);
rndrect.setIntrinsicWidth(100);
rndrect.getPaint().setColor(Color.CYAN);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rndrect);
```

The resulting round-corner rectangle is shown in Figure.

You can also specify an inner-rounded rectangle within the outer rectangle, if you so choose. The following creates an inner rectangle with rounded edges within the outer white rectangle with rounded edges:

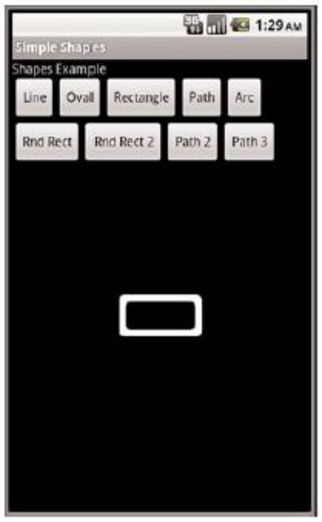
```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.RoundRectShape;
...
float[] outerRadii = new float[]{ 6, 6, 6, 6, 6, 6, 6, 6 };
RectF insetRectangle = new RectF(8, 8, 8, 8);
float[] innerRadii = new float[]{ 6, 6, 6, 6, 6, 6, 6, 6 };
ShapeDrawable rndrect = new ShapeDrawable(new RoundRectShape(
     outerRadii,insetRectangle , innerRadii));
rndrect.setIntrinsicHeight(50);
rndrect.setIntrinsicWidth(100);
rndrect.getPaint().setColor(Color.WHITE);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(rndrect);
```

***A cyan rectangle with rounded corners*.**



The resulting round rectangle with an inset rectangle is shown in Figure.

A white rectangle with rounded corners, with an inset rounded rectangle.

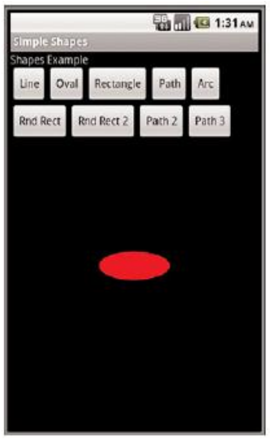


Drawing Ovals and Circles

You can create ovals and circles (which are ovals with equal height/width values) by creating a ShapeDrawable using an OvalShape object. The OvalShape object has no dimensions but is bound by the container object—in this case, the ShapeDrawable. You can set some basic properties of the ShapeDrawable, such as the Paint color and the default size. For example, here we create a red oval that is 40-pixels high and 100-pixels wide, which looks like a Frisbee:

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.OvalShape;
...
ShapeDrawable oval = new ShapeDrawable(new OvalShape());
oval.setIntrinsicHeight(40);
oval.setIntrinsicWidth(100);
oval.getPaint().setColor(Color.RED);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(oval);
```

The resulting red oval is shown in Figure. ***A red oval.***



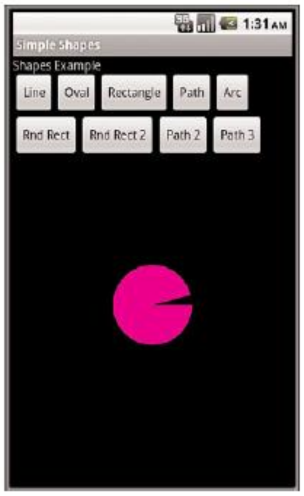
Drawing ArcsM

You can draw arcs, which look like pie charts or Pac-Man, depending on the sweep angle you specify. You can create arcs by creating a ShapeDrawable by using an ArcShape object. The ArcShape object requires two parameters: a startAngle and a sweepAngle.The startAngle begins at 3 o'clock. Positive sweepAngle values sweep clockwise; negative values sweep counterclockwise. You can create a circle by using the values 0 and 360.

The following code creates an arc that looks like a magenta Pac-Man:

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.ArcShape;
...
ShapeDrawable pacMan =
new ShapeDrawable(new ArcShape(0, 345));
pacMan.setIntrinsicHeight(100);
pacMan.setIntrinsicWidth(100);
pacMan.getPaint().setColor(Color.MAGENTA);
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageDrawable(pacMan);
```

The resulting arc is shown in Figure.
A magenta arc of 345 degrees (resembling Pac-Man).



Drawing Paths

You can specify any shape you want by breaking it down into a series of points along a path. The android.graphics.Path class encapsulates a series of lines and curves that make up some larger shape. For example, the following Path defines a rough five-point star shape:

```
import android.graphics.Path;
...
Path p = new Path();
p.moveTo(50, 0);
p.lineTo(25,100);
p.lineTo(100,50);
p.lineTo(0,50);
p.lineTo(75,100);
p.lineTo(50,0);
```

You can then encapsulate this star Path in a PathShape, create a ShapeDrawable, and paint it yellow.

```
import android.graphics.drawable.ShapeDrawable;
import android.graphics.drawable.shapes.PathShape;
...
ShapeDrawable star =
new ShapeDrawable(new PathShape(p, 100, 100));
star.setIntrinsicHeight(100);
star.setIntrinsicWidth(100);
star.getPaint().setColor(Color.YELLOW);
```

By default, this generates a star shape filled with the Paint color yellow *A yellow star*)Or, you can set the Paint style to Stroke for a line drawing of a star.

```
    star.getPaint().setStyle(Paint.Style.STROKE);
```

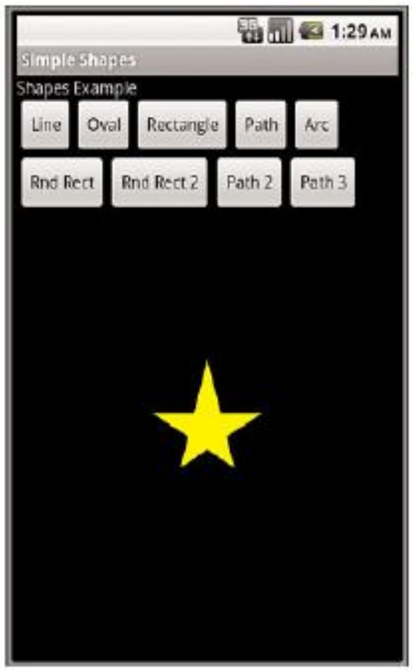
The resulting star would look something like Figure *A yellow star using the stroke style of Paint.*

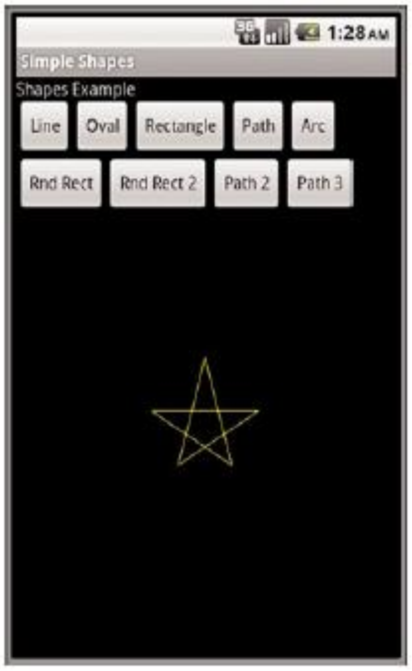
Working with Animation

The Android platform supports three types of graphics animation:

- Animated GIF images
- Frame-by-frame animation
- Tweened animation

A yellow star and yellow star using the stroke style of Paint.





Animated GIFs store the animation frames within the image, and you simply include these GIFs like any other graphic drawable resource. For frame-by-frame animation, the developer must provide all graphics frames of the animation. However, with tweened animation, only a single graphic is needed, upon which transforms can be programmatically applied.

Working with Frame-by-Frame Animation

You can think of frame-by-frame animation as a digital flipbook in which a series of similar images display on the screen in a sequence, each subtly different from the last. When you display these images quickly, they give the illusion of movement. This technique is called frame-by-frame animation and is often used on the Web in the form of animated GIF images.

Frame-by-frame animation is best used for complicated graphics transformations that are not easily implemented programmatically.

For example, we can create the illusion of a genie juggling gifts using a sequence of three images, as shown in Figure. ***Three frames for an animation of a genie juggling.***

GIFT GURU
GIFT GURU
GIFT GURU

In each frame, the genie remains fixed, but the gifts are repositioned slightly. The smoothness of the animation is controlled by providing an adequate number of frames and choosing the appropriate speed on which to swap them.

The following code demonstrates how to load three Bitmap resources (our three genie frames) and create an AnimationDrawable. We then set the AnimationDrawable as the background resource of an ImageView and start the animation:

```
ImageView img = (ImageView)findViewById(R.id.ImageView1);
BitmapDrawable frame1 = (BitmapDrawable)getResources().
     getDrawable(R.drawable.f1);
BitmapDrawable frame2 = (BitmapDrawable)getResources().
      getDrawable(R.drawable.f2);
BitmapDrawable frame3 = (BitmapDrawable)getResources().
     getDrawable(R.drawable.f3);
int reasonableDuration = 250;
AnimationDrawable mAnimation = new AnimationDrawable();
mAnimation.addFrame(frame1, reasonableDuration);
mAnimation.addFrame(frame2, reasonableDuration);
mAnimation.addFrame(frame3, reasonableDuration);
img.setBackgroundDrawable(mAnimation);
```

To name the animation loop continuously,we can call the setOneShot() method:

```
     mAnimation.setOneShot(false);
```

To begin the animation,we call the start() method:

```
     mAnimation.start();
```

We can end our animation at any time using the stop() method:

```
     mAnimation.stop();
```

Although we used an ImageView background in this example, you can use a variety of different View widgets for animations. For example, you can instead use the Image Switcher view and change the displayed Drawable resource using a timer. This sort of operation is best done on a separate thread. The resulting animation might look something like Figure—you just have to imagine it moving.

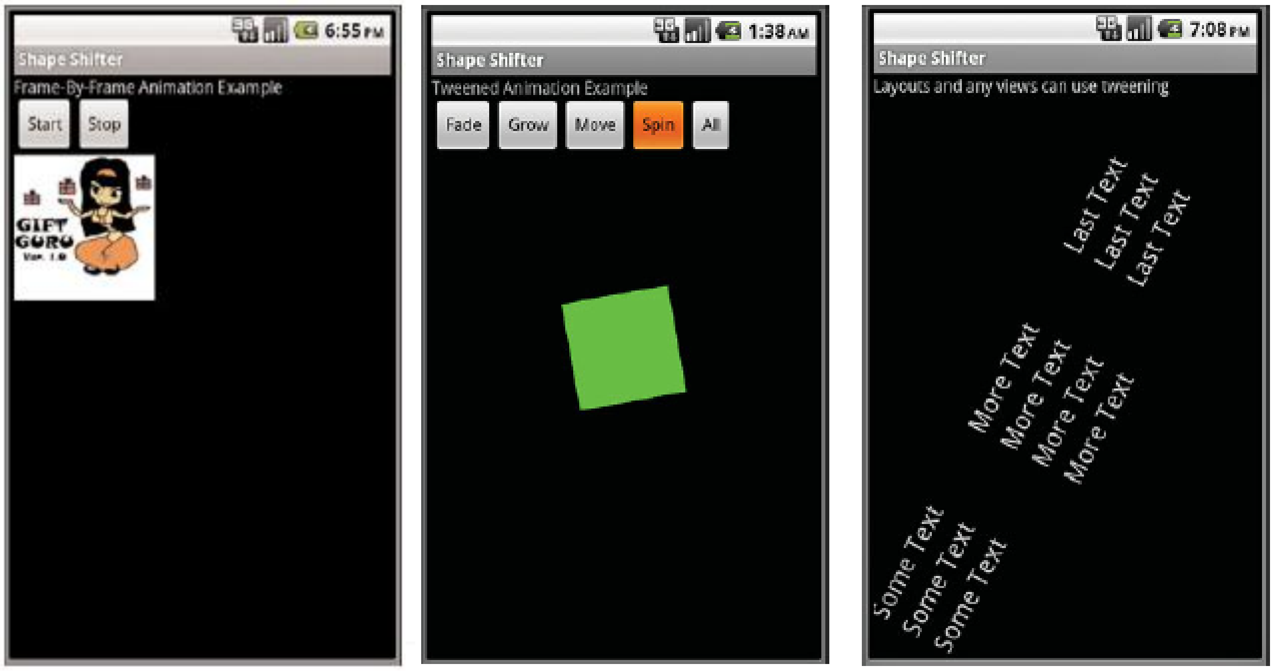
Working with Tweened Animations

With tweened animation, you can provide a single Drawable resource—it is a Bitmap graphic, a ShapeDrawable, a TextView, or any other type of View object—and the intermediate frames of the animation are rendered by the system.Android provides tweening support for several common image transformations, including alpha, rotate, scale, and translate animations. You can apply tweened animation transformations to any View, whether it is an ImageView with a Bitmap or shape Drawable, or a layout such as a TableLayout.

Defining Tweening Transformations

You can define tweening transformations as XML resource files or programmatically. All tweened animations share some common properties, including when to start, how long to animate, and whether to return to the starting state upon completion.

The genie animation in the Android emulator and rotating a green rectangle shape drawable (left) and a TableLayout (right).



Defining Tweened Animations as XML Resources

In Chapter, we showed you how to store animation sequences as specially formatted XML files within the /res/anim/ resource directory. For example, the following resource file called /res/anim/spin.xml describes a simple five-second rotation:

```
<?xml version="1.0" encoding="utf-8" ?>
<set xmlns:android = "http://schemas.android.com/apk/res/android"
     android:shareInterpolator="false">
<rotate
     android:fromDegrees="0"
     android:toDegrees="360"
     android:pivotX="50%"
     android:pivotY="50%"
     android:duration="5000" />
</set>
```

Defining Tweened Animations Programmatically

You can programmatically define these animations. The different types of transformations are available as classes within the android.view.animation package. For example, you can define the aforementioned rotation animation as follows:

```
import android.view.animation.RotateAnimation;
...
RotateAnimation rotate = new RotateAnimation(
0, 360, RotateAnimation.RELATIVE_TO_SELF, 0.5f,
RotateAnimation.RELATIVE_TO_SELF, 0.5f);
rotate.setDuration(5000);
```

Defining Simultaneous and Sequential Tweened Animations

Animation transformations can happen simultaneously or sequentially when you set the startOffset and duration properties, which control when and for how long an animation takes to complete. You can combine animations into the <set> tag (programmatically, using AnimationSet) to share properties.

For example, the following animation resource file /res/anim/grow.xml includes a set of two scale animations: First, we take 2.5 seconds to double in size, and then at 2.5 seconds, we start a second animation to shrink back to our starting size:

```
<?xml version=”1.0” encoding=”utf-8” ?>
<set xmlns:android= http://schemas.android.com/apk/res/android
     android:shareInterpolator=”false”>
<scale
     android:pivotX=”50%”
     android:pivotY=”50%”
     android:fromXScale=”1.0”
     android:fromYScale=”1.0”
     android:toXScale=”2.0”
     android:toYScale=”2.0”
     android:duration=”2500” />
<scale
     android:startOffset=”2500”
     android:duration=”2500”
     android:pivotX=”50%”
     android:pivotY=”50%”
     android:fromXScale=”1.0”
     android:fromYScale=”1.0”
     android:toXScale=”0.5”
     android:toYScale=”0.5” />
</set>
```

Loading Animations

Loading animations is made simple by using the AnimationUtils helper class. The following code loads an animation XML resource file called /res/anim/grow.xml and applies it to an ImageView whose source resource is a green rectangle shape drawable:

```
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
...
ImageView iView = (ImageView)findViewById(R.id.ImageView1);
iView.setImageResource(R.drawable.green_rect);
Animation an =
AnimationUtils.loadAnimation(this, R.anim.grow);
iView.startAnimation(an);
```

We can listen for Animation events, including the animation start, end, and repeat events, by implementing an AnimationListener class, such as the MyListener class shown here:

```
class MyListener implements Animation.AnimationListener {
     public void onAnimationEnd(Animation animation) {
            // Do at end of animation
     }
     public void onAnimationRepeat(Animation animation) {
            // Do each time the animation loops
     }
     public void onAnimationStart(Animation animation)  {
            // Do at start of animation
     }
}
```

You can then register your AnimationListener as follows:

```
an.setAnimationListener(new MyListener());
```

Exploring the Four Different Tweening Transformations

Now let’s look at each of the four types of tweening transformations individually. These types are

- Transparency changes (Alpha)
- Rotations (Rotate)
- Scaling (Scale)
- Movement (Translate)

Working with Alpha Transparency Transformations

Transparency is controlled using Alpha transformations. Alpha transformations can be used to fade objects in and out of view or to layer them on the screen. Alpha values range from 0.0 (fully transparent or invisible) to 1.0 (fully opaque or visible).

Alpha animations involve a starting transparency (fromAlpha) and an ending transparency (toAlpha).

The following XML resource file excerpt defines a transparency-change animation, taking five seconds to fade in from fully transparent to fully opaque:

```
<alpha
android:fromAlpha=”0.0”
android:toAlpha=”1.0”
android:duration=”5000”>
</alpha>
```

Programmatically, you can create this same animation using the AlphaAnimation class within the android.view.animation package.

Working with Rotating Transformations

You can use rotation operations to spin objects clockwise or counterclockwise around a pivot point within the object’s boundaries.

Rotations are defined in terms of degrees. For example, you might want an object to make one complete clockwise rotation. To do this, you set the fromDegrees property to 0 and the toDegrees property to 360. To rotate the object counterclockwise instead, you set the toDegrees property to -360.

By default, the object pivots around the (0,0) coordinate, or the top-left corner of the object. This is great for rotations such as those of a clock's hands, but much of the time, you want to pivot from the center of the object; you can do this easily by setting the pivot point, which can be a fixed coordinate or a percentage. The following XML resource file excerpt defines a rotation animation, taking five seconds to make one full clockwise rotation, pivoting from the center of the object:

```
<rotate
android:fromDegrees="0"
android:toDegrees="360"
android:pivotX="50%"
android:pivotY="50%"
android:duration="5000" />
```

Programmatically, you can create this same animation using the RotateAnimation class within the android.view.animation package.

Working with Scaling Transformations

You can use scaling operations to stretch objects vertically and horizontally. Scaling operations are defined as relative scales. Think of the scale value of 1.0 as 100 percent, or fullsize. To scale to half-size, or 50 percent, set the target scale value of 0.5.

You can scale horizontally and vertically on different scales or on the same scale (to preserve aspect ratio).You need to set four values for proper scaling: starting scale (fromXScale, fromYScale) and target scale (toXScale, toYScale).Again, you can use a pivot point to stretch your object from a specific (x,y) coordinate such as the center or another coordinate.

The following XML resource file excerpt defines a scaling animation, taking five seconds to double an object's size, pivoting from the center of the object:

```
<scale
android:pivotX="50%"
android:pivotY="50%"
android:fromXScale="1.0"
android:fromYScale="1.0"
android:toXScale="2.0"
android:toYScale="2.0"
android:duration="5000" />
```

Programmatically, you can create this same animation using the ScaleAnimation class within the android.view.animation package.

Working with Moving Transformations

You can move objects around using translate operations. Translate operations move an object from one position on the (x,y) coordinate to another coordinate.

To perform a translate operation, you must specify the change, or delta, in the object's coordinates.You can set four values for translations: starting position (fromXDelta, fromYDelta) and relative target location (toXDelta, toYDelta).

The following XML resource file excerpt defines a translate animation, taking 5 seconds to move an object up (negative) by 100 on the y-axis. We also set the fillAfter 230 Chapter 9 *Drawing and Working with Animation* property to be true, so the object doesn’t “jump” back to its starting position when the animation finishes:

```
<translate android:toYDelta=”-100”
android:fillAfter=”true”
android:duration=”2500” />
```

Programmatically, you can create this same animation using the TranslateAnimation class within the android.view.animation package.

Working with Different Interpolators

The animation interpolator determines the rate at which a transformation happens in time. There are a number of different interpolators provided as part of the Android SDK framework. Some of these interpolators include

- AccelerateDecelerateInterpolator: Animation starts slowly, speeds up, and ends slowly
- AccelerateInterpolator: Animation starts slowly and then accelerates
- AnticipateInterpolator: Animation starts backward, and then flings forward
- AnticipateOvershootInterpolator: Animation starts backward, flings forward, overshoots its destination, and then settles at the destination
- BounceInterpolator: Animation “bounces” into place at its destination
- CycleInterpolator: Animation is repeated a certain number of times smoothly transitioning from one cycle to the next
- DecelerateInterpolator: Animation begins quickly, and then decelerates
- LinearInterpolator: Animation speed is constant throughout
- OvershootInterpolator: Animation overshoots its destination, and then settles at the destination

You can specify the interpolator used by an animation programmatically using the setInterpolator() method or in the animation XML resource using the android: interpolator attribute.

Using Android Data and Storage APIs

RKUniversity™

By : Ketan Bhimani

Using Android Data and Storage APIs

Applications are about functionality and data. In this chapter, we explore the various ways you can store, manage, and share application data with Android. Applications can store and manage data in different ways. For example, applications can use a combination of application preferences, the file system, and built-in SQLite database support to store information locally. The methods your application uses depend on your requirements. In this chapter, you learn how to use each of these mechanisms to store, retrieve, and interact with data.

Working with Application Preferences

Many applications need a lightweight data storage mechanism called shared preferences for storing application state, simple user information, configuration options, and other such information.

Android provides a simple preferences system for storing primitive application data at the Activity level and preferences shared across all of an application’s activities. You cannot share preferences outside of the package. Preferences are stored as groups of key/value pairs. The following data types are supported as preference settings:

- Boolean values
- Float values
- Integer values
- Long values
- String values

Preference functionality can be found in the SharedPreferences interface of the android. content package. To add preferences support to your application, you must take the following steps:

1. Retrieve an instance of a SharedPreferences object.
2. Create a SharedPreferences.Editor to modify preference content.
3. Make changes to the preferences using the Editor.
4. Commit your changes.

Creating Private and Shared Preferences

Individual activities can have their own private preferences. These preferences are for the specific Activity only and are not shared with other activities within the application. The activity gets only one group of private preferences.

The following code retrieves the activity's private preferences:

```
import android.content.SharedPreferences;
...
SharedPreferences settingsActivity = getPreferences(MODE_PRIVATE);
```

Creating shared preferences is similar. The only two differences are that we must name our preference set and use a different call to get the preference instance:

```
import android.content.SharedPreferences;
...
SharedPreferences settings =
getSharedPreferences("MyCustomSharedPreferences", 0);
```

You can access shared preferences by name from any activity in the application. There is no limit to the number of different shared preferences you can create. You can have some shared preferences called UserNetworkPreferences and another called AppDisplayPreferences. How you organize shared preferences is up to you, the developer. However, you want to declare your preference name as a variable (in a base class or header) so that you can reuse the name across multiple activities. For example

```
public static final String PREFERENCE_FILENAME = "AppPrefs";
```

Searching and Reading Preferences

Reading preferences is straightforward. Simply retrieve the SharedPreferences instance you want to read. You can check for a preference by name, retrieve strongly typed preferences, and register to listen for changes to the preferences. Table describes some helpful methods in the SharedPreferences interface.

Important android.content.SharedPreferences Methods

Method	Purpose
`SharedPreferences.contains()`	Sees whether a specific preference exists by name
`SharedPreferences.edit()`	Retrieves the editor to change these preferences
`SharedPreferences.getAll()`	Retrieves a map of all preference key/value pairs
`SharedPreferences.getBoolean()`	Retrieves a specific Boolean-type preference by name
`SharedPreferences.getFloat()`	Retrieves a specific Float-type preference by name
`SharedPreferences.getInt()`	Retrieves a specific Integer-type preference by name
`SharedPreferences.getLong()`	Retrieves a specific Long-type preference by name
`SharedPreferences.getString()`	Retrieves a specific String-type preference by name

Adding, Updating, and Deleting Preferences

To change preferences, you need to open the preference Editor, make your changes, and commit them. Table describes some helpful methods in the Shared Preferences. Editor interface.

Important android.content.SharedPreferences. Editor Methods

Method	Purpose
`SharedPreferences.Editor.clear()`	Removes all preferences. This operation happens first, regardless of when it is called within an editing session; then all other changes are made and committed.
`SharedPreferences.Editor.remove()`	Removes a specific preference by name. This operation happens first, regardless of when it is called within an editing session; then all other changes are made and committed.
`SharedPreferences.Editor.putBoolean()`	Sets a specific Boolean-type preference by name.
`SharedPreferences.Editor.putFloat()`	Sets a specific Float-type preference by name.
`SharedPreferences.Editor.putInt()`	Sets a specific Integer-type preference by name.
`SharedPreferences.Editor.putLong()`	Sets a specific Long-type preference by name.
`SharedPreferences.Editor.putString()`	Sets a specific String-type preference by name.
`SharedPreferences.Editor.commit()`	Commits all changes from this editing session.

The following block of code retrieves the activity's private preferences, opens the preference editor, adds a long preference called SomeLong, and saves the change:

```
import android.content.SharedPreferences;
...
SharedPreferences settingsActivity = getPreferences(MODE_PRIVATE);
SharedPreferences.Editor prefEditor = settingsActivity.edit();
prefEditor.putLong("SomeLong", java.lang.Long.MIN_VALUE);
prefEditor.commit();
```

Finding Preferences Data on the Android File System

Internally, application preferences are stored as XML files. You can access the preferences file using DDMS using the File Explorer. You find these files on the Android file system in the following directory:

```
/data/data/<package name>/shared_prefs/<preferences filename>.xml
```

The preferences filename is the Activity's class name for private preferences or the name you give for the shared preferences. Here is an example of the file contents of a simple preference file with a preference in each data type:

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<map>
     <string name="String_Pref">Test String</string>
     <int name="Int_Pref" value="-2147483648" />
     <float name="Float_Pref" value="-Infinity" />
     <long name="Long_Pref" value="9223372036854775807" />
     <boolean name="Boolean_Pref" value="false" />
</map>
```

Understanding the application preferences file format can be helpful for testing purposes. You can use Dalvik Debug Monitor Service (DDMS) to copy the preferences files to and from the device.

Working with Files and Directories

Remember from Chapter "Introducing Android," that each Android application is its own user on the underlying Linux operating system. It has its own private application directory and files. Within the Android SDK, you can also find a variety of standard Java file utility classes (such as java.io) for handling different types of files, such as text files, binary files, and XML files.

In Chapter "Managing Application Resources," you also learned that Android applications can also include static raw and XML files as resources. Although retrieving the file is handled slightly differently when accessing resources, the file can be read like any other file.

Android application files are stored in a standard directory hierarchy on the Android file system. You can browse an application's directory structure using the DDMS File Explorer.

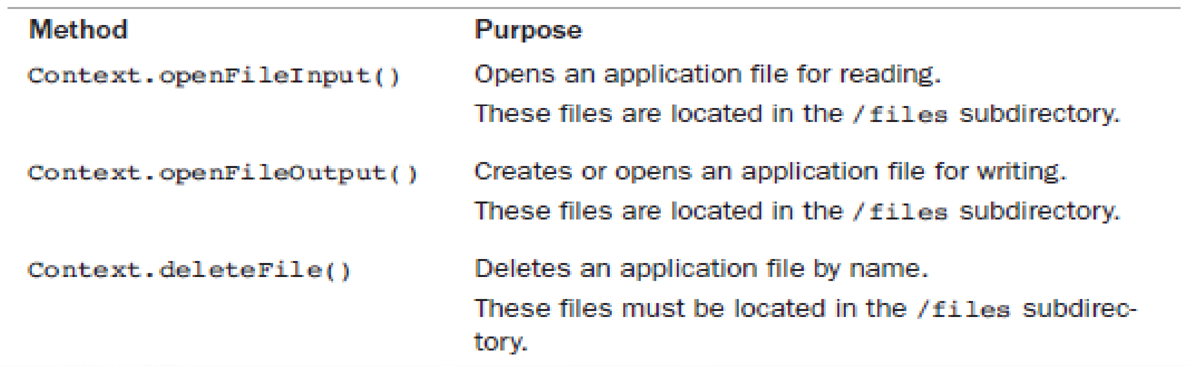
Exploring with the Android Application Directories

Android application data is stored on the Android file system in the following top-level directory:

```
/data/data/<package name>/
```

Several default subdirectories are created for storing databases, preferences, and files as necessary. You can also create other custom directories as needed. File operators all begin by interacting with the application Context object. Table lists some important methods available for application file management. You can use all the standard java.io package utilities to work with FileStream objects and such.

Important android.content.Context File and Directory Management Methods



Method	Purpose
`Context.openFileInput()`	Opens an application file for reading. These files are located in the `/files` subdirectory.
`Context.openFileOutput()`	Creates or opens an application file for writing. These files are located in the `/files` subdirectory.
`Context.deleteFile()`	Deletes an application file by name. These files must be located in the `/files` subdirec-tory.

`Context.fileList()`	Gets a list of all files in the `/files` subdirectory.
`Context.getFilesDir()`	Retrieves the application `/files` subdirectory object.
`Context.getCacheDir()`	Retrieves the application `/cache` subdirectory object.
`Context.getDir()`	Creates or retrieves an application subdirectory by name.

Creating and Writing to Files to the Default Application Directory

Android applications that require only the occasional file rely upon the helpful method called openFileOutput(). Use this method to create files in the default location under the application data directory:

```
/data/data/<package name>/files/
```

For example, the following code snippet creates and opens a file called Filename.txt. We write a single line of text to the file and then close the file:

```
import java.io.FileOutputStream;
...
FileOutputStream fos;
String strFileContents = “Some text to write to the file.”;
fos = openFileOutput(“Filename.txt”, MODE_PRIVATE);
fos.write(strFileContents.getBytes());
fos.close();
```

We can append data to the file by opening it with the mode set to MODE_APPEND:

```
import java.io.FileOutputStream;
...
FileOutputStream fos;
String strFileContents = “More text to write to the file.”;
fos = openFileOutput(“Filename.txt”, MODE_APPEND);
fos.write(strFileContents.getBytes());
fos.close();
```

The file we created has the following path on the Android file system:

```
/data/data/<package name>/files/Filename.txt
```

Reading from Files in the Default Application Directory

Again we have a shortcut for reading files stored in the default /files subdirectory. The following code snippet opens a file called Filename.txt for read operations:

```
import java.io.FileInputStream;
...
String strFileName = "Filename.txt";
FileInputStream fis = openFileInput(strFileName);
```

Reading Raw Files Byte-by-Byte

You handle file-reading and -writing operations using standard Java methods. Check out the subclasses of java.io.InputStream for reading bytes from different types of primitive file types. For example, DataInputStream is useful for reading one line at a time.

Here's a simple example of how to read a text file, line by line, and store it in a StringBuffer:

```
FileInputStream fis = openFileInput(filename);
StringBuffer sBuffer = new StringBuffer();
DataInputStream dataIO = new DataInputStream(fis);
String strLine = null;
while ((strLine = dataIO.readLine()) != null) {
     sBuffer.append(strLine + "\n");
}
dataIO.close();
fis.close();
```

Reading XML Files

The Android SDK includes several utilities for working with XML files, including SAX, an XML Pull Parser, and limited DOM, Level 2 Core support. Table lists the packages helpful for XML parsing on the Android platform.

Important XML Utility Packages

Method	Purpose
`android.sax.*`	Framework to write standard SAX handlers.
`android.util.Xml.*`	XML utilities including the `XMLPullParser`.
`org.xml.sax.*`	Core SAX functionality. Project: www.saxproject.org/.
`javax.xml.*`	SAX and limited DOM, Level 2 Core support.
`org.w3c.dom`	Interfaces for DOM, Level 2 Core.
`org.xmlpull.*`	`XmlPullParser` and `XMLSerializer` interfaces as well as a SAX2 Driver class. Project: www.xmlpull.org/.

Working with Other Directories and Files on the Android File System

Using Context.openFileOutput() and Context.openFileInput() are great if you have a few files and you want them stored in the /files subdirectory, but if you have more sophisticated file-management needs, you need to set up your own directory structure. To do this, you must interact with the Android file system using the standard java.io.File class methods.

The following code gets a File object for the /files application subdirectory and retrieves a list of all filenames in that directory:

```
import java.io.File;
...
File pathForAppFiles = getFilesDir();
String[] fileList = pathForAppFiles.list();
```

Here is a more generic method to create a file on the file system. This method works anywhere on the Android file system you have permission to access, not the /files directory:

```
import java.io.File;
import java.io.FileOutputStream;
...
File fileDir = getFilesDir();
String strNewFileName = "myFile.dat";
String strFileContents = "Some data for our file";
File newFile = new File(fileDir, strNewFileName);
newFile.createNewFile();
FileOutputStream fo =
     new FileOutputStream(newFile.getAbsolutePath());
fo.write(strFileContents.getBytes());
fo.close();
```

You can use File objects to manage files within a desired directory and create sub directories. For example, you might want to store "track" files within "album" directories. Or perhaps you want to create a file in a directory other than the default.

Let's say you want to cache some data to speed up your application's performance and how often it accesses the network. In this instance, you might want to create a cache file. There is also a special application directory for storing cache files. Cache files are stored in the following location on the Android file system:

```
     /data/data/<package name>/cache/
```

The following code gets a File object for the /cache application subdirectory, creates a new file in that specific directory, writes some data to the file, closes the file, and then deletes it:

```
File pathCacheDir = getCacheDir();
String strCacheFileName = "myCacheFile.cache";
String strFileContents = "Some data for our file";
File newCacheFile = new File(pathCacheDir, strCacheFileName);
newCacheFile.createNewFile();
FileOutputStream foCache =
new FileOutputStream(newCacheFile.getAbsolutePath());
foCache.write(strFileContents.getBytes());
foCache.close();
newCacheFile.delete();
```

Storing Structured Data Using SQLite Databases

For occasions when your application requires a more robust data storage mechanism, the Android file system includes support for application-specific relational databases using SQLite. SQLite databases are lightweight and file-based, making them ideally suited for embedded devices.

These databases and the data within them are private to the application. To share application data with other applications, you must expose the data you want to share by making your application a content provider (discussed later in this chapter).

The Android SDK includes a number of useful SQLite database management classes. Many of these classes are found in the android.database.sqlite package. Here you can find utility classes for managing database creation and versioning, database management, and query builder helper classes to help you format proper SQL statements and queries. The package also includes specialized Cursor objects for iterating query results. You can also find all the specialized exceptions associated with SQLite.

Here we focus on creating databases within our Android applications. For that, we use the built-in SQLite support to programmatically create and use a SQLite database to store application information. However, if your application works with a different sort of database, you can also find more generic database classes (within the android.database package) to help you work with data from other providers.

In addition to programmatically creating and using SQLite databases, developers can also interact directly with their application's database using the sqlite3 command-line tool that's accessible through the ADB shell interface. This can be an extremely helpful debugging tool for developers and quality assurance personnel, who might want to manage the database state (and content) for testing purposes.

Creating a SQLite Database

You can create a SQLite database for your Android application in several ways. To illustrate how to create and use a simple SQLite database, let's create an Android project called SimpleDatabase.

Creating a SQLite Database Instance Using the Application Context

The simplest way to create a new SQLiteDatabase instance for your application is to use the openOrCreateDatabase() method of your application Context, like this:

```
import android.database.sqlite.SQLiteDatabase;
...
SQLiteDatabase mDatabase;
mDatabase = openOrCreateDatabase(
“my_sqlite_database.db”,
SQLiteDatabase.CREATE_IF_NECESSARY,
null);
```

Finding the Application’s Database File on the Device File System

Android applications store their databases (SQLite or otherwise) in a special application directory:

```
/data/data/<application package name>/databases/<databasename>
```

So, in this case, the path to the database would be

```
/data/data/com.androidbook.SimpleDatabase/databases/my_sqlite_dat
abase.db
```

You can access your database using the sqlite3 command-line interface using this path.

Configuring the SQLite Database Properties

Now that you have a valid SQLiteDatabase instance, it’s time to configure it. Some important database configuration options include version, locale, and the thread-safe locking feature.

```
import java.util.Locale;
...
mDatabase.setLocale(Locale.getDefault());
mDatabase.setLockingEnabled(true);
mDatabase.setVersion(1);
```

Creating Tables and Other SQLite Schema Objects

Creating tables and other SQLite schema objects is as simple as forming proper SQLite statements and executing them. The following is a valid CREATE TABLE SQL statement. This statement creates a table called tbl_authors. The table has three fields: a unique id number, which auto-increments with each record and acts as our primary key, and firstname and lastname text fields:

```
CREATE TABLE tbl_authors (
id INTEGER PRIMARY KEY AUTOINCREMENT,
firstname TEXT,
lastname TEXT);
```

You can encapsulate this CREATE TABLE SQL statement in a static final String variable (called CREATE_AUTHOR_TABLE) and then execute it on your database using the execSQL() method:

```
    mDatabase.execSQL(CREATE_AUTHOR_TABLE);
```

The execSQL() method works for nonqueries. You can use it to execute any valid SQLite SQL statement. For example, you can use it to create, update, and delete tables, views, triggers, and other common SQL objects. In our application, we add another table called tbl_books. The schema for tbl_books looks like this:

```
CREATE TABLE tbl_books (
id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, dateadded DATE,
authorid INTEGER NOT NULL CONSTRAINT authorid REFERENCES
   tbl_authors(id) ON DELETE
CASCADE);
```

Unfortunately, SQLite does not enforce foreign key constraints. Instead, we must enforce them ourselves using custom SQL triggers. So we create triggers, such as this one that enforces that books have valid authors:

```
private static final String CREATE_TRIGGER_ADD =
  "CREATE TRIGGER fk_insert_book BEFORE INSERT ON tbl_books
  FOR EACH ROW BEGIN SELECT RAISE(ROLLBACK, 'insert on table
  \"tbl_books\" violates foreign key constraint \"fk_authorid\"')
  WHERE (SELECT id FROM tbl_authors WHERE id = NEW.authorid) IS NULL;
  END;";
```

We can then create the trigger simply by executing the CREATE TRIGGER SQL statement:

```
mDatabase.execSQL(CREATE_TRIGGER_ADD);
```

We need to add several more triggers to help enforce our link between the author and book tables, one for updating tbl_books and one for deleting records from tbl_authors.

Creating, Updating, and Deleting Database Records

Now that we have a database set up, we need to create some data. The SQLiteDatabase class includes three convenience methods to do that. They are, as you might expect, insert(), update(), and delete().

Inserting Records

We use the insert() method to add new data to our tables. We use the ContentValues object to pair the column names to the column values for the record we want to insert. For example, here we insert a record into tbl_authors for J.K. Rowling:

```
import android.content.ContentValues;
...
ContentValues values = new ContentValues();
values.put("firstname", "J.K.");
values.put("lastname", "Rowling");
long  newAuthorID = mDatabase.insert("tbl_authors", null, values);
```

The insert() method returns the id of the newly created record. We use this author id to create book records for this author.

You might want to create simple classes (that is, class Author and class Book) to encapsulate your application record data when it is used programmatically.

Updating Records

You can modify records in the database using the update() method. The update() method takes four arguments:

- The table to update records
- A ContentValues object with the modified fields to update
- An optional WHERE clause, in which ? identifies a WHERE clause argument
- An array of WHERE clause arguments, each of which is substituted in place of the?'s from the second parameter

Passing null to the WHERE clause modifies all records within the table, which can be useful for making sweeping changes to your database.

Most of the time, we want to modify individual records by their unique identifier. The following function takes two parameters: an updated book title and a bookId. We find the record in the table called tbl_books that corresponds with the id and update that book's title. Again, we use the ContentValues object to bind our column names to our data values:

```
public void updateBookTitle(Integer bookId, String newtitle) {
     ContentValues values = new ContentValues();
     values.put("title", newtitle);
     mDatabase.update("tbl_books", values, "id=?", new String[] {
     bookId.toString() });
}
```

Because we are not updating the other fields, we do not need to include them in the ContentValues object. We include only the title field because it is the only field we change.

Deleting Records

You can remove records from the database using the remove() method. The remove() method takes three arguments:

- The table to delete the record from
- An optional WHERE clause, in which ? identifies a WHERE clause argument
- An array of WHERE clause arguments, each of which is substituted in place of the ?'s from the second parameter

Passing null to the WHERE clause deletes all records within the table. For example, this function call deletes all records within the table called tbl_authors:

```
mDatabase.delete("tbl_authors", null, null);
```

Most of the time, though, we want to delete individual records by their unique identifiers. The following function takes a parameter bookId and deletes the record corresponding to that unique id (primary key) within the table called tbl_books:

```
public void deleteBook(Integer bookId) {
mDatabase.delete("tbl_books", "id=?",
new String[] { bookId.toString() });
}
```

You need not use the primary key (id) to delete records; the WHERE clause is entirely up to you. For instance, the following function deletes all book records in the table tbl_books for a given author by the author's unique id:

```
public void deleteBooksByAuthor(Integer authorID) {
     int numBooksDeleted = mDatabase.delete("tbl_books",
     "authorid=?", new String[] { authorID.toString() });
  }
```

Working with Transactions

Often you have multiple database operations you want to happen all together or not at all. You can use SQL Transactions to group operations together; if any of the operations fails, you can handle the error and either recover or roll back all operations. If the operations all succeed, you can then commit them. Here we have the basic structure for a transaction:

```
mDatabase.beginTransaction();
try {
     // Insert some records, updated others, delete a few
     // Do whatever you need to do as a unit, then commit it
     mDatabase.setTransactionSuccessful();
} catch (Exception e) {
     // Transaction failed. Failed! Do something here.
     // It's up to you.
}  finally {
     mDatabase.endTransaction();
}
```

Now let's look at the transaction in a bit more detail. A transaction always begins with a call to beginTransaction() method and a try/catch block. If your operations are successful, you can commit your changes with a call to the setTransactionSuccessful() method. If you do not call this method, all your operations are rolled back and not committed. Finally, you end your transaction by calling endTransaction(). It's as simple as that.

In some cases, you might recover from an exception and continue with the transaction. For example, if you have an exception for a read-only database, you can open the database and retry your operations.

Finally, note that transactions can be nested, with the outer transaction either committing or rolling back all inner transactions.

Querying SQLite Databases

Databases are great for storing data in any number of ways, but retrieving the data you want is what makes databases powerful. This is partly a matter of designing an appropriate database schema, and partly achieved by crafting SQL queries, most of which are SELECT statements.

Android provides many ways in which you can query your application database. You can run raw SQL query statements (strings), use a number of different SQL statement< builder utility classes to generate proper query statements from the ground up, and bind specific user interface controls such as container views to your backend database directly.

Working with Cursors

When results are returned from a SQL query, you often access them using a Cursor found in the android.database.Cursor class. Cursor objects are rather like file pointers; they allow random access to query results.

You can think of query results as a table, in which each row corresponds to a returned record. The Cursor object includes helpful methods for determining how many results were returned by the query the Cursor represents and methods for determining the column names (fields) for each returned record. The columns in the query results are defined by the query, not necessarily by the database columns. These might include calculated columns, column aliases, and composite columns.

Cursor objects are generally kept around for a time. If you do something simple (such as get a count of records or in cases when you know you retrieved only a single simple record), you can execute your query and quickly extract what you need; don't forget to close the Cursor when you're done, as shown here:

```
// SIMPLE QUERY: select * from tbl_books
  Cursor c =
      mDatabase.query("tbl_books",null,null,null,null,null,null);
      //  Do something quick with the Cursor here...
      c.close();
```

Managing Cursors as Part of the Application Lifecycle

When a Cursor returns multiple records, or you do something more intensive, you need to consider running this operation on a thread separate from the UI thread. You also need to manage your Cursor.

Cursor objects must be managed as part of the application lifecycle. When the application pauses or shuts down, the Cursor must be deactivated with a call to the deactivate () method, and when the application restarts, the Cursor should refresh its data using the requery() method. When the Cursor is no longer needed, a call to close() must be made to release its resources.

As the developer, you can handle this by implementing Cursor management calls within the various lifecycle callbacks, such as onPause(), onResume(), and onDestroy().

If you're lazy, like us, and you don't want to bother handling these lifecycle events, you can hand off the responsibility of managing Cursor objects to the parent Activity by using the Activity method called startManagingCursor().The Activity handles the rest, deactivating and reactivating the Cursor as necessary and destroying the Cursor when the Activity is destroyed. You can always begin manually managing the Cursor object again later by simply calling stopManagingCursor().

Here we perform the same simple query and then hand over Cursor management to the parent Activity:

```
// SIMPLE QUERY: select * from tbl_books
Cursor c =
     mDatabase.query("tbl_books",null,null,null,null,null,null);
     startManagingCursor(c);
```

Note that, generally, the managed Cursor is a member variable of the class, scope-wise.

Iterating Rows of Query Results and Extracting Specific Data

You can use the Cursor to iterate those results, one row at a time using various navigation methods such as moveToFirst(), moveToNext(), and isAfterLast().

On a specific row, you can use the Cursor to extract the data for a given column in the query results. Because SQLite is not strongly typed, you can always pull fields out as Strings using the getString() method, but you can also use the type-appropriate extraction utility function to enforce type safety in your application.

For example, the following method takes a valid Cursor object, prints the number of returned results, and then prints some column information (name and number of columns). Next, it iterates through the query results, printing each record.

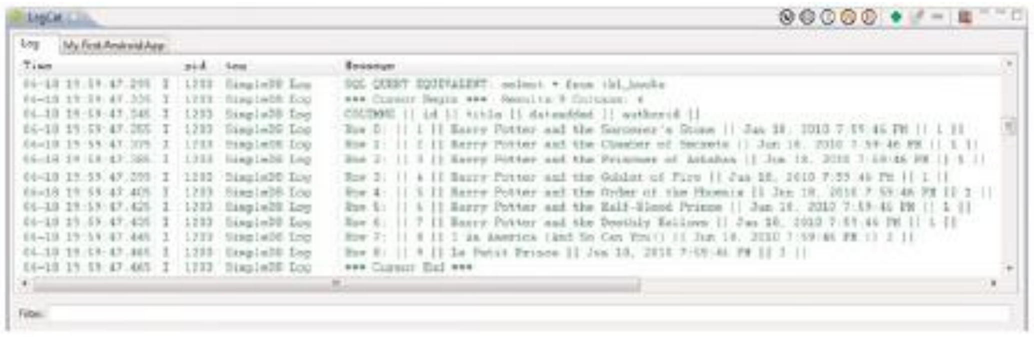
```
public void logCursorInfo(Cursor c) {
     Log.i(DEBUG_TAG, "*** Cursor Begin *** " + " Results:" +
            c.getCount() + " Columns: " + c.getColumnCount());
     // Print column names
     String rowHeaders = "|| ";
     for (int i = 0; i < c.getColumnCount(); i++) {
       rowHeaders = rowHeaders.concat(c.getColumnName(i)+"||");
     }
     Log.i(DEBUG_TAG, "COLUMNS " + rowHeaders);
     // Print records
     c.moveToFirst();
     while (c.isAfterLast() == false) {
            String rowResults = "|| ";
            for (int i = 0; i < c.getColumnCount(); i++) {
            rowResults = rowResults.concat(c.getString(i) + " || ");
     }
     Log.i(DEBUG_TAG,
     "Row " + c.getPosition() + ": " +  rowResults);
     c.moveToNext();
     }
Log.i(DEBUG_TAG, "*** Cursor End ***");
}
```

The output to the LogCat for this function might look something like Figure

Executing Simple Queries

Your first stop for database queries should be the query() methods available in the SQLiteDatabase class. This method queries the database and returns any results as in a Cursor object.

Sample log output for the logCursorInfo() method.



```
Time                 pid   tag            Message
06-18 19:59:47.295 I 1233 SimpleDB Log   SQL QUERY EQUIVALENT: select * from tbl_books
06-18 19:59:47.335 I 1233 SimpleDB Log   *** Cursor Begin *** Results:9 Columns: 4
06-18 19:59:47.345 I 1233 SimpleDB Log   COLUMNS || id || title || dateadded || authorid ||
06-18 19:59:47.355 I 1233 SimpleDB Log   Row 0: || 1 || Harry Potter and the Sorcerer's Stone || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.375 I 1233 SimpleDB Log   Row 1: || 2 || Harry Potter and the Chamber of Secrets || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.385 I 1233 SimpleDB Log   Row 2: || 3 || Harry Potter and the Prisoner of Azkaban || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.395 I 1233 SimpleDB Log   Row 3: || 4 || Harry Potter and the Goblet of Fire || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.405 I 1233 SimpleDB Log   Row 4: || 5 || Harry Potter and the Order of the Phoenix || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.425 I 1233 SimpleDB Log   Row 5: || 6 || Harry Potter and the Half-Blood Prince || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.435 I 1233 SimpleDB Log   Row 6: || 7 || Harry Potter and the Deathly Hallows || Jun 18, 2010 7:59:46 PM || 1 ||
06-18 19:59:47.445 I 1233 SimpleDB Log   Row 7: || 8 || I am America (And So Can You!) || Jun 18, 2010 7:59:46 PM || 2 ||
06-18 19:59:47.465 I 1233 SimpleDB Log   Row 8: || 9 || Le Petit Prince || Jun 18, 2010 7:59:46 PM || 3 ||
06-18 19:59:47.465 I 1233 SimpleDB Log   *** Cursor End ***
```

The query() method we mainly use takes the following parameters:

- [String]:The name of the table to compile the query against
- [String Array]: List of specific column names to return (use null for all)
- [String] The WHERE clause: Use null for all; might include selection args as ?'s
- [String Array]: Any selection argument values to substitute in for the ?'s in the earlier parameter
- [String] GROUP BY clause: null for no grouping
- [String] HAVING clause: null unless GROUP BY clause requires one
- [String] ORDER BY clause: If null, default ordering used
- [String] LIMIT clause: If null, no limit

Previously in the chapter, we called the query() method with only one parameter set to the table name.

```
Cursor c = mDatabase.query("tbl_books",null,null,null,null,null,null);
```

This is equivalent to the SQL query

```
      SELECT * FROM tbl_books;
```

Add a WHERE clause to your query, so you can retrieve one record at a time:

```
  Cursor c = mDatabase.query("tbl_books", null,  "id=?", new
String[]{"9"}, null, null, null);
```

This is equivalent to the SQL query

```
SELECT * tbl_books WHERE id=9;
```

Selecting all results might be fine for tiny databases, but it is not terribly efficient. You should always tailor your SQL queries to return only the results you require with no extraneous information included. Use the powerful language of SQL to do the heavy lifting for you whenever possible, instead of programmatically processing results yourself. For example, if you need only the titles of each book in the book table, you might use the following call to the query() method:

```
String asColumnsToReturn[] = { "title", "id" };
String strSortOrder = "title ASC";
Cursor c = mDatabase.query("tbl_books", asColumnsToReturn,
null, null, null, null, strSortOrder);
```

This is equivalent to the SQL query

```
SELECT title, id FROM tbl_books ORDER BY title ASC;
```

Executing More Complex Queries Using SQLiteQueryBuilder

As your queries get more complex and involve multiple tables, you should leverage the SQLiteQueryBuilder convenience class, which can build complex queries (such as joins) programmatically.

When more than one table is involved, you need to make sure you refer to columns within a table by their fully qualified names. For example, the title column within the tbl_books table is tbl_books.title. Here we use a SQLiteQueryBuilder to build and execute a simple INNER JOIN between two tables to get a list of books with their authors:

```
import android.database.sqlite.SQLiteQueryBuilder;
...
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
queryBuilder.setTables("tbl_books, tbl_authors");
queryBuilder.appendWhere("tbl_books.authorid=tbl_authors.id");
String asColumnsToReturn[] = {
"tbl_books.title",
"tbl_books.id",
"tbl_authors.firstname",
"tbl_authors.lastname",
"tbl_books.authorid" };
String strSortOrder = "title ASC";
Cursor c = queryBuilder.query(mDatabase, asColumnsToReturn,
null, null, null, null,strSortOrder);
```

First, we instantiate a new SQLiteQueryBuilder object. Then we can set the tables involved as part of our JOIN and the WHERE clause that determines how the JOIN occurs. Then, we call the query() method of the SQLiteQueryBuilder that is similar to the query() method we have been using, except we supply the SQLiteDatabase instance instead of the table name. The earlier query built by the SQLiteQueryBuilder is equivalent to the SQL query:

```
SELECT tbl_books.title,
tbl_books.id,
tbl_authors.firstname,
tbl_authors.lastname,
tbl_books.authorid
FROM tbl_books
INNER JOIN tbl_authors on tbl_books.authorid=tbl_authors.id
ORDER BY title ASC;
```

Executing Raw Queries Without Builders and Column-Mapping

All these helpful Android query utilities can sometimes make building and performing a nonstandard or complex query too verbose. In this case, you might want to consider the rawQuery() method.The rawQuery() method simply takes a SQL statement String (with optional selection arguments if you include ?'s) and returns a Cursor of results. If you know your SQL and you don't want to bother learning the ins and outs of all the different SQL query building utilities, this is the method for you.

For example, let's say we have a UNION query. These types of queries are feasible with the QueryBuilder, but their implementation is cumbersome when you start using column aliases and the like.

Let's say we want to execute the following SQL UNION query, which returns a list of all book titles and authors whose name contains the substring ow (that is *Hallows, Rowling*), as in the following:

```
SELECT title AS Name,
'tbl_books' AS OriginalTable
FROM tbl_books
WHERE Name LIKE '%ow%'
UNION
SELECT (firstname||' '|| lastname) AS Name,
'tbl_authors' AS OriginalTable
FROM tbl_authors
WHERE Name LIKE '%ow%'
ORDER BY Name ASC;
```

We can easily execute this by making a string that looks much like the original query and executing the rawQuery() method.

```
String sqlUnionExample = "SELECT title AS Name, 'tbl_books' AS
OriginalTable from tbl_books WHERE Name LIKE ? UNION SELECT
(firstname||' '|| lastname) AS Name, 'tbl_authors' AS OriginalTable
from tbl_authors WHERE Name LIKE ? ORDER BY Name ASC;";
Cursor c = mDatabase.rawQuery(sqlUnionExample,
new String[]{ "%ow%", "%ow%"});
```

We make the substrings (ow) into selection arguments, so we can use this same code to look for other substrings searches).

Closing and Deleting a SQLite Database

Although you should always close a database when you are not using it, you might on occasion also want to modify and delete tables and delete your database.

Deleting Tables and Other SQLite Objects

You delete tables and other SQLite objects in exactly the same way you create them. Format the appropriate SQLite statements and execute them. For example, to drop our tables and triggers,we can execute three SQL statements:

```
mDatabase.execSQL("DROP TABLE tbl_books;");
mDatabase.execSQL("DROP TABLE tbl_authors;");
mDatabase.execSQL("DROP TRIGGER IF EXISTS fk_insert_book;");
```

Closing a SQLite Database

You should close your database when you are not using it. You can close the database using the close() method of your SQLiteDatabase instance, like this:

```
    mDatabase.close();
```

Deleting a SQLite Database Instance Using the Application Context

The simplest way to delete a SQLiteDatabase is to use the deleteDatabase() method of your application Context. You delete databases by name and the deletion is permanent. You lose all data and schema information.

```
    deleteDatabase("my_sqlite_database.db");
```

Designing Persistent Databases

Generally speaking, an application creates a database and uses it for the rest of the application's lifetime—by which we mean until the application is uninstalled from the phone. So far, we've talked about the basics of creating a database, using it, and then deleting it.

In reality, most mobile applications do not create a database on-the-fly, use them, and then delete them. Instead, they create a database the first time they need it and then use it. The Android SDK provides a helper class called SQLiteOpenHelper to help you manage your application's database.

To create a SQLite database for your Android application using the SQLiteOpenHelper, you need to extend that class and then instantiate an instance of it as a member variable for use within your application. To illustrate how to do this, let's create a new Android project called PetTracker.

Keeping Track of Database Field Names

You've probably realized by now that it is time to start organizing your database fields programmatically to avoid typos and such in your SQL queries. One easy way you do this is to make a class to encapsulate your database schema in a class, such as PetDatabase, shown here:

```
 import android.provider.BaseColumns;
 public final class PetDatabase {
     private PetDatabase() {}
     public static final class Pets implements BaseColumns {
       private Pets() {}
       public static final String PETS_TABLE_NAME=”table_pets”;
       public static final String PET_NAME=”pet_name”;
       public static final String PET_TYPE_ID=”pet_type_id”;
       public static final String DEFAULT_SORT_ORDER=”pet_name ASC”;
 }
 public static final class PetType implements BaseColumns {
     private PetType() {}
 public static final String PETTYPE_TABLE_NAME=”table_pettypes”;
 public static final String PET_TYPE_NAME=”pet_type”;
 public static final String DEFAULT_SORT_ORDER=”pet_type ASC”;
 }
}
```

By implementing the BaseColumns interface, we begin to set up the underpinnings for using database-friendly user interface controls in the future, which often require a specially named column called _id to function properly. We rely on this column as our primary key.

Extending the SQLiteOpenHelper Class

To extend the SQLiteOpenHelper class, we must implement several important methods, which help manage the database versioning. The methods to override are onCreate(), onUpgrade(), and onOpen().We use our newly defined PetDatabase class to generate appropriate SQL statements, as shown here:

```
import android.content.Context;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import com.androidbook.PetTracker.PetDatabase.PetType;
import com.androidbook.PetTracker.PetDatabase.Pets;
class PetTrackerDatabaseHelper extends SQLiteOpenHelper {
private static final String DATABASE_NAME = “pet_tracker.db”;
private static final int DATABASE_VERSION = 1;
PetTrackerDatabaseHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
@Override
public void onCreate(SQLiteDatabase db) {
db.execSQL("CREATE TABLE " +PetType.PETTYPE_TABLE_NAME+" ("
+ PetType._ID + " INTEGER PRIMARY KEY  AUTOINCREMENT ,"
+ PetType.PET_TYPE_NAME + " TEXT"
+ ");");
db.execSQL("CREATE TABLE " + Pets.PETS_TABLE_NAME + " ("
+ Pets._ID + " INTEGER PRIMARY KEY AUTOINCREMENT  ,"
+ Pets.PET_NAME + " TEXT,"
+ Pets.PET_TYPE_ID + " INTEGER" // FK to pet type  table
+ ");");
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion,
int newVersion){
// Housekeeping here.
// Implement how "move" your application data
// during an upgrade of schema versions
// Move or delete data as required. Your call.
}
@Override
public void onOpen(SQLiteDatabase db) {
super.onOpen(db);
}
}
```

Now we can create a member variable for our database like this:

```
PetTrackerDatabaseHelper mDatabase = new
PetTrackerDatabaseHelper(this.getApplicationContext());
```

Now, whenever our application needs to interact with its database,we request a valid database object. We can request a read-only database or a database that we can also write to. We can also close the database. For example, here we get a database we can write data to:

SQLiteDatabase db = mDatabase.getWritableDatabase();

Binding Data to the Application User Interface

In many cases with application databases, you want to couple your user interface with the data in your database. You might want to fill drop-down lists with values from a database table, or fill out form values, or display only certain results. There are various ways to bind database data to your user interface. You, as the developer, can decide whether to use builtin data-binding functionality provided with certain user interface controls, or you can build your own user interfaces from the ground up.

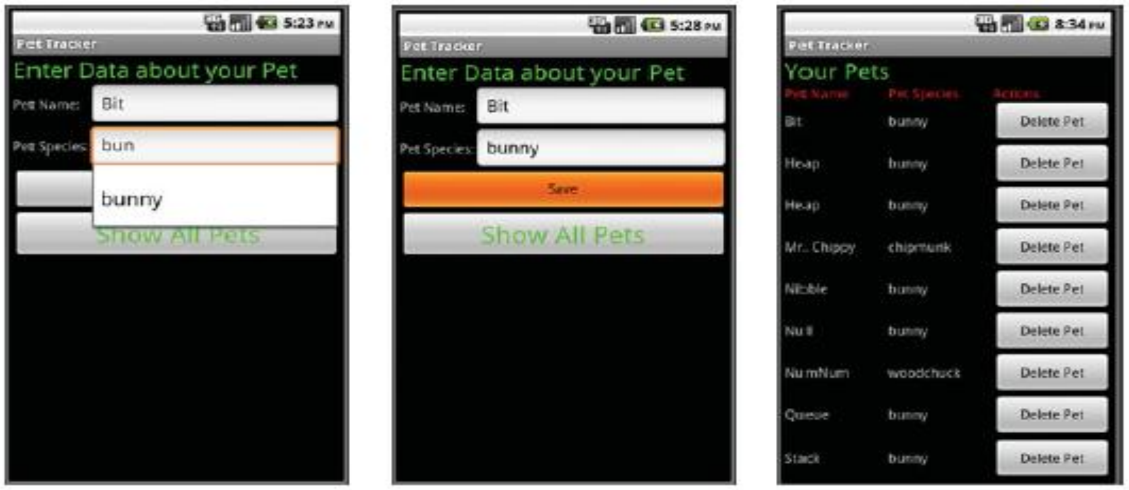
Working with Database Data Like Any Other Data

If you peruse the PetTracker application provided on the book website, you notice that its functionality includes no magical data-binding features, yet the application clearly uses the database as part of the user interface.

Specifically, the database is leveraged:

- When you fill out the Pet Type field, the AutoComplete feature is seeded with pet types already in listed in the table_pettypes table (left).

The PetTracker application: Entry Screen (left, middle) and Pet Listing Screen (right).



Pet Name: Bit
Pet Species: bun
bunny

Pet Name: Bit
Pet Species: bunny

Pet Name	Pet Species	Actions
Bit	bunny	Delete Pet
Heap	bunny	Delete Pet
Heap	bunny	Delete Pet
Mr. Chippy	chipmunk	Delete Pet
Nibble	bunny	Delete Pet
Null	bunny	Delete Pet
NumNum	woodchuck	Delete Pet
Queue	bunny	Delete Pet
Stack	bunny	Delete Pet

- When you save new records using the Pet Entry Form (middle).
- When you display the Pet List screen, you query for all pets and use a Cursor to programmatically build a TableLayout on-the-fly (right).

This might work for small amounts of data; however, there are various drawbacks to this method. For example, all the work is done on the main thread, so the more records you add, the slower your application response time becomes. Second, there’s quite a bit of custom code involved to map the database results to the individual user interface components. If you decide you want to use a different control to display your data, you have quite a lot of rework to do. Third, we constantly requery the database for fresh results, and we might be requerying far more than necessary.

Yes, we really named our pet bunnies after data structures and computer terminology. We are that geeky. Null, for example, is a rambunctious little black bunny. Shane enjoys pointing at him and calling himself a Null pointer.

Binding Data to Controls Using Data Adapters

Ideally, you’d like to bind your data to user interface controls and let them take care of the data display. For example, we can use a fancy ListView to display the pets instead of building a TableLayout from scratch. We can spin through our Cursor and generate ListView child items manually, or even better, we can simply create a data adapter to map the Cursor results to each TextView child within the ListView.

We included a project called PetTracker2 on the book website that does this. It behaves much like the PetTracker sample application, except that it uses the SimpleCursorAdapter with ListView and an ArrayAdapter to handle AutoCompleteTextView features.

The source code for subsequent upgrades to the PetTracker application (for example, Pet-Tracker2, PetTracker3, and so on) is provided for download on the book website.

Binding Data Using SimpleCursorAdapter

Let’s now look at how we can create a data adapter to mimic our Pet Listing screen, with each pet’s name and species listed. We also want to continue to have the ability to delete records from the list.

Remember from Chapter “Designing User Interfaces with Layouts,” that the ListView container can contain children such as TextView objects. In this case, we want to display each Pet’s name and type. We therefore create a layout file called pet_item.xml that becomes our ListView item template:

```
<?xml version=”1.0” encoding=”utf-8”?>
<RelativeLayout
     xmlns:android=”http://schemas.android.com/apk/res/android”
     android:id=”@+id/RelativeLayoutHeader”
     android:layout_height=”wrap_content”
     android:layout_width=”fill_parent”>
<TextView
     android:id=”@+id/TextView_PetName”
     android:layout_width=”wrap_content”
     android:layout_height=”?android:attr/listPreferredItemHeight”
     android:layout_alignParentLeft=”true” />
<TextView
     android:id=”@+id/TextView_PetType”
     android:layout_width=”wrap_content”
     android:layout_height=”?android:attr/listPreferredItemHeight”
     android:layout_alignParentRight=”true” />
</RelativeLayout>
```

Next, in our main layout file for the Pet List, we place our ListView in the appropriate place on the overall screen. The ListView portion of the layout file might look something like this:

```
<ListView
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:id="@+id/petList" android:divider="#000"  />
```

Now to programmatically fill our ListView,we must take the following steps:

1. Perform our query and return a valid Cursor (a member variable).
2. Create a data adapter that maps the Cursor columns to the appropriate TextView controls within our pet_item.xml layout template.
3. Attach the adapter to the ListView.

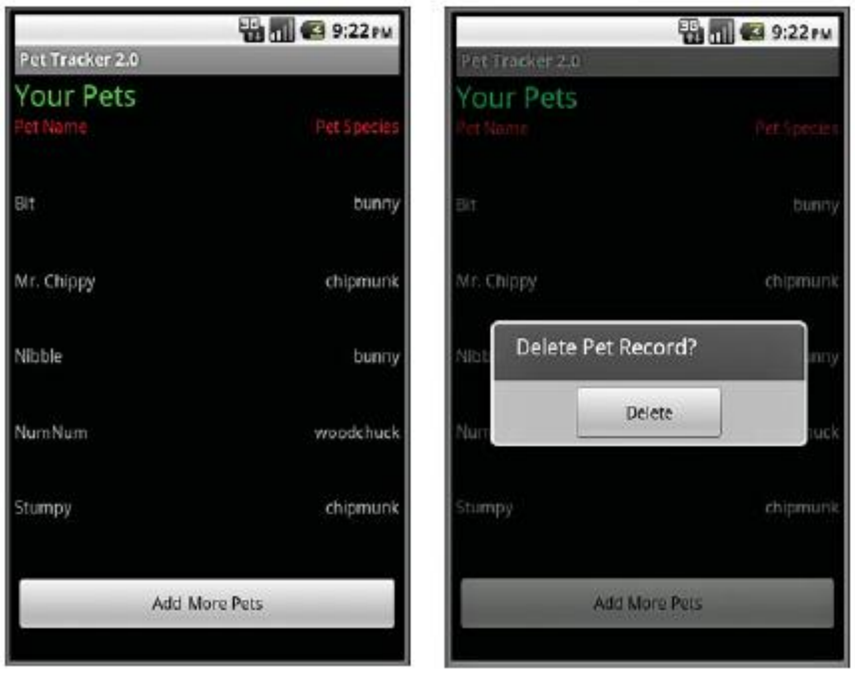
In the following code,we perform these steps:

```
SQLiteQueryBuilder queryBuilder = new SQLiteQueryBuilder();
queryBuilder.setTables(Pets.PETS_TABLE_NAME  +", " +
PetType.PETTYPE_TABLE_NAME);
queryBuilder.appendWhere(Pets.PETS_TABLE_NAME  + "." +
Pets.PET_TYPE_ID + "=" +  PetType.PETTYPE_TABLE_NAME + "." +
PetType._ID);
String asColumnsToReturn[] = { Pets.PETS_TABLE_NAME + "." +
Pets.PET_NAME, Pets.PETS_TABLE_NAME +
"." + Pets._ID, PetType.PETTYPE_TABLE_NAME + "."  +
PetType.PET_TYPE_NAME };
mCursor = queryBuilder.query(mDB, asColumnsToReturn, null, null,
null, null, Pets.DEFAULT_SORT_ORDER);
startManagingCursor(mCursor);
ListAdapter adapter = new SimpleCursorAdapter(this,
R.layout.pet_item, mCursor,
new String[]{Pets.PET_NAME, PetType.PET_TYPE_NAME},
new int[]{R.id.TextView_PetName, R.id.TextView_PetType });
ListView av = (ListView)findViewById(R.id.petList);
av.setAdapter(adapter);
```

Notice that the _id column as well as the expected name and type columns appears in the query. This is required for the adapter and ListView to work properly.

Using a ListView instead of a custom user interface enables us to take advantage of the ListView control's built-in features, such as scrolling when the list becomes longer, and the ability to provide context menus as needed. The _id column is used as the unique identifier for each ListView child node. If we choose a specific item on the list, we can act on it using this identifier, for example, to delete the item.

The PetTracker2 application: Pet Listing Screen ListView (left) with Delete feature (right).



Now we re-implement the Delete functionality by listening for onItemClick() events and providing a Delete Confirmation dialog (right):

```
av.setOnItemClickListener(new AdapterView.OnItemClickListener() {
  public void onItemClick( AdapterView<?> parent, View view,
   int position, long id) {
   final long deletePetId = id;
new AlertDialog.Builder(PetTrackerListActivity.this).setMessage(
"Delete Pet Record?").setPositiveButton(
"Delete", new  DialogInterface.OnClickListener() {
@Override
public void onClick(DialogInterface dialog,int which) {
deletePet(deletePetId);
mCursor.requery();
}}).show();
}
});
```

You can see what this would look like on the screen in Figure.

Note that within the PetTracker2 sample application, we also use an ArrayAdapter to bind the data in the pet_types table to the AutoCompleteTextView on the Pet Entry screen. Although our next example shows you how to do this in a preferred manner, we left this code in the PetTracker sample to show you that you can always intercept the data your Cursor provides and do what you want with it. In this case, we create a String array for the AutoText options by hand. We use a built-in Android layout resource called android.R.layout.simple_dropdown_item_1line to specify what each individual item within the AutoText listing looks like. You can find the built-in layout resources provided within your appropriate Android SDK version's resource subdirectory.

Storing Nonprimitive Types (Such as Images) in the Database

Because SQLite is a single file, it makes little sense to try to store binary data within the database. Instead store the *location* of data, as a file path or a URI in the database, and access it appropriately. We show an example of storing image URIs in the database in the next chapter.

Sharing Data between Applications with Content Provider

RKUNIVERSITY™

By : Ketan Bhimani

Sharing Data Between Applications with Content Providers

Applications can access data within other applications on the Android system through content provider interfaces and expose internal application data to other applications by becoming a content provider.

First, we take a look at some of the other content providers available on the Android platform and what you can do with them. Next, you see some examples of how to use content providers to improve the sample applications used in previous chapters. Finally, you learn how applications can become content providers to share information— for example, with LiveFolders.

Exploring Android's Content Providers

Android devices ship with a number of built-in applications, many of which expose their data as content providers. Your application can access content provider data from a variety of sources. You can find the content providers included with Android in the package android.provider. Table lists some useful content providers in this package.

Useful Built-In Content Providers

Provider	Purpose
MediaStore	Audio-visual data on the phone and external storage
CallLog	Sent and received calls
Browser	Browser history and bookmarks
ContactsContract	Phone contact database or phonebook
Settings	System-wide device settings and preferences
UserDictionary	A dictionary of user-defined words for use with predictive text input

Now let’s look at the individual content providers in more detail.

Using the MediaStore Content Provider

You can use the MediaStore content provider to access media on the phone and on external storage devices. The primary types of media that you can access are audio, images, and video. You can access these different types of media through their respective content provider classes under android.provider.MediaStore.

Most of the MediaStore classes allow full interaction with the data. You can retrieve, add, and delete media files from the device. There are also a handful of helper classes that define the most common data columns that can be requested.

Table lists some commonly used classes that you can find under android. provider. MediaStore.

Common MediaStore Classes

Class	Purpose
`Video.Media`	Manages video files on the device
`Images.Media`	Manages image files on the device
`Images.ThumbNails`	Retrieves thumbnails for the images
`Audio.Media`	Manages audio files on the device
`Audio.Albums`	Manages audio files organized by the album
`Audio.Artists`	Manages audio files by the artist who created them
`Audio.Genres`	Manages audio files belonging to a particular genre
`Audio.Playlists`	Manages audio files that are part of a particular playlist

The following code demonstrates how to request data from a content provider. A query is made to the MediaStore to retrieve the titles of all the audio files on the SD card of the handset and their respective durations. This code requires that you load some audio files onto the virtual SD card in the emulator.

```
String[] requestedColumns = {
    MediaStore.Audio.Media.TITLE,
    MediaStore.Audio.Media.DURATION
};
Cursor cur = managedQuery(
    MediaStore.Audio.Media.EXTERNAL_CONTENT_URI,
    requestedColumns, null, null, null);
    Log.d(DEBUG_TAG, "Audio files: " + cur.getCount());
    Log.d(DEBUG_TAG, "Columns: " + cur.getColumnCount());
    String[] columns = cur.getColumnNames();
    int name = cur.getColumnIndex(MediaStore.Audio.Media.TITLE);
    int size = cur.getColumnIndex(MediaStore.Audio.Media.DURATION);
    cur.moveToFirst();
    while (!cur.isAfterLast()){Log.d(DEBUG_TAG,"Title"+
            cur.getString(name));
    Log.d(DEBUG_TAG, "Length: " +  cur.getInt(size) / 1000 + " seconds");
    cur.moveToNext();
}
```

The MediaStore.Audio.Media class has predefined strings for every data field (or column) exposed by the content provider. You can limit the audio file data fields requested as part of the query by defining a string array with the column names required. In this case, we limit the results to only the track title and the duration of each audio file.

We then use a managedQuery() method call. The first parameter is the predefined URI of the content provider you want to query (in most cases, the primary external storage is the SD card).The second parameter is the list of columns to return (audio file titles and durations).The third and fourth parameters control any selection filtering arguments, and the fifth parameter provides a sort method for the results. We leave these null, as we want all audio files at this location. By using the managedQuery() method, we get a managed Cursor as a result. We then examine our Cursor for the results.

Using the CallLog Content Provider

Android provides a content provider to access the call log on the handset via the class android.provider.CallLog. At first glance, the CallLog might not seem to be a useful provider for developers, but it has some nifty features. You can use the CallLog to filter recently dialed calls, received, and missed calls. The date and duration of each call is logged and tied back to the Contact application for caller identification purposes.

The CallLog is a useful content provider for customer relationship management (CRM) applications. The user can also tag specific phone numbers with custom labels within the Contact application.

To demonstrate how the CallLog content provider works, let's look at a hypothetical situation where we want to generate a report of all calls to a number with the custom labeled HourlyClient123.Android allows for custom labels on these numbers, which we leverage for this example:

```
String[] requestedColumns = {
     CallLog.Calls.CACHED_NUMBER_LABEL,CallLog.Calls.DURATION
     };
Cursor calls = managedQuery(
CallLog.Calls.CONTENT_URI,  requestedColumns,
CallLog.Calls.CACHED_NUMBER_LABEL
+ " = ?",  new String[] { "HourlyClient123" } , null);
Log.d(DEBUG_TAG, "Call count: " + calls.getCount());
int durIdx = calls.getColumnIndex(CallLog.Calls.DURATION);
int totalDuration = 0;
```

```
calls.moveToFirst();
while (!calls.isAfterLast()) {
Log.d(DEBUG_TAG, "Duration: " + calls.getInt(durIdx));
totalDuration  += calls.getInt(durIdx);
calls.moveToNext();
}
Log.d(DEBUG_TAG, "HourlyClient123 Total Call Duration: " +
     totalDuration);
```

This code is similar to the code shown for the MediaStore audio files.Again, we start with listing our requested columns: the call label and the duration of the call. This time, however, we don't want to get every call in the log, only those with a label of HourlyClient123.To filter the results of the query to this specific label, it is necessary to specify the third and fourth parameters of the managedQuery() call. Together, these two parameters are equivalent to a database WHERE clause. The third parameter specifies the format of the WHERE clause with the column name with selection parameters (shown as ?s) for each selection argument value. The fourth parameter, the String array, provides the values to substitute for each of the selection arguments (?s) in order as you would do for a simple SQLite database query.

As before, the Activity manages the Cursor object lifecycle. We use the same method to iterate the records of the Cursor and add up all the call durations.

Accessing Content Providers That Require Permissions

Your application needs a special permission to access the information provided by the CallLog content provider. You can declare the uses-permission tag using the Eclipse Wizard or by adding the following to your AndroidManifest.xml file:

```
<uses-permission
xmlns:android=http://schemas.android.com/apk/res/android
```

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

Although it’s a tad confusing, there is no CallLog permission. Instead, applications that access the CallLog use the READ_CONTACTS permission. Although the values are cached within this content provider, the data is similar to what you might find in the contacts provider.

Using the Browser Content Provider

Another useful, built-in content provider is the Browser. The Browser content provider exposes the user’s browser site history and their bookmarked websites. You access this content provider via the android.provider.Browser class. As with the CallLog class, you can use the information provided by the Browser content provider to generate statistics and to provide cross-application functionality. You might use the Browser content provider to add a bookmark for your application support website.

In this example, we query the Browser content provider to find the top five most frequently visited bookmarked sites.

```
String[] requestedColumns = {
     Browser.BookmarkColumns.TITLE, Browser.BookmarkColumns.VISITS,
     Browser.BookmarkColumns.BOOKMARK
     };
Cursor faves = managedQuery(Browser.BOOKMARKS_URI, requestedColumns,
Browser.BookmarkColumns.BOOKMARK  + “=1”, null,
Browser.BookmarkColumns.VISITS  + “ DESC limit 5”);
Log.d(DEBUG_TAG, “Bookmarks count: “ + faves.getCount());
int titleIdx = faves.getColumnIndex(Browser.BookmarkColumns.TITLE);
int visitsIdx = aves.getColumnIndex(Browser.BookmarkColumns.VISITS);
int bmIdx = faves.getColumnIndex(Browser.BookmarkColumns.BOOKMARK);
faves.moveToFirst();
```

```
while (!faves.isAfterLast()) {
Log.d("SimpleBookmarks", faves.getString(titleIdx)+ "visited" +
faves.getInt(visitsIdx)+"times:"+(faves.getInt(bmIdx)!=0?"true":
"false"));
faves.moveToNext();
}
```

Again,the requested columns are defined, the query is made, and the cursor iterates through the results.

Note that the managedQuery() call has become substantially more complex. Let’s take a look at the parameters to this method in more detail. The first parameter, Browser .BOOKMARKS_URI, is a URI for all browser history, not only the Bookmarked items. The second parameter defines the requested columns for the query results. The third parameter specifies that the bookmark property must be true. This parameter is needed in order to filter within the query. Now the results are only browser history entries that have been bookmarked. The fourth parameter, selection arguments, is used only when replacement values are used, which is not used in this case, so the value is set to null. Lastly, the fifth parameter specifies an order to the results (most visited in descending order). Retrieving browser history information requires setting the READ_ HISTORY_ BOOKMARKS permission.

Using the Contacts Content Provider

The Contacts database is one of the most commonly used applications on the mobile phone. People always want phone numbers handy for calling friends, family, coworkers, and clients.Additionally, most phones show the identity of the caller based on the contacts application, including nicknames, photos, or icons.

Android provides a built-in Contact application, and the contact data is exposed to other Android applications using the content provider interface. As an application developer, this means you can leverage the user's contact data within your application for a more robust user experience.

Accessing Private Contact Data

Your application needs special permission to access the private user information provided by the Contacts content provider. You must declare a uses-permission tag using the permission READ_CONTACTS to read this information. The code to start reading contact data from the Contacts application should look familiar.

```
Cursor oneContact = managedQuery( People.CONTENT_URI, null, null,
null, "name desc LIMIT 1");
Log.d(debugTag, "Count: " + oneContact.getCount());
```

This short example simply shows querying for a single contact. We used LIMIT to retrieve one contact record. If you actually look at the returned columns of data, you find that there is little more than the contact name and some indexes. The data fields are not explicitly returned. Instead, the results include the values needed to build specific URIs to those pieces of data. We need to request the data for the contact using these indexes.

Specifically,we retrieve the primary email and primary phone number for this contact.

```
int nameIdx = oneContact.getColumnIndex(Contacts.People.NAME);
int emailIDIdx = oneContact
  .getColumnIndex(Contacts.People.PRIMARY_EMAIL_ID);
```

```
int phoneIDIdx = oneContact
  .getColumnIndex(Contacts.People.PRIMARY_PHONE_ID);
oneContact.moveToFirst();
int emailID = oneContact.getInt(emailIDIdx);
int phoneID = oneContact.getInt(phoneIDIdx);
```

Now that we have the column index values for the contact's name, primary email address, and primary phone number, we need to build the Uri objects associated with those pieces of information and query for the primary email and primary phone number.

```
Uri emailUri = ContentUris.withAppendedId(
Contacts.ContactMethods.CONTENT_URI,
emailID);
Uri phoneUri = ContentUris.withAppendedId(
Contacts.Phones.CONTENT_URI, phoneID);
Cursor primaryEmail = managedQuery(emailUri,
new String[] {
Contacts.ContactMethods.DATA
},
null, null, null);
Cursor primaryNumber = managedQuery(phoneUri,
new String[] {
Contacts.Phones.NUMBER
},
null, null, null);
```

After retrieving the appropriate column indexes for a contact's specific email and phone number, we call ContentUris.withAppendedId() to create the new Uri objects from existing ones and the identifiers we now have. This enables direct selection of a particular row from the table when the index of that row is known. You can use a selection parameter to do this, as well. Lastly, we used the two new Uri objects to perform two calls to managed Query().

Now we take a shortcut with the requested columns String array because each query only has one column:

```
String name = oneContact.getString(nameIdx);
primaryEmail.moveToFirst();
```

```
String email = primaryEmail.getString(0);
primaryNumber.moveToFirst();
String number = primaryNumber.getString(0);
```

If an email or phone number doesn’t exist, an exception called android. database. Cursor IndexOut OfBounds Exception is thrown. This can be caught, or you can check to see that a result was actually returned in the Cursor first.

Querying for a Specific Contact

If that seemed like quite a lot of coding to get a phone number, you’re not alone. For getting a quick piece of data, there is a faster way. The following block of code demonstrates how we can get the primary number and name for one contact. The primary number for a contact is designated as the default number within the contact manager on the handset. It might be useful to use the primary number field if you don’t get any results back from the query.

```
String[] requestedColumns = {
     Contacts.Phones.NAME,
     Contacts.Phones.NUMBER,
};
Cursor contacts = managedQuery(Contacts.Phones.CONTENT_URI,
  requestedColumns, Contacts.Phones.ISPRIMARY  + "<>0",
  null, "name desc limit 1");
Log.d(debugTag, "Contacts count: "  + contacts.getCount());
int nameIdx = contacts.getColumnIndex(Contacts.Phones.NAME);
int phoneIdx = contacts.getColumnIndex(Contacts.Phones.NUMBER);
contacts.moveToFirst();
Log.d(debugTag, "Name: " + contacts.getString(nameIdx));
Log.d(debugTag, "Phone: " + contacts.getString(phoneIdx));
```

This block of code should look somewhat familiar, yet it is a much shorter and more straightforward method to query for phone numbers by Contact name. The Contacts. Phones. CONTENT _URI contains phone numbers but it also happens to have the contact name. This is similar to the CallLog content provider.

Using the UserDictionary Content Provider

Another useful content provider is the UserDictionary provider. You can use this content provider for predictive text input on text fields and other user input mechanisms. Individual words stored in the dictionary are weighted by frequency and organized by locale. You can use the addWord() method within the UserDictionary. Words class to add words to the custom user dictionary.

Using the Settings Content Provider

Another useful content provider is the Settings provider. You can use this content provider to access the device settings and user preferences. Settings are organized much as they are in the Settings application—by category. You can find information about the Settings content provider in the android.provider.Settings class.

Modifying Content Providers Data

Content providers are not only static sources of data. They can also be used to add, update, and delete data, if the content provider application has implemented this functionality. Your application must have the appropriate permissions (that is, WRITE_ CONTACTS as opposed to READ_ CONTACTS) to perform some of these actions.

Adding Records

Using the Contacts content provider, we can, for example, add a new record to the contacts database programmatically.

```
ContentValues values = new ContentValues();
values.put(Contacts.People.NAME, "Sample User");
Uri uri = getContentResolver().insert(
Contacts.People.CONTENT_URI, values);
Uri phoneUri = Uri.withAppendedPath(uri,
Contacts.People.Phones.CONTENT_DIRECTORY);
values.clear();
values.put(Contacts.Phones.NUMBER, "2125551212");
values.put(Contacts.Phones.TYPE, Contacts.Phones.TYPE_WORK);
getContentResolver().insert(phoneUri, values);
values.clear();
values.put(Contacts.Phones.NUMBER, "3135551212");
values.put(Contacts.Phones.TYPE, Contacts.Phones.TYPE_MOBILE);
getContentResolver().insert(phoneUri, values);
```

Just as we used the ContentValues class to insert records into an application's SQLite database,we use it again here. The first action we take is to provide a name for the Contacts.People.NAME column. We need to create the contact with a name before we can assign information, such as phone numbers. Think of this as creating a row in a table that provides a one-to-many relationship to a phone number table.

Next, we insert the data in the database found at the Contacts.People.CONTENT_URI path. We use a call to getContentResolver() to retrieve the ContentResolver associated with our Activity. The return value is the Uri of our new contact. We need to use it for adding phone numbers to our new contact. We then reuse the ContentValues instance by clearing it and adding a Contacts.Phones.NUMBER and the Contacts.Phones.TYPE for it. Using the ContentResolver,we insert this data into the newly created Uri.

Updating Records

Inserting data isn't the only change you can make. You can update one or more rows, as well. The following block of code shows how to update data within a content provider. In this case, we update a note field for a specific contact, using its unique identifier.

```
ContentValues values = new ContentValues();
values.put(People.NOTES, "This is my boss");
Uri updateUri = ContentUris.withAppendedId(People.CONTENT_URI,
  rowId);
int rows = getContentResolver().update(updateUri, values, null,
  null);
Log.d(debugTag, "Rows updated: " + rows);
```

Again, we use an instance of the ContentValues object to map the data field we want to update with the data value—in this case, the note field. This replaces any current note stored in the NOTES field currently stored with the contact. We then create the Uri for the specific contact we are updating. A simple call to the update() method of the Content Resolver class completes our change. We can then confirm that only one row was updated.

Deleting Records

Now that you cluttered up your contacts application with sample user data, you might want to delete some of it. Deleting data is fairly straightforward.

Deleting All Records

The following code deletes all rows at the given URI, although you should execute operations like this with extreme care:

```
int rows = getContentResolver().delete(People.CONTENT_URI, null,
   null);
Log.d(debugTag, "Rows: "+ rows);
```

The delete() method deletes all rows at a given URI filtered by the selection parameters, which, in this case, includes all rows at the People.CONTENT_URI location; in other words, all contact entries.

Deleting Specific Records

Often you want to select specific rows to delete by adding the unique identifier index to the end of the URI or remove rows matching a particular pattern.

For example, the following deletion matches all contact records with the name Sample User, which we used when we created sample contacts previously in the chapter.

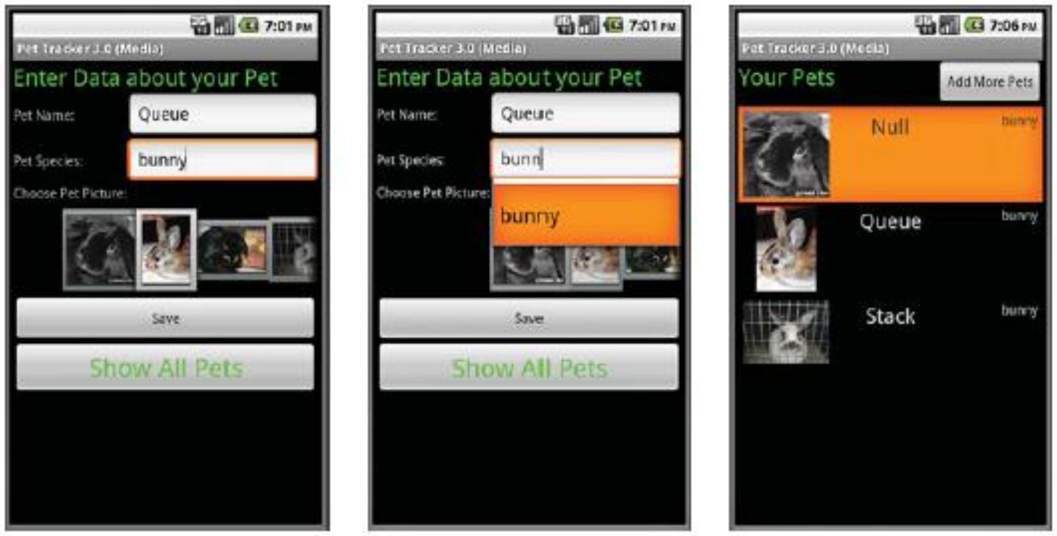
```
int rows = getContentResolver().delete(People.CONTENT_URI,
People.NAME  + "=?", new String[] {"Sample User"});
Log.d(debugTag, "Rows: "+ rows);
```

Enhancing Applications Using Content Providers

The concept of a content provider is complex and best understood by working through an example. The Pet Tracker series of applications from the previous chapter are nice and all, but the application could really use some graphics. Wouldn't it be great if we could include photos for each pet record? Well, let's do it! There's only one catch: We need to access pictures provided through another application on the Android system—the Media Store application.

In Figure, you can see the results of extending the previous Pet Tracking projects using the Media Store content provider.

Pet Tracker application: Entry Screen (left, middle) and Pet Listing Screen (right).



Accessing Images on the Device

Now that you can visualize what adding photos looks like,let's break down the steps needed to achieve this feature. The PetTracker3 application has the same basic structure as our previous Pet Tracker projects, with several key differences:

- On the Pet Entry screen, you can choose a photo from a Gallery control, which displays all the images available on the SD card, or simulated SD card on the emulator, by accessing the MediaStore content provider (left).
- On the Pet Listing screen, each picture is displayed in the ListView control (right), again using the MediaStore content provider to access specific images.
- On the Pet Listing screen, each item in the ListView (right) is a custom layout. The new PetTracker3 sample application provides two methods to achieve this: by inflating a custom layout XML file, and by generating the layout programmatically.
- Internally,we extend BaseAdapter on two different occasions to successfully bind pet data to the ListView and Gallery with our own custom requirements.

- Finally,we provide custom implementations of the methods for Simple CursorAdapter .CursorToString Converter and FilterQuery Provider to allow the AutoComplete TextView to bind directly to the internal SQLite database table called pet_types (middle), and change the AutoComplete TextView behavior to match all substrings, not only the beginning of the word. Although we won't go into detail about this in the subsequent text, check out the sample code for more information on the specific details of implementation.

First, we need to decide where we are going to get our photos. We can take pictures with the built-in camera and access those, but for simplicity's sake with the emulator (which can only take "fake pictures"), it is easier if we download those cute, fuzzy pictures from the browser onto the SD card and access them that way.

Locating Content on the Android System Using URIs

Most access to content providers comes in the form of queries: a list of contacts, a list of bookmarks, a list of calls, a list of pictures, and a list of audio files. Applications make these requests much as they would access a database, and they get the same type of structured results. The results of a query are often iterated through using a cursor. However, instead of crafting queries, we use URIs.

You can think of a URI as an "address" to the location where content exists. URI addresses are hierarchical. Most content providers, such as the Contacts and the Media Store, have URI addresses predefined. For example, to access images the External Media Device (also known as the SD card), we use the following URI defined in the Media Store.Images.Media class:

```
Uri mMedia = Media.EXTERNAL_CONTENT_URI;
```

Retrieving Content Provider Data with managedQuery()

We can query the Media Store content provider using the URI much like we would query a database. We now use the managedQuery() method to return a managed Cursor containing all image media available on the SD card.

```
String[] projection = new String[] { Media._ID, Media.TITLE };
Uri mMedia = Media.EXTERNAL_CONTENT_URI;
Cursor mCursorImages = managedQuery(mMedia, projection, null, null,
Media.DATE_TAKEN + " ASC"); // Order-by clause.
```

We have retrieved the records for each piece of media available on the SD card.

Now we have this Cursor, but we still have some legwork to get our Gallery widget to display the individual images.

Data-Binding to the Gallery Control

We need to extend the BaseAdapter class for a new type of data adapter called ImageUriAdapter to map the URI data we retrieved to the Gallery widget. Our custom Image UriAdapter maps the Cursor results to an array of GalleryRecord objects, which correspond to the child items within the Gallery widget. Although the code for the ImageUriAdapter is too long to show here, we go over some of the methods you must implement for the adapter to work properly.

- The ImageUriAdapter() constructor is responsible for mapping the Cursor to an array of GalleryRecord objects, which encapsulate the base URI and the individual image's id. The image id is tacked on to the end of the URI, resulting in a fully qualified URI for the individual image.
- The getItem() and getItemId() methods return the unique identifier for the specific image. This is the value we require when the user clicks on a

specific image within the Gallery. We save this information in our database so that we know which image corresponds to which pet.

- The getView() method returns the custom View widget that corresponds to each child View within the Gallery. In this case, we return an ImageView with the corresponding image. We set each view's Tag property to the associated GalleryRecord object, which includes all our Cursor information we mapped for that record. This is a nifty trick for storing extra information with widgets for later use.

After all this magic has been implemented, we can set our newly defined custom adapter to the adapter used by the Gallery with our new Cursor.

```
ImageUriAdapter iAdapter = new ImageUriAdapter(this,
   mCursorImages, mMedia);
final Gallery pictureGal = (Gallery)fndViewById(R.id.GalleryOfPics);
pictureGal.setAdapter(iAdapter);
```

Retrieving Gallery Images and Saving Them in the Database

Notice that we added two new columns to our SQLite database: the base URI for the image and the individual image id, which is the unique identifier tacked to the end of the URI. We do not save the image itself in the database, only the URI information to retrieve it.

When the user presses the Save button on the Pet Entry screen, we examine the Gallery item selected and extract the information we require from the Tag property of the selected View, like this:

```
final Gallery gall = (Gallery) findViewById(R.id.GalleryOfPics);
ImageView selectedImageView = (ImageView) gall.getSelectedView();
GalleryRecord galleryItem;
if ( selectedImageView != null) {
      galleryItem = (GalleryRecord)selectedImageView.getTag();
      long imageId = galleryItem.getImageId();
      String strImageUriPathString = galleryItem.getImageUriPath();
}
```

We can then save our Pet Record as we have before.

Displaying Images Retrieved from the SD Card Using URIs

Now that our Pet Entry form is saved properly, we must turn our attention to the Pet Listing screen. Our ListView is getting more complicated; each item needs to contain an ImageView and two TextView widgets for the pet name and species. We begin by defining a custom layout template for each ListView item called pet_item.xml. This should be familiar; it contains an ImageView and two TextView objects.

We want to make sure this implementation is scalable, in case we want to add new features to individual ListView items in the future. So instead of taking shortcuts and using standard adapters and built-in Android layout templates, we implement another custom adapter called PetListAdapter.

The PetListAdapter is similar to the ImageUriAdapter we previously implemented for the Gallery widget. This time, instead of Gallery child items, we work with the ListView child records, which correspond to each pet. Again, the constructor maps the Cursor data to an array of PetRecord objects.

The getView() method of the PetListAdapter is where the magic occurs. Here we use a LayoutInflater to inflate our custom layout file called pet_item.xml for each ListView item. Again we use the Tag property of the view to store any information about the record that we might use later. It is here that we use the URI information we stored in our database to rebuild the fully qualified image URI using the Uri.parse() and ContentUris.withAppendedId() utility methods and assign this URI to the ImageView widget using the setImageURI() method. Now that we've set up everything,we assign the PetListAdapter to our ListView:

```
String asColumnsToReturn[] = {
Pets.PETS_TABLE_NAME + "." + Pets.PET_NAME,
Pets.PETS_TABLE_NAME + "." + Pets.PET_IMAGE_URI,
Pets.PETS_TABLE_NAME + "." + Pets._ID,
Pets.PETS_TABLE_NAME + "." + Pets.PET_IMAGE_ID,
PetType.PETTYPE_TABLE_NAME + "." + PetType.PET_TYPE_NAME };
mCursor = queryBuilder.query(mDB, asColumnsToReturn, null, null,
null, null, Pets.DEFAULT_SORT_ORDER);
startManagingCursor(mCursor);
SetListAdapter adapter = new PetListAdapter(this, mCursor);
ListView av = (ListView) findViewById(R.id.petList);
av.setAdapter(adapter);
```

That's about it. Note that you can also create the ListView item layout programmatically (see the PetListItemView class and the PetListAdapter.getView() method comments for more information).

Now you've seen how to leverage a content provider to make your application more robust, but this example has scratched only the surface of how powerful content providers can be.

Acting as a Content Provider

Do you have data in your application? Can another application do something interesting with that data? To share the information within your application with other applications, you need to make the application a content provider by providing the standardized content provider interface for other applications; then you must register your application as a content provider within the Android manifest file. The most straightforward way to make an application a content provider is to store the information you want to share in a SQLite database.

One example is a content provider for GPS track points. This content provider enables users of it to query for points and store points. The data for each point contains a time stamp, the latitude and longitude, and the elevation.

Implementing a Content Provider Interface

Implementing a content provider interface is relatively straightforward. The following code shows the basic interface that an application needs to implement to become a content provider, requiring implementations of five important methods:

```
public class TrackPointProvider extends ContentProvider {
public int delete(Uri uri,
String selection, String[] selectionArgs) {
return 0;
}
public String getType(Uri uri) {
return null;
}
public Uri insert(Uri uri, ContentValues values) {
return null;
}
public boolean onCreate() {
return false;
}
public Cursor query(Uri uri, String[] projection,
String selection, String[] selectionArgs, String sortOrder) {
return null;
}
public int update(Uri uri, ContentValues values,
String selection, String[] selectionArgs) {
return 0;
}
}
```

Defining the Data URI

The provider application needs to define a base URI that other applications will use to access this content provider. This must be in the form of a public static final Uri named CONTENT_URI, and it must start with content://.The URI must be unique. The best practice for this naming is to use the fully qualified class name of the content provider. Here, we have created a URI name for our GPS track point provider book example:

```
public static final Uri CONTENT_URI =
 Uri.parse("content://com.androidbook.TrackPointProvider");
```

Defining Data Columns

The user of the content provider needs to know what columns the content provider has available to it. In this case, the columns used are timestamp, latitude and longitude, and the elevation. We also include a column for the record number, which is called _id.

```
public final static String _ID = "_id";
public final static String TIMESTAMP = "timestamp";
public final static String LATITUDE = "latitude";
public final static String LONGITUDE = "longitude";
public final static String ELEVATION = "elevation";
```

Users of the content provider use these same strings. A content provider for data such as this often stores the data within a SQLite database. If this is the case, matching these columns' names to the database column names simplifies the code.

Implementing Important Content Provider Methods

This section shows example implementations of each of the methods that are used by the system to call this content provider when another application wants to use it. The system, in this case, is the ContentResolver interface that was used indirectly in the previous section when built-in content providers were used.

Some of these methods can make use of a helper class provided by the Android SDK, UriMatcher, which is used to match incoming Uri values to patterns that help speed up development.The use of UriMatcher is described and then used in the implementation of these methods.

Implementing the query() Method

Let’s start with a sample query implementation. Any query implementation needs to return a Cursor object. One convenient way to get a Cursor object is to return the Cursor from the underlying SQLite database that many content providers use. In fact, the interface to ContentProvider.query() is compatible with the SQL ite Query Builder .query() call. This example uses it to quickly build the query and return a Cursor object.

```
public Cursor query(Uri uri, String[] projection,
String selection, String[] selectionArgs,
String sortOrder) {
SQLiteQueryBuilder qBuilder = new SQLiteQueryBuilder();
qBuilder.setTables(TrackPointDatabase.TRACKPOINTS_TABLE);
if ((sURIMatcher.match(uri)) == TRACKPOINT_ID) {
qBuilder.appendWhere("_id=" + uri.getLastPathSegment());
}
Cursor resultCursor = qBuilder.query(mDB.getReadableDatabase(),
 projection, selection, selectionArgs, null, null, sortOrder, null);
resultCursor.setNotificationUri(getContext().getContentResolver(),
 uri);
return resultCursor;
}
```

First, the code gets an instance of a SQLiteQueryBuilder object, which builds up a query with some method calls. Then, the setTables() method configures which table in the database is used. The UriMatcher class checks to see which specific rows are requested. UriMatcher is discussed in greater detail later.

Next, the actual query is called. The content provider query has fewer specifications than the SQLite query, so the parameters are passed through and the rest is ignored. The instance of the SQLite database is read-only. Because this is only a query for data, it’s acceptable.

Finally, the Cursor needs to know if the source data has changed. This is done by a call to the setNotificationUri() method telling it which URI to watch for data changes. The call to the application’s query() method might be called from multiple threads, as it calls to update(), so it’s possible the data can change after the Cursor is returned. Doing this keeps the data synchronized.

Exploring the UriMatcher Class

The UriMatcher class is a helper class for pattern matching on the URIs that are passed to this content provider. It is used frequently in the implementations of the content provider functions that must be implemented. Here is the UriMatcher used in these sample implementations:

```
public static final String AUTHORITY =
“com.androidbook.TrackPointProvider”
private static final int TRACKPOINTS = 1;
private static final int TRACKPOINT_ID = 10;
private static final UriMatcher sURIMatcher =
new UriMatcher(UriMatcher.NO_MATCH);
static {
sURIMatcher.addURI(AUTHORITY, “points”, TRACKPOINTS);
sURIMatcher.addURI(AUTHORITY, “points/#”, TRACKPOINT_ID);
}
```

First, arbitrary numeric values are defined to identify each different pattern. Next, a static UriMatcher instance is created for use. The code parameter that the constructor wants is merely the value to return when there is no match. A value for this is provided for use within the UriMatcher class itself.

Next, the URI values are added to the matcher with their corresponding identifiers. The URIs are broken up in to the authority portion, defined in AUTHORITY, and the path portion, which is passed in as a literal string. The path can contain patterns, such as the “#” symbol to indicate a number. The “*” symbol is used as a wildcard to match anything.

Implementing the insert() Method

Theinsert() method is used for adding data to the content provider. Here is a sample implementation of the insert() method:

```
public Uri insert(Uri uri, ContentValues values) {
int match = sURIMatcher.match(uri);
if (match != TRACKPOINTS) {
throw new IllegalArgumentException(
“Unknown or Invalid URI “ + uri);
}
SQLiteDatabase  sqlDB = mDB.getWritableDatabase();
long newID = sqlDB.
insert(TrackPointDatabase.TRACKPOINTS_TABLE, null, values);
if (newID > 0){
Uri newUri = ContentUris.withAppendedId(uri, newID);
getContext()
.getContentResolver().notifyChange(newUri, null);
return newUri;
}
throw new SQLException(“Failed to insert row into “ + uri);
}
```

The Uri is first validated to make sure it’s one where inserting makes sense. A Uri targeting a particular row would not, for instance. Next, a writeable database object instance is retrieved. Using this, the database insert() method is called on the table defined by the incoming Uri and with the values passed in. At this point, no error checking is performed on the values. Instead, the underlying database implementation throws exceptions that can be handled by the user of the content provider.

If the insert was successful, a Uri is created for notifying the system of a change to the underlying data via a call to the notifyChange() method of the Content Resolver. Otherwise, an exception is thrown.

Implementing the update() Method

The update() method is used to modify an existing row of data. It has elements similar to the insert() and query() methods. The update is applied to a particular selection defined by the incoming Uri.

```
public int update(Uri uri, ContentValues values,
String selection, String[] selectionArgs) {
     SQLiteDatabase sqlDB = mDB.getWritableDatabase();
     int match = sURIMatcher.match(uri);
     int rowsAffected;
     switch (match) {
            case TRACKPOINTS:
            rowsAffected = sqlDB.update(
            TrackPointDatabase.TRACKPOINTS_TABLE,
            values, selection, selectionArgs);
            break;
            case TRACKPOINT_ID:
            String id = uri.getLastPathSegment();
            if (TextUtils.isEmpty(selection)) {
            rowsAffected = sqlDB.update(
            TrackPointDatabase.TRACKPOINTS_TABLE,
            values, _ID + “=” +  id, null);
            } else {
            rowsAffected = sqlDB.update(
            TrackPointDatabase.TRACKPOINTS_TABLE,
            values, selection + “ and “ + _ID + “=”
            + id, selectionArgs);
     }
     break;
     default:
     throw new IllegalArgumentException(
            “Unknown or Invalid  URI “ + uri);
     }
     getContext().getContentResolver().notifyChange(uri, null);
     return rowsAffected;
}
```

In this block of code, a writable SQLiteDatabase instance is retrieved and the Uri type the user passed in is determined with a call to the match() method of the UriMatcher. No checking of values or parameters is performed here. However, to block updates to a specific Uri, such as a Uri affecting multiple rows or a match on TRACKPOINT_ID, java.lang.UnsupportedOperationException can be thrown to indicate this. In this example, though, trust is placed in the user of this content provider.

After calling the appropriate update() method, the system is notified of the change to the URI with a call to the notifyChange() method. This tells any observers of the URI that data has possibly changed. Finally, the affected number of rows is returned, which is information conveniently returned from the call to the update() method.

Implementing the delete() Method

Now it’s time to clean up the database. The following is a sample implementation of the delete() method. It doesn’t check to see if the user might be deleting more data than they should. You also notice that this is similar to the update() method.

```
public int delete(Uri uri, String selection, String[] selectionArgs)
{
     int match = sURIMatcher.match(uri);
     SQLiteDatabase sqlDB = mDB.getWritableDatabase();
     int rowsAffected = 0;
     switch (match) {
     case TRACKPOINTS:
     rowsAffected = sqlDB.delete(
     TrackPointDatabase.TRACKPOINTS_TABLE,
     selection, selectionArgs);
     break;
     case TRACKPOINT_ID:
     String id = uri.getLastPathSegment();
     if (TextUtils.isEmpty(selection)) {
     rowsAffected =
     sqlDB.delete(TrackPointDatabase.TRACKPOINTS_TABLE,
     _ID+"="+id, null);
     } else {
     rowsAffected =
     sqlDB.delete(TrackPointDatabase.TRACKPOINTS_TABLE,
     selection + " and " +_ID+"="+id, selectionArgs);
  }
     break;
     default:
     throw new IllegalArgumentException(
     "Unknown or Invalid URI " + uri);
  }
     getContext().getContentResolver().notifyChange(uri,  null);
     return rowsAffected;
}
```

Again, a writable database instance is retrieved and the Uri type is determined using the match method of UriMatcher. If the result is a directory Uri, the delete is called with the selection the user passed in. However, if the result is a specific row, the row index is used to further limit the delete, with or without the selection. Allowing this without a specific selection enables deletion of a specified identifier without having to also know exactly where it came from.

As before, the system is then notified of this change with a call to the notifyChange() method of ContentResolver. Also as before, the number of affect rows is returned, which we stored after the call to the delete() method.

Implementing the getType() Method

The last method to implement is the getType() method. The purpose of this method is to return the MIME type for a particular Uri that is passed in. It does not need to return MIME types for specific columns of data.

```
public static final String CONTENT_ITEM_TYPE =
ContentResolver.CURSOR_ITEM_BASE_TYPE  +  “/track-points”;
public static final String CONTENT_TYPE =
      ContentResolver.CURSOR_DIR_BASE_TYPE  +  “/track-points”;
public String getType(Uri uri) {
     int matchType = sURIMatcher.match(uri);
switch (matchType) {
     case TRACKPOINTS:
     return CONTENT_TYPE;
     case TRACKPOINT_ID:
     return CONTENT_ITEM_TYPE;
     default:
     throw new
     IllegalArgumentException(“Unknown or Invalid URI “  + uri);
  }
}
```

To start, a couple of MIME types are defined. The Android SDK provides some guideline values for single items and directories of items, which are used here. The corresponding string for each is vnd.android.cursor.item and vnd.android.cursor.dir, respectively. Finally, the match() method is used to determine the type of the provided Uri so that the appropriate MIME type can bereturned.

Updating the Manifest File

Finally, you need to update your application’s AndroidManifest.xml file so that it reflects that a content provider interface is exposed to the rest of the system. Here, the class name and the authorities, or what might considered the domain of the content:// URI, need to be set. For instance, content: //com .android book. Track Point Provider is the base URI used in this content provider example, which means the authority is com.androidbook.TrackPointProvider. The following XML shows an example of this:

```
<provider
android:authorities=”com.androidbook.gpx.TrackPointProvider”
android:multiprocess=”true”
android:name=”com.androidbook.gpx.TrackPointProvider”
</provider>
```

The value of multiprocess is set to true because the data does not need to be synchronized between multiple running versions of this content provider. It’s possible that two or more applications might access a content provider at the same time, so proper synchronization might be necessary.

Working with Live Folders

A LiveFolder (android.provider.LiveFolders) is a powerful feature that complements the content provider interface. A LiveFolder is a special folder containing content generated by a content provider. For example, a user might want to create a LiveFolder with favorite contacts (“Fave Five”), most frequently viewed emails in a custom email application, or high-priority tasks in a task management application.

When the user chooses to create a LiveFolder, the Android system provides a list of all activities that respond to the ACTION_CREATE_LIVE_FOLDER Intent. If the user chooses your Activity, that Activity creates the LiveFolder and passes it back to the system using the setResult() method.

The LiveFolder consists of the following components:

- Folder name
- Folder icon
- Display mode (grid or list)
- Content provider URI for the folder contents

The first task when enabling a content provider to serve up data to a LiveFolder is to provide an <intent-filter> for an Activity that handles enabling the LiveFolder.This is done within the AndroidManifest.xml file as follows:

```
<intent-filter>
<action android:name=
“android.intent.action.CREATE_LIVE_FOLDER” />
<category
android:name=”android.intent.category.DEFAULT” />
</intent-filter>
```

Next, this action needs to be handled within the onCreate() method of the Activity it has been defined for. Within the preceding provider example, you can place the following code to handle this action:

```
super.onCreate(savedInstanceState);
final Intent intent = getIntent();
final String action = intent.getAction();
if (LiveFolders.ACTION_CREATE_LIVE_FOLDER.equals(action)) {
final Intent resultIntent = new Intent();
resultIntent.setData(TrackPointProvider.LIVE_URI);
resultIntent.putExtra(
LiveFolders.EXTRA_LIVE_FOLDER_NAME, “GPX Sample”);
resultIntent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_ICON,
Intent.ShortcutIconResource.fromContext(
this, R.drawable.icon));
resultIntent.putExtra(LiveFolders.EXTRA_LIVE_FOLDER_DISPLAY_MODE,
LiveFolders.DISPLAY_MODE_LIST);
setResult(RESULT_OK, resultIntent);
} // ... rest of onCreate()
```

This defines the core components of the LiveFolder: its name, icon, display mode, and Uri. The Uri is not the same as one that already existed because it needs certain specific fields to work properly. This leads directly to the next task: modifying the content provider to prepare it for serving up data to the LiveFolder.

First, you define a new Uri. In this case, you add ”/live” to the end of the existing CONTENT_URI. For example:

```
public static final Uri LIVE_URI = Uri.parse(“content://”
+  AUTHORITY + “/” + TrackPointDatabase.TRACKPOINTS_TABLE
+ “/live”);
```

You add this new Uri pattern the UriMatcher. Next, modify the query() implementation to recognize this new Uri and add a projection, which is defined next:

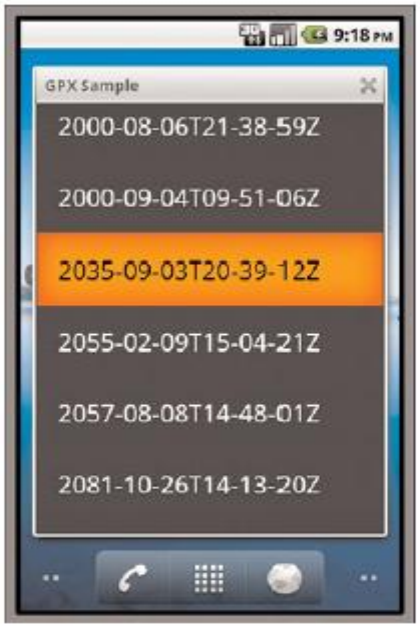
```
switch (sURIMatcher.match(uri)) {
     case TRACKPOINT_ID:
     qBuilder.appendWhere(“_id=” + uri.getLastPathSegment());
     break;
     case TRACKPOINTS_LIVE:
     qBuilder.setProjectionMap(
     TRACKPOINTS_LIVE_FOLDER_PROJECTION_MAP);
     break;
     // ... other cases
     }
Cursor c = qBuilder.query( // ...
```

The projection is critical for a working LiveFolder provider. There are two mandatory fields that must be in the resulting Cursor: LiveFolder._ID and Live Folder .NAME. In addition to these, other fields, such as LiveFolder. DESCRIPTION, are available to modify the look and behavior of the view. In this example, we use TIMESTAMP for the name, as shown here in the following projection implementation:

```
private static final HashMap<String,String>
TRACKPOINTS_LIVE_FOLDER_PROJECTION_MAP; static {
TRACKPOINTS_LIVE_FOLDER_PROJECTION_MAP  =
new HashMap<String,String>();
TRACKPOINTS_LIVE_FOLDER_PROJECTION_MAP.put(
LiveFolders._ID, _ID + " as " + LiveFolders._ID);
TRACKPOINTS_LIVE_FOLDER_PROJECTION_MAP.put(
LiveFolders.NAME, TIMESTAMP + " as " + LiveFolders.NAME);
}
```

After this is done, the LiveFolder should be, well, live. In this example, only a list of dates is shown, as in Figure.

Sample LiveFolder list with dates.



Using Android Networking APIs

RKUNIVERSITY™

By : Ketan Bhimani

Using Android Networking APIs

Applications written with networking components are far more dynamic and content rich than those that are not. Applications leverage the network for a variety of reasons: to deliver fresh and updated content, to enable social networking features of an otherwise standalone application, to offload heavy processing to high-powered servers, and to enable data storage beyond what the user can achieve on the device.

Those accustomed to Java networking will find the java.net package familiar. There are also some helpful Android utility classes for various types of network operations and protocols. This chapter focuses on Hypertext Transfer Protocol (HTTP), the most common protocol for networked mobile applications.

Understanding Mobile Networking Fundamentals

Networking on the Android platform is standardized, using a combination of powerful yet familiar technologies and libraries such as java.net. Network implementation is generally straightforward, but mobile application developers need to plan for less stable connectivity than one might expect in a home or office network setting— connectivity depends on the location of the users and their devices. Users demand stable, responsive applications. This means that you must take extra care when designing network-enabled applications. Luckily, the Android SDK provides a number of tools and classes for ensuring just that.

Accessing the Internet (HTTP)

The most common way to transfer data to and from the network is to use HTTP. You can use HTTP to encapsulate almost any type of data and to secure the data with Secure Sockets Layer (SSL), which can be important when you transmit data that falls under privacy requirements. Also, most common ports used by HTTP are typically open from the phone networks.

Reading Data from the Web

Reading data from the Web can be extremely simple. For example, if all you need to do is read some data from a website and you have the web address of that data, you can leverage the URL class (available as part of the java.net package) to read a fixed amount of text from a file on a web server, like this:

```
import java.io.InputStream;
import java.net.URL;
  // ...
URL text = new URL(
"http://api.flickr.com/services/feeds/photos_public.gne"  +
"?id=26648248@N04&lang=en-us&format=atom");
InputStream isText = text.openStream();
byte[] bText = new byte[250];
int readSize = isText.read(bText);
Log.i("Net", "readSize = " + readSize);
Log.i("Net", "bText = "+ new String(bText));
isText.close();
```

First, a new URL object is created with the URL to the data we want to read. A stream is then opened to the URL resource. From there, we read the data and close the InputStream. Reading data from a server can be that simple.

However, remember that because we work with a network resource, errors can be more common. Our phone might not have network coverage; the server might be down for maintenance or disappear entirely; the URL might be invalid; and network users might experience long waits and timeouts.

This method might work in some instances—for example, when your application has lightweight, noncritical network features—but it's not particularly elegant. In many cases, you might want to know more about the data before reading from it from the URL. For instance, you might want to know how big it is.

Finally, for networking to work in any Android application, permission is required. Your application needs to have the following statement in its AndroidManifest.xml file:

```
<uses-permission
android:name=”android.permission.INTERNET”/>
```

Using HttpURLConnection

We can use the HttpURLConnection object to do a little reconnaissance on our URL before we transfer too much data. HttpURLConnection retrieves some information about the resource referenced by the URL object, including HTTP status and header information.

Some of the information you can retrieve from the HttpURLConnection includes the length of the content, content type, and date-time information so that you can check to see if the data changed since the last time you accessed the URL.

Here is a short example of how to use HttpURLConnection to query the same URL previously used:

```
import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;
// ...
URL text = new URL(
"http://api.flickr.com/services/feeds/photos_public.gne
?id=26648248@N04&lang=en-us&format=atom");
HttpURLConnection http =
(HttpURLConnection)text.openConnection();
Log.i("Net", "length = " + http.getContentLength());
Log.i("Net", "respCode = " + http.getResponseCode());
Log.i("Net", "contentType = "+ http.getContentType());
Log.i("Net", "content = "+http.getContent());
```

The log lines demonstrate a few useful methods with the HttpURLConnection class. If the URL content is deemed appropriate, you can then call http.getInputStream() to get the same InputStream object as before. From there, reading from the network resource is the same, but more is known about the resource.

Parsing XML from the Network

A large portion of data transmitted between network resources is stored in a structured fashion in Extensible Markup Language (XML). In particular, RSS feeds are provided in a standardized XML format, and many web services provide data using these feeds.

Android SDK provides a variety of XML utilities. We dabble with the XML Pull Parser in Chapter "Managing Application Resources."We also cover the various SAX and DOM support available in Chapter 10,"Using Android Data and Storage APIs."

Parsing XML from the network is similar to parsing an XML resource file or a raw file on the file system. Android provides a fast and efficient XML Pull Parser, which is a parser of choice for networked applications.

The following code demonstrates how to use the XML Pull Parser to read an XML file from flickr.com and extract specific data from within it. A TextView called status is assigned before this block of code is executed and displays the status of the parsing operation.

```
import java.net.URL;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserFactory;
// ...
URL text = new URL(
     “http://api.flickr.com/services/feeds/photos_public.gne
     ?id=26648248@N04&lang=en-us&format=atom”);
     XmlPullParserFactory parserCreator =
     XmlPullParserFactory.newInstance();
     XmlPullParser parser = parserCreator.newPullParser();
     parser.setInput(text.openStream(), null);
     status.setText(“Parsing...”);
     int parserEvent = parser.getEventType();
     while (parserEvent != XmlPullParser.END_DOCUMENT) {
     switch(parserEvent) {case XmlPullParser.START_TAG:
     String tag = parser.getName();
     if (tag.compareTo(“link”) == 0) {
     String relType = parser.getAttributeValue(null, “rel”);
     if (relType.compareTo(“enclosure”) == 0 ) {
     String encType = parser.getAttributeValue(null, “type”);
     if (encType.startsWith(“image/”)) {
     String imageSrc = parser.getAttributeValue(null,  “href”);
     Log.i(“Net”, “image source = “ +  imageSrc);
     }
  }
}
break;
}
parserEvent =  parser.next();
}
status.setText(“Done...”);
```

After the URL is created, the next step is to retrieve an XmlPullParser instance from the XmlPullParserFactory. A Pull Parser has a main method that returns the next event. The events returned by a Pull Parser are similar to methods used in the implementation of a SAX parser handler class. Instead, though, the code is handled iteratively. This method is more efficient for mobile use.

In this example, the only event that we check for is the START_TAG event, signifying the beginning of an XML tag. Attribute values are queried and compared. This example looks specifically for image URLs within the XML from a flickr feed query. When found, a log entry is made.

You can check for the following XML Pull Parser events:

- START_TAG: Returned when a new tag is found (that is, <tag>)
- TEXT: Returned when text is found (that is, <tag>text</tag> where text has been found)
- END_TAG: Returned when the end of tag is found (that is, </tag>)
- END_DOCUMENT: Returned when the end of the XML file is reached

Additionally, the parser can be set to validate the input. Typically, parsing without validation is used when under constrained memory environments, such as a mobile environment. Compliant, nonvalidating parsing is the default for this XML Pull Parser.

Processing Asynchronously

Users demand responsive applications, so time-intensive operations such as networking should not block the main UI thread. The style of networking presented so far causes the UI thread it runs on to block until the operation finishes. For small tasks, this might be acceptable. However, when timeouts, large amounts of data, or additional processing, such as parsing XML, is added into the mix, you should move these time-intensive operations off of the main UI thread.

Offloading intensive operations such as networking provides a smoother, more stable experience to the user. The Android SDK provides two easy ways to manage offload processing from the main UI thread: the AsyncTask class and the standard Java Thread class.

The AsyncTask class is a special class for Android development that encapsulates background processing and helps facilitate communication to the UI thread while managing the lifecycle of the background task within the context of the activity lifecycle. Developers can also construct their own threading solutions using the standard Java methods and classes—but they are then responsible for managing the entire thread lifecycle as well.

Working with AsyncTask

AsyncTask is an abstract helper class for managing background operations that eventually post back to the UI thread. It creates a simpler interface for asynchronous operations than manually creating a Java Thread class.

Instead of creating threads for background processing and using messages and message handlers for updating the UI, you can create a subclass of AsyncTask and implement the appropriate event methods. The onPreExecute() method runs on the UI thread before background processing begins. The doInBackground() method handles background processing, whereas publishProgress() informs the UI thread periodically about the background processing progress. When the background processing finishes, the onPostExecute() method runs on the UI thread to give a final update.

The following code demonstrates an example implementation of AsyncTask to perform the same functionality as the code for the Thread:

```
private class ImageLoader extends
AsyncTask<URL, String, String> {
     @Override
     protected String doInBackground(URL... params) {
            // just one param
            try {
                   URL text = params[0];
                   //  ... parsing code {
            publishProgress("imgCount = " + curImageCount);
            //  ... end parsing code }
            }
            catch (Exception e ) {
            Log.e("Net", "Failed in parsing XML", e);
            return "Finished with failure.";
            }
            return "Done...";
     }
protected void onCancelled() {
     Log.e("Net", "Async task Cancelled");
}
protected void onPostExecute(String result) {
     mStatus.setText(result);
}
protected void onPreExecute() {
     mStatus.setText("About to load URL");
}
protected void onProgressUpdate(String... values) {
     // just one value, please
      mStatus.setText(values[0]);
}}
```

When launched with the AsyncTask.execute() method, doInBackground() runs in a background thread while the other methods run on the UI thread. There is no need to manage a Handler or post a Runnable object to it. This simplifies coding and debugging.

Using Threads for Network Calls

The following code demonstrates how to launch a new thread that connects to a remote server, retrieves and parses some XML, and posts a response back to the UI thread to change a TextView:

```
import java.net.URL;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserFactory;
// ...
new Thread() {
public void run() {
 try {
 URL text = new URL(
 "http://api.flickr.com/services/feeds/photos_public.gne?
 id=26648248@N04&lang=en-us&format=atom");
 XmlPullParserFactory parserCreator =
 XmlPullParserFactory.newInstance();
 XmlPullParser parser =  parserCreator.newPullParser();
 parser.setInput(text.openStream(), null);
 mHandler.post(new Runnable() {
      public void run() {
             status.setText("Parsing...");
      }
 });
 int parserEvent = parser.getEventType();
 while (parserEvent !=  XmlPullParser.END_DOCUMENT){
      // Parsing code here ...
      parserEvent = parser.next();
 }
 mHandler.post(new Runnable() {
      public void run(){
             status.setText("Done...");
      }
 });
 } catch (Exception e){
      Log.e("Net", "Error in network call", e);
 }
 }
 }.start();
```

For this example, an anonymous Thread object will do. We create it and call its start() method immediately. However, now that the code runs on a separate thread, the user interface updates must be posted back to the main thread. This is done by using a Handler object on the main thread and creating Runnable objects that execute to call setText() on the TextView widget named status.

The rest of the code remains the same as in the previous examples. Executing both the parsing code and the networking code on a separate thread allows the user interface to continue to behave in a responsive fashion while the network and parsing operations are done behind the scenes, resulting in a smooth and friendly user experience. This also allows for handling of interim actions by the user, such as canceling the transfer

You can Accessing the Internet (HTTP) 295 accomplish this by implementing the Thread to listen for certain events and check for certain flags.

Displaying Images from a Network Resource

Now that we have covered how you can use a separate thread to parse XML, let's take our example a bit deeper and talk about working with non-primitive data types.

Continuing with the previous example of parsing for image locations from a flickr feed, let's display some images from the feed. The following example reads the image data and displays it on the screen, demonstrating another way you can use network resources:

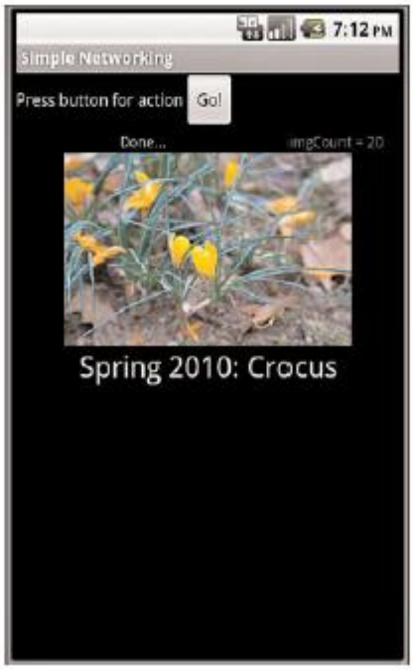
```
import java.io.InputStream;
import java.net.URL;
import org.xmlpull.v1.XmlPullParser;
import org.xmlpull.v1.XmlPullParserFactory;
import android.os.Handler;
// ...
final String imageSrc =  parser.getAttributeValue(null,“href”);
final String currentTitle = new String(title);
imageThread.queueEvent(new Runnable(){
     public void run(){
            InputStream  bmis;
            try{
                   bmis = new URL(imageSrc).openStream();
                   final Drawable image = new BitmapDrawable(
                   BitmapFactory.decodeStream(bmis));
                   mHandler.post(new Runnable(){
                          public void run(){
                          imageSwitcher.setImageDrawable(image);
                          info.setText(currentTitle);
                   }
            });
            } catch (Exception e){
     Log.e(“Net”, “Failed to grab image”, e);
     }
}
});
```

You can find this block of code within the parser thread, as previously described. After the image source and title of the image have been determined, a new Runnable object is queued for execution on a separate image handling thread. The thread is merely a queue that receives the anonymous Runnable object created here and executes it at least 10 seconds after the last one, resulting in a slideshow of the images from the feed.

As with the first networking example, a new URL object is created and an InputStream retrieved from it. You need a Drawable object to assign to the ImageSwitcher. Then you use the BitmapFactory.decodeStream() method, which takes an InputStream.

Finally, from this Runnable object, which runs on a separate queuing thread, spacing out image drawing, another anonymous Runnable object posts back to the main thread to actually update the ImageSwitcher with the new image. Figure shows what the screen might look like showing decoding status and displaying the current image.

Screen showing a flickr image and decoding status of feed.



Although all this continues to happen while the feed from flickr is decoded, certain operations are slower than others. For instance, while the image is decoded or drawn on the screen, you can notice a distinct hesitation in the progress of the decoding. This is to be expected on current mobile devices because most have only a single thread of execution available for applications. You need to use careful design to provide a reasonably smooth and responsive experience to the user.

Retrieving Android Network Status

The Android SDK provides utilities for gathering information about the current state of the network. This is useful to determine if a network connection is even available before trying to use a network resource. The ConnectivityManager class provides a number of methods to do this. The following code determines if the mobile (cellular) network is available and connected. In addition, it determines the same for the Wi-Fi network:

```
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
// ...
ConnectivityManager cm = (ConnectivityManager)
getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo ni =  cm.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiAvail = ni.isAvailable();
boolean isWifiConn = ni.isConnected();
ni = cm.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileAvail = ni.isAvailable();
boolean isMobileConn = ni.isConnected();
status.setText("WiFi\nAvail = "+ isWifiAvail + "\nConn = " +
     isWifiConn + "\nMobile\nAvail  = "+ isMobileAvail +
     "\nConn = " + isMobileConn);
```

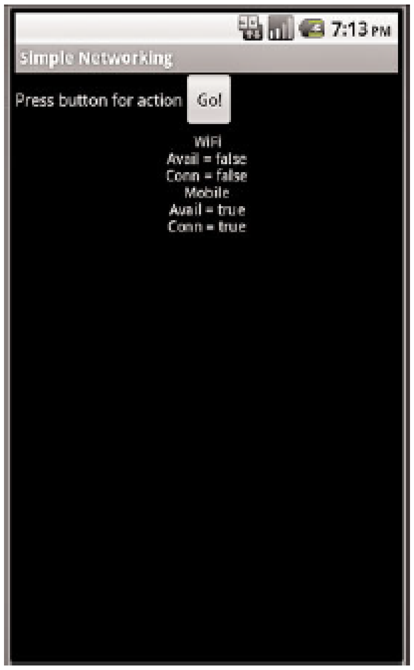
First, an instance of the ConnectivityManager object is retrieved with a call to the getSystemService() method, available as part of your application Context. Then this instance retrieves NetworkInfo objects for both TYPE_WIFI and TYPE_MOBILE (for the cellular network).These objects are queried for their availability but can also be queried at a more detailed status level to learn exactly what state of connection (or disconnection) the network is in. Figure shows the typical output for the emulator in which the mobile network is simulated but Wi-Fi isn't available.

If the network is available, this does not necessarily mean the server that the network resource is on is available. However, a call to the ConnectivityManager method requestRouteToHost() can answer this question. This way, the application can give the user better feedback when there are network problems.

For your application to read the status of the network, it needs explicit permission. The following statement is required to be in its AndroidManifest.xml file:

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

Typical network status of the Android SDK emulator.



```
WIFI
Avail = false
Conn = false
Mobile
Avail = true
Conn = true
```

Using Android Web APIs

Using Android Web APIs

Mobile developers often rely upon web technologies to enrich their applications, provide fresh content, and integrate with popular web services such as social networks. Android application can harness the power of the Internet in a variety of ways, including adding browser functionality to applications using the special WebView control and extending web-based functionality using standard WebKit libraries. Newer Android devices can also run Flash applications. In this chapter, we discuss the web technologies available on the Android platform.

Browsing the Web with WebView

Applications that retrieve and display content from the Web often end up displaying that data on the screen. Instead of customizing various screens with custom controls, Android applications can simply use the WebView control to display web content to the screen. You can think of the WebView control as a browser-like view.

The WebView control uses the WebKit rendering engine to draw HTML content on the screen. This content could be HTML pages on the Web or it can be locally sourced. WebKit is an open source browser engine. You can read more about it on its official website .

Using the WebView control requires the android.permission.INTERNET permission. You can add this permission to your application’s Android manifest file as follows:

```
<uses-permission android:name=”android.permission.INTERNET” />
```

When deciding if the WebView control is right for your application, consider that you can always launch the Browser application using an Intent. When you want the user to have full access to all Browser features, such as bookmarking and browsing, you’re better off launching into the Browser application to a specific website, letting users do their browsing, and having them return to your application when they’re done. You can do this as follows:

```
Uri uriUrl = Uri.parse(“http://androidbook.blogspot.com/”);
Intent launchBrowser = new Intent(Intent.ACTION_VIEW, uriUrl);
startActivity(launchBrowser);
```

Launching the Browser via an Intent does not require any special permissions. This means that your application is not required to have the android .permission .INTERNET permission. In addition, because Android transitions from your application’s current activity to a specific Browser application’s activity, and then returns when the user presses the back key, the experience is nearly as seamless as implementing your own Activity class with an embedded WebView.

Designing a Layout with a WebView Control

The WebView control can be added to a layout resource file like any other view. It can take up the entire screen or just a portion of it. A typical WebView definition in a layout resource might look like this:

```
<WebView
android:id=”@+id/web_holder”
android:layout_height=”wrap_content”
android:layout_width=”fill_parent”
/>
```

Generally speaking, you should give your WebView controls ample room to display text and graphics. Keep this in mind when designing layouts using the WebView control.

Loading Content into a WebView Control

You can load content into a WebView control in a variety of ways. For example, a WebView control can load a specific website or render raw HTML content. Web pages can be stored on a remote web server or stored on the device.

Here is an example of how to use a WebView control to load content from a specific website:

```
final WebView wv = (WebView) findViewById(R.id.web_holder);
wv.loadUrl("http://www.perlgurl.org/");
```

You do not need to add any additional code to load the referenced web page on the screen. Similarly, you could load an HTML file called webby.html stored in the application's assets directory like this:

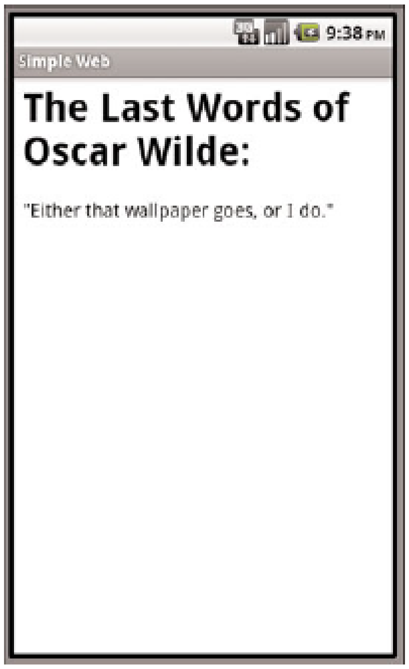
```
     wv.loadUrl("file:///android_asset/webby.html");
```

If, instead, you want to render raw HTML, you can use the loadData() method:

```
String strPageTitle = "The Last Words of Oscar Wilde";
String strPageContent = "<h1>" + strPageTitle +
": </h1>\"Either that wallpaper goes, or I do.\"";
String myHTML = "<html><title>" + strPageTitle
+"</title><body>"+ strPageContent +"</body></html>";
wv.loadData(myHTML, "text/html", "utf-8");
```

The resulting WebView control is shown in Figure.

WebView control used to display HTML.



The Last Words of Oscar Wilde:

"Either that wallpaper goes, or I do."

Unfortunately, not all websites are designed for mobile devices. It can be handy to change the scale of the web content to fit comfortably within the WebView control. You can achieve this by setting the initial scale of the control, like this:

```
wv.setInitialScale(30);
```

The call to the setInitialScale() method scales the view to 30 percent of the original size. For pages that specify absolute sizes, scaling the view is necessary to see the entire page on the screen. Some text might become too small to read, though, so you might need to test and make page design changes (if the web content is under your control) for a good user experience.

Adding Features to the WebView Control

You might have noticed that the WebView control does not have all the features of a full browser. For example, it does not display the title of a webpage or provide buttons for reloading pages. In fact, if the user clicks on a link within the WebView control, that action does not load the new page within the view. Instead, it fires up the Browser application.

By default, all the WebView control does is display the web content provided by the developer using its internal rendering engine,WebKit. You can enhance the WebView control in a variety of ways, though. You can use three classes, in particular, to help modify the behavior of the control: the WebSettings class, the WebViewClient class, and the WebChromeClient class.

Modifying WebView Settings with WebSettings

By default, a WebView control has various default settings: no zoom controls, JavaScript disabled, default font sizes, user-agent string, and so on. You can change the settings of a WebView control using the getSettings() method. The getSettings() method returns a WebSettings object that can be used to configure the desired WebView settings. Some useful settings include

- Enabling and disabling zoom controls using the setSupportZoom() and setBuiltIn ZoomControls() methods
- Enabling and disabling JavaScript using the setJavaScriptEnabled() method
- Enabling and disabling mouseovers using the setLightTouchEnabled() method
- Configuring font families, text sizes, and other display characteristics

You can also use the WebSettings class to configure WebView plug-ins and allow for multiple windows.

Handling WebView Events with WebViewClient

The WebViewClient class enables the application to listen for certain WebView events, such as when a page is loading, when a form is submitted, and when a new URL is about to be loaded. You can also use the WebViewClient class to determine and handle any errors that occur with page loading. You can tie a valid WebViewClient object to a WebView using the setWebViewClient() method.

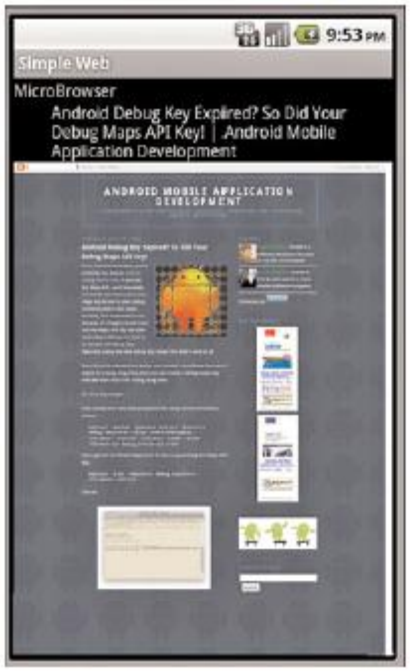
The following is an example of how to use WebViewClient to handle the onPageFinished() method to draw the title of the page on the screen:

```
WebViewClient webClient = new WebViewClient() {
public void onPageFinished(WebView view, String url) {
     super.onPageFinished(view,  url);
     String title = wv.getTitle();
     pageTitle.setText(title);
}};
wv.setWebViewClient(webClient);
```

When the page finishes loading, as indicated by the call to onPageFinished(), a call to the getTitle() method of the WebView object retrieves the title for use. The result of this call is shown in Figure.

WebView control with microbrowser features such as title display.

Adding Browser Chrome with WebChromeClient


9:53 PM
Simple Web
MicroBrowser
Android Debug Key Expired? So Did Your Debug Maps API Key! | Android Mobile Application Development
ANDROID MOBILE APPLICATION DEVELOPMENT

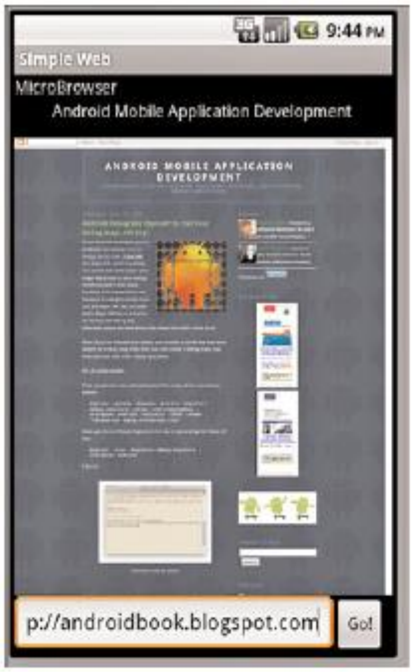
You can use the WebChromeClient class in a similar way to the WebViewClient. However, WebChromeClient is specialized for the sorts of items that will be drawn outside the region in which the web content is drawn, typically known as *browser chrome*. The WebChromeClient class also includes callbacks for certain JavaScript calls, such as onJsBeforeUnload(), to confirm navigation away from a page. A valid WebChromeClient object can be tied to a WebView using the set Web Chrome

Client() method. The following code demonstrates using WebView features to enable interactivity with the user. An EditText and a Button control are added below the WebView control, and a Button handler is implemented as follows:

```
Button go = (Button) findViewById(R.id.go_button);
go.setOnClickListener(new View.OnClickListener() {
     public void onClick(View  v) {
            wv.loadUrl(et.getText().toString());
     }
});
```

Calling the loadUrl() method again, as shown, is all that is needed to cause the WebView control to download another HTML page for display, as shown in Figure. From here, you can build a generic web browser in to any application, but you can apply restrictions so that the user is restricted to browsing relevant materials.

WebView with EditText allowing entry of arbitrary URLs.


9:44 PM
Simple Web
MicroBrowser
Android Mobile Application Development
ANDROID MOBILE APPLICATION DEVELOPMENT
p://androidbook.blogspot.com
Go!

Using WebChromeClient can help add some typical chrome on the screen. For instance, you can use it to listen for changes to the title of the page, various JavaScript dialogs that might be requested, and even for developer-oriented pieces, such as the console messages.

```
WebChromeClient webChrome = new WebChromeClient() {
      @Override
      public void onReceivedTitle(WebView  view, String title) {
              Log.v(DEBUG_TAG, "Got new title");
              super.onReceivedTitle(view, title);
              pageTitle.setText(title);
      }
};
wv.setWebChromeClient(webChrome);
```

Here the default WebChromeClient is overridden to receive changes to the title of the page. This title of the web page is then set to a TextView visible on the screen.

Whether you use WebView to display the main user interface of your application or use it sparingly to draw such things as help pages, there are circumstances where it might be the ideal control for the job to save coding time, especially when compared to a custom screen design. Leveraging the power of the open source engine, WebKit, WebView can provide a powerful, standards-based HTML viewer for applications. Support for WebKit is widespread because it is used in various desktop browsers, including Apple Safari and Google Chrome, a variety of mobile browsers, including those on the Apple iOS, Nokia, Palm WebOS, and BlackBerry handsets, and various other platforms, such as Adobe AIR.

Building Web Extensions Using WebKit

All HTML rendering on the Android platform is done using the WebKit rendering engine. The android.webkit package provides a number of APIs for browsing the Internet using the powerful WebView control. You should be aware of the WebKit interfaces and classes available, as you are likely to need them to enhance the WebView user experience.

These are not classes and interfaces to the Browser app (although you can interact with the Browser data using contact providers). Instead, these are the classes and interfaces that you must use to control the browsing abilities of WebView controls you implement in your applications.

Browsing the WebKit APIs

Some of the most helpful classes of the android.webkit package are

- The CacheManager class gives you some control over cache items of a WebView.
- The ConsoleMessage class can be used to retrieve JavaScript console output from a WebView.
- The CookieManager class is used to set and retrieve user cookies for a WebView.
- The URLUtil class is handy for validating web addresses of different types.
- The WebBackForwardList and WebHistoryItem classes can be used to inspect the web history of the WebView.

Now let's take a quick look at how you might use some of these classes to enhance a WebView.

Extending Web Application Functionality to Android

Let's take some of the WebKit features we have discussed so far in this chapter and work through an example. It is fairly common for mobile developers to design their applications as web applications in order to reach users across a variety of platforms. This minimizes the amount of platform-specific code to develop and maintain. However, on its own, a web application cannot call into native platform code and take advantage of the features that native apps (such as those written in Java for the Android platform) can, such as using a built-in camera or accessing some other underlying Android feature.

Developers can enhance web applications by designing a lightweight shell application in Java and using a WebView control as a portal to the web application content. Two-way communication between the web application and the native Java application is possible through scripting languages such as JavaScript.

Let’s create a simple Android application that illustrates communication between web content and native Android code. This example requires that you understand JavaScript. To create this application, take the following steps:

1. Create a new Android application.
2. Create a layout with a WebView control called html_viewer and a Button control called call_js. Set the onClick attribute of the Button control to a method called setHTMLText.
3. In the onCreate() method of your application activity, retrieve the WebView control using the findViewById() method.
4. Enable JavaScript within the WebView by retrieving its WebSettings and calling the setJavaScriptEnabled() method.
5. Create a WebChromeClient object and implement its onConsoleMessage() method in order to monitor the JavaScript console messages.
6. Add the WebChromeClient object to the WebView using the setWebChromeClient() method.
7. Allow the JavaScript interface to control your application by calling the addJavascriptInterface() method of the WebView control. You will need to define the functionality that you want the JavaScript interface to be able to control and within what namespace the calls will be available. In this case, we allow the JavaScript to initiate Toast messages.

8. Load your content into the WebView control using one of the standard methods, such as the loadUrl() method. In this case, we load an HTML asset we defined within the application package.

If you followed these steps, you should end up with your activity's onCreate() method looking something like this:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    final WebView wv = (WebView) findViewById(R.id.html_viewer);
    WebSettings settings = wv.getSettings();
    settings.setJavaScriptEnabled(true);
    WebChromeClient webChrome = new WebChromeClient() {
            @Override
    public boolean onConsoleMessage(ConsoleMessage consoleMessage)
    {
            Log.v(DEBUG_TAG, consoleMessage.lineNumber()
            + ": " + consoleMessage.message());
            return true;
    }
    };
wv.setWebChromeClient(webChrome);
wv.addJavascriptInterface(new JavaScriptExtensions(), "jse");
wv.loadUrl("file:///android_asset/sample.html");
}
```

A custom WebChromeClient class is set so that any JavaScript console.log messages go out to LogCat output, using a custom debug tag as usual to enable easy tracking of log output specific to the application. Next, a new JavaScript interface is defined with the namespace called jse—the namespace is up to you. To call from JavaScript to this Java class, the JavaScript calls must all start with namespace jse., followed by the appropriate exposed method—for instance, jse.javaMethod().

You can define the JavaScriptExtensions class as a subclass within the activity as a subclass with a single method that can trigger Android Toast messages:

```
class JavaScriptExtensions {
public static final int TOAST_LONG = Toast.LENGTH_LONG;
public static final int TOAST_SHORT = Toast.LENGTH_SHORT;
public void toast(String message, int length) {
Toast.makeText(SimpleWebExtension.this, message, length).show();
}
}
```

The JavaScript code has access to everything in the JavaScriptExtensions class, including the member variables as well as the methods. Return values work as expected from the methods, too. Now switch your attention to defining the web page to load in the WebView control. For this example, simply create a file called sample.html in the /assets directory of the application.

The contents of the sample.html file are shown here:

```
<html>
<head>
<script type="text/javascript">
function doToast() {
jse.toast("'"+document.getElementById('form_text').value  +
     "' -From Java!", jse.TOAST_LONG);
}
function doConsoleLog() {
     console.log("Console logging.");
}
function doAlert() {
     alert("This is an alert.");
}
function doSetFormText(update) {
     document.getElementById('form_text').value =  update;
}
</script>
</head>
<body>
<h2>This is a test.</h2>
<input type="text" id="form_text" value="Enter something here..." />
<input type="button" value="Toast" onclick="doToast();" /><br  />
```

```
<input type=”button” value=”Log” onclick=”doConsoleLog();”  /><br />
<input type=”button” value=”Alert” onclick=”doAlert();” />
</body>
</html>
```

The sample.html file defines four JavaScript functions and displays the form shown within the WebView:

- The doToast() function calls into the Android application using the jse object defined earlier with the call to the addJavaScriptInterface() method. The add JavaScriptInterface() method, for all practical intents and purposes, can be treated literally as the JavaScriptExtensions class as if that class had been written in JavaScript. If the doToast() function had returned a value, we could assign it to a variable here.
- The doConsoleLog() function writes into the JavaScript console log, which is picked up by the onConsoleMessage() callback of the WebChromeClient.
- The doAlert() function illustrates how alerts work within the WebView control by launching a dialog. If you want to override what the alert looks like, you can override the WebChromeClient.onJSAlert() method.
- The doSetFormText() function illustrates how native Java code can communicate back through the JavaScript interface and provide data to the web application.

Finally, to demonstrate making a call from Java back to JavaScript, you need to define the click handler for the Button control within your Activity class. Here, the onClick handler, called setHTMLText(), executes some JavaScript on the currently loaded page by calling a JavaScript function called doSetFormText(), which we defined earlier in the web page. Here is an implementation of the setHTMLText() method:

```
public void setHTMLText(View view) {
     WebView wv = (WebView) findViewById(R.id.html_viewer);
     wv.loadUrl(“javascript:doSetFormText(‘Java->JS  call’);”);
}
```

This method of making a call to the JavaScript on the currently loaded page does not allow for return values. There are ways, however, to structure your design to allow checking of results, generally by treating the call as asynchronous and implementing another method for determining the response.

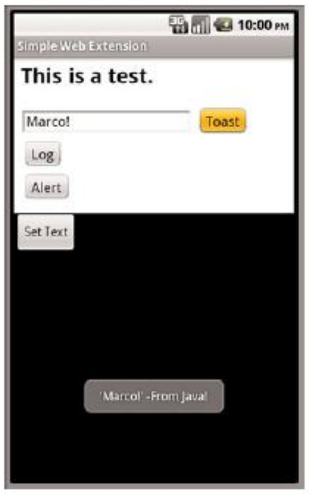
Figure shows how this application might behave on an Android device.

This style of development has been popularized by the open source PhoneGap project, which aims to provide a set of standard JavaScript interfaces to native code across a variety of platforms, including iOS,Android, BlackBerry, Symbian, and Palm.

Working with Flash

For those web developers wanting to bring their Flash applications to mobile,Android is the only smart phone platform currently supporting desktop Flash 10.1 (as opposed to Flash Lite, a common mobile variant of Flash that's very limited). However, there are both benefits and drawbacks to including Flash technology on the platform. Let's look at some of the facts:

- Flash might not be the "future," but it's the "status quo" in some web circles. There are millions of Flash applications and websites out there that can now be accessed from Android devices. This makes users happy, which should make the rest of us happy.



A simple Android application with a JavaScript interface.

- Native Android applications are always going to perform better, use fewer resources (read: drain the battery slower), provide tighter platform integration, have fewer platform prerequisites, and support more Android devices than Flash applications.

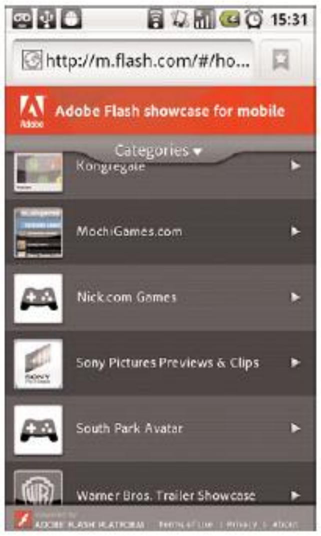
- Deciding to build Flash applications for the Android platform instead of native Java applications is a design decision that should not be taken lightly. There are performance and security tradeoffs as well as limited device support (and no backward compatibility) for Flash.
- You can't expect all Flash applications to just be loaded up work. All the usual mobile constraints and UI paradigms apply. This includes designing around such constraints as a touch interface on a small screen, a relatively slow processor, and interruptions (such as phone calls) being the norm.

Still, there are those millions of great Flash applications out there. Let's look at how you can bring these applications to the Android platform.

Enabling Flash Applications

Android devices with Android 2.2 and higher can run Flash applications (currently Flash 10.1). In order to run Flash, the Android device must have Adobe's Flash Player for Android installed.

Users can download the Adobe's Flash Player for Android application from the Android Market. Android handsets might also ship with the Adobe application pre-loaded. Keep in mind that only the faster, more powerful Android devices are likely to run Flash smoothly and provide a positive user experience. After it's installed, the Flash Player for Android application behaves like a typical browser plug-in. Users can enable or disable it, and you can control whether plug-ins are enabled or not within your screens that use the WebView control.


15:31
http://m.flash.com/#/ho...
Adobe Flash showcase for mobile
Categories
Kongregate
MochiGames.com
Nick.com Games
Sony Pictures Previews & Clips
South Park Avatar
Warner Bros. Trailer Showcase

The Nexus One running a Flash application showing many mobile Flash applications available.

Building AIR Applications for Android

Adobe has created tools for developing cross-platform applications using their AIR tool suite in ActionScript 3, which is Adobe's web scripting language for web and Flash applications. The company recently announced Adobe AIR for Android, which enables developers to create AIR applications that can be compiled into native Android APK files that can then be published like any other Android application. Developers use Adobe's Flash Professional CS5 tools with a special extension to develop AIR applications that can be compiled into Android package files and distributed like native Android applications.

Using Android Telephony APIs

RKUNIVERSITY™

By : Ketan Bhimani

Using Android Telephony APIs

Although the Android platform has been designed to run on almost any type of device, the Android devices available on the market are primarily phones. Applications can take advantage of this fact by integrating phone features into their feature set.

This chapter introduces you to the telephony-related APIs available within the Android SDK.

Working with Telephony Utilities

The Android SDK provides a number of useful utilities for applications to integrate phone features available on the device. Generally speaking, developers should consider an Android device first and foremost as a phone. Although these devices might also run applications, phone operations generally take precedence. Your application should not interrupt a phone conversation, for example. To avoid this kind of behavior, your application should know something about what the user is doing, so that it can react differently. For instance, an application might query the state of the phone and determine that the user is talking on the phone and then choose to vibrate instead of play an alarm.

In other cases, applications might need to place a call or send a text message. Phones typically support a Short Message Service (SMS), which is popular for texting (text messaging). Enabling the capability to leverage this feature from an application can enhance the appeal of the application and add features that can't be easily replicated on a desktop environment. Because many Android devices are phones, applications frequently deal with phone numbers and the contacts database; some might want to access the phone dialer to place calls or check phone status information.

Adding telephony features to an application enables a more integrated user experience and enhances the overall value of the application to the users.

Gaining Permission to Access Phone State Information

Let’s begin by looking at how to determine telephony state of the device, including the ability to request the hook state of the phone, information of the phone service, and utilities for handling and verifying phone numbers. The TelephonyManager object within the android.telephony package is a great place to start.

Many of the method calls in this section require explicit permission set with the Android application manifest file. The READ_PHONE_STATE permission is required to retrieve information such as the call state, handset phone number, and device identifiers or serial numbers. The ACCESS_COARSE_LOCATION permission is required for cellular location information. Recall that we cover location-based services in detail in Chapter “Using Location- Based Services (LBS) APIs.”

The following block of XML is typically needed in your application’s AndroidManifest.xml file to access basic phone state information:

```
<uses-permission
android:name=”android.permission.READ_PHONE_STATE” />
```

Requesting Call State

You can use the TelephonyManager object to retrieve the state of the phone and some information about the phone service itself, such as the phone number of the handset.

You can request an instance of TelephonyManager using the getSystemService() method, like this:

```
TelephonyManager telManager = (TelephonyManager)
getSystemService(Context.TELEPHONY_SERVICE);
```

With a valid TelephonyManager instance, an application can now make several queries.One important method is getCallState().This method can determine the voice call status of the handset. The following block of code shows how to query for the call state and all the possible return values:

```
int callStatus = telManager.getCallState();
String callState = null;
switch (callStatus) {
     case TelephonyManager.CALL_STATE_IDLE:
            callState = "Phone is idle.";
            break;
     case TelephonyManager.CALL_STATE_OFFHOOK:
            callState = "Phone is in use.";
            break;
     case TelephonyManager.CALL_STATE_RINGING:
            callState = "Phone is ringing!";
            break;
}
Log.i("telephony", callState);
```

The three call states can be simulated with the emulator through the Dalvik Debug Monitor

Querying for the call state can be useful in certain circumstances. However, listening for changes in the call state can enable an application to react appropriately to something the user might be doing. For instance, a game might automatically pause and save state information when the phone rings so that the user can safely answer the call. An application can register to listen for changes in the call state by making a call to the listen() method of TelephonyManager.

```
telManager.listen(new PhoneStateListener() {
     public void onCallStateChanged(
     int state, String incomingNumber) {
            String newState = getCallStateString(state);
            if (state == TelephonyManager.CALL_STATE_RINGING) {
                   Log.i("telephony", newState +
                   " number = " + incomingNumber);
            } else {
            Log.i("telephony", newState);
            }
     }
}, PhoneStateListener.LISTEN_CALL_STATE);
```

The listener is called, in this case, whenever the phone starts ringing, the user makes a call, the user answers a call, or a call is disconnected. The listener is also called right after it is assigned so an application can get the initial state.

Another useful state of the phone is determining the state of the service. This information can tell an application if the phone has coverage at all, if it can only make emergency calls, or if the radio for phone calls is turned off as it might be when in airplane mode. To do this, an application can add the Phone State Listener .LISTEN_ SERVICE_STATE flag to the listener described earlier and implement the onServiceStateChanged method, which receives an instance of the ServiceState object. Alternatively, an application can check the state by constructing a ServiceState object and querying it directly, as shown here:

```
int serviceStatus = serviceState.getState();
String serviceStateString = null;
switch (serviceStatus) {
     case ServiceState.STATE_EMERGENCY_ONLY:
            serviceStateString = "Emergency calls only";
            break;
     case ServiceState.STATE_IN_SERVICE:
            serviceStateString = "Normal service";
            break;
     case ServiceState.STATE_OUT_OF_SERVICE:
            serviceStateString = "No service available";
            break;
     case ServiceState.STATE_POWER_OFF:
            serviceStateString = "Telephony radio is off";
            break;
}
Log.i("telephony", serviceStateString);
```

In addition, a status such as whether the handset is roaming can be determined by a call to the getRoaming() method. A friendly and frugal application can use this method to warn the user before performing any costly roaming operations such as data transfers within the application.

Requesting Service Information

In addition to call and service state information, your application can retrieve other information about the device. This information is less useful for the typical application but can diagnose problems or provide specialized services available only from certain provider networks. The following code retrieves several pieces of service information:

```
String opName = telManager.getNetworkOperatorName();
Log.i("telephony", "operator name = " + opName);
String phoneNumber = telManager.getLine1Number();
Log.i("telephony", "phone number = " + phoneNumber);
String providerName = telManager.getSimOperatorName();
Log.i("telephony", "provider name = " + providerName);
```

The network operator name is the descriptive name of the current provider that the handset connects to. This is typically the current tower operator. The SIM operator name is typically the name of the provider that the user is subscribed to for service. The phone number for this application programming interface (API) is defined as the MSISDN, typically the directory number of a GSM handset (that is, the number someone would dial to reach that particular phone).

Monitoring Signal Strength and Data Connection Speed

Sometimes an application might want to alter its behavior based upon the signal strength or service type of the device. For example, a high-bandwidth application might alter stream quality or buffer size based on whether the device has a low-speed connection (such as 1xRTT or EDGE) or a high-speed connection (such as EVDO or HSDPA). TelephonyManager can be used to determine such information.

If your application needs to react to changes in telephony state, you can use the listen() method of TelephonyManager and implement a PhoneStateListener to receive changes in service, data connectivity, call state, signal strength, and other phone state information.

Working with Phone Numbers

Applications that deal with telephony, or even just contacts, frequently have to deal with the input, verification, and usage of phone numbers. The Android SDK includes a set of helpful utility functions that simplify handling of phone number strings. Applications can have phone numbers formatted based on the current locale setting. For example, the following code uses the formatNumber() method:

```
String formattedNumber = PhoneNumberUtils.formatNumber("9995551212");
Log.i("telephony", formattedNumber);
```

The resulting output to the log would be the string 999-555-1212 in my locale. Phone numbers can also be compared using a call to the PhoneNumberUtils.compare() method. An application can also check to see if a given phone number is an emergency phone number by calling PhoneNumberUtils.isEmergencyNumber(), which enables your application to warn users before they call an emergency number. This method is useful when the source of the phone number data might be questionable.

The formatNumber() method can also take an Editable as a parameter to format a number in place. The useful feature here is that you can assign the Phone Number Formatting TextWatcher object to watch a TextView (or EditText for user input) and format phone numbers as they are entered. The following code demonstrates the ease of configuring an EditText to format phone numbers that are entered:

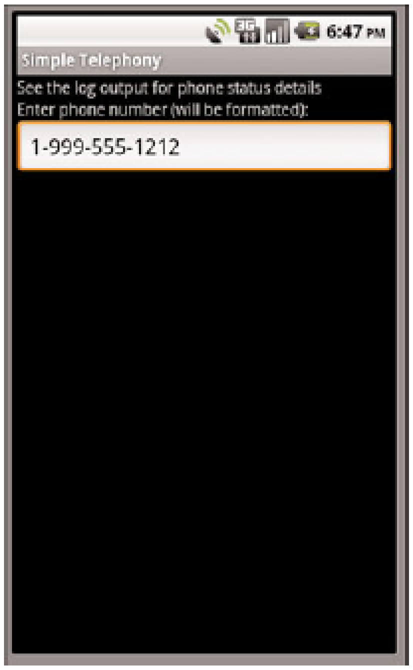
```
EditText numberEntry = (EditText) findViewById(R.id.number_entry);
numberEntry.addTextChangedListener(new
     PhoneNumberFormattingTextWatcher());
```

While the user is typing in a valid phone number, the number is formatted in a way suitable for the current locale. Just the numbers for 19995551212 were entered on the EditText shown in Figure.

Using SMS

SMS usage has become ubiquitous in the last several years. Integrating messaging services, even if only outbound, to an application can provide familiar social functionality to the user. SMS functionality is provided to applications through the android.telephony package.

Screen showing formatting results after entering only digits.



See the log output for phone status details
Enter phone number (will be formatted):
1-999-555-1212

Gaining Permission to Send and Receive SMS Messages

SMS functionality requires two different permissions, depending on if the application sends or receives messages. The following XML, to be placed with Android Manifest .xml, shows the permissions needed for both actions:

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

Sending an SMS

To send an SMS, an application first needs to get an instance of the SmsManager. Unlike other system services, this is achieved by calling the static method get Default() of SmsManager:

```
final SmsManager sms = SmsManager.getDefault();
```

Now that the application has the SmsManager, sending SMS is as simple as a single call:

```
sms.sendTextMessage("9995551212", null, "Hello!", null, null);
```

The application does not know if the actual sending of the SMS was successful without providing a PendingIntent to receive the broadcast of this information. The following code demonstrates configuring a PendingIntent to listen for the status of the SMS:

```
Intent msgSent = new Intent("ACTION_MSG_SENT");
final PendingIntent pendingMsgSent =
     PendingIntent.getBroadcast(this, 0, msgSent, 0);
registerReceiver(new BroadcastReceiver() {
     public void onReceive(Context context, Intent intent) {
            int result = getResultCode();
            if (result != Activity.RESULT_OK) {
                  Log.e("telephony",
                        "SMS send failed code = " + result);
                  pendingMsgReceipt.cancel();
             } else {
                  messageEntry.setText("");
             }
     }
}, new IntentFilter("ACTION_MSG_SENT"));
```

The PendingIntent pendingMsgSent can be used with the call to the sendText Message(). The code for the message-received receipt is similar but is called when the sending handset receives acknowledgment from the network that the destination handset received the message.

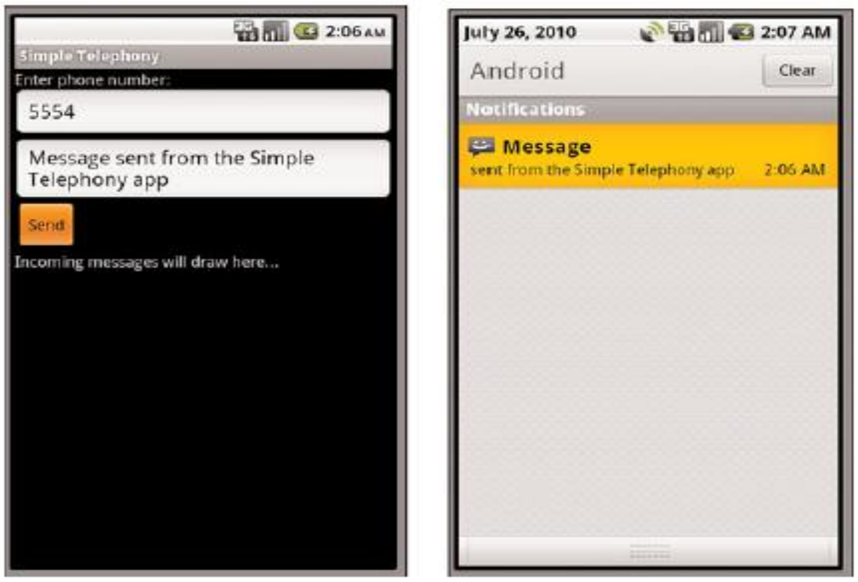
If we put all this together with the preceding phone number formatting EditText, a new entry field for the message, and a button, we can create a simple form for sending an SMS message. The code for the button handling looks like the following:

```
Button sendSMS = (Button) findViewById(R.id.send_sms);
sendSMS.setOnClickListener(new View.OnClickListener() {
     public void onClick(View v) {
```

```
                String destination =
                        numberEntry.getText().toString();
                String message =
                        messageEntry.getText().toString();
                sms.sendTextMessage(destination, null, message,
                pendingMsgSent, pendingMsgReceipt);
                registerReceiver(...);
        }
}
```

After this code is hooked in, the result should look something like Figure.Within this application, we used the emulator "phone number" trick (its port number).This is a great way to test sending SMS messages without using hardware or without incurring charges by the handset operator.

Two emulators, one sending an SMS from an application and one receiving an SMS.


2:06 AM
Simple Telephony
Enter phone number:
5554
Message sent from the Simple Telephony app
Send
Incoming messages will draw here...
July 26, 2010
2:07 AM
Android
Clear
Notifications
Message
sent from the Simple Telephony app
2:06 AM

A great way to extend this would be to set the sent receiver to modify a graphic on the screen until the sent notification is received. Further, you could use another graphic to indicate when the recipient has received the message. Alternatively, you could use ProgressBar widgets track the progress to the user.

Receiving an SMS

Applications can also receive SMS messages. To do so, your application must register a BroadcastReceiver to listen for the Intent action associated with receiving an SMS. An application listening to SMS in this way doesn’t prevent the message from getting to other applications.

Expanding on the previous example, the following code shows how any incoming text message can be placed within a TextView on the screen:

```
final TextView receivedMessage = (TextView)findViewById(
     R.id.received_message);
rcvIncoming  = new BroadcastReceiver() {
     public void onReceive(Context context, Intent intent) {
            Log.i(“telephony”, “SMS received”);
            Bundle data = intent.getExtras();
            if (data != null) {
                   Object pdus[] = (Object[]) data.get(“pdus”);
                   String message = “New message:\n”;
                   String sender = null;
                   for (Object pdu : pdus) {
                          SmsMessage part = SmsMessage.
                          createFromPdu((byte[])pdu);
                          message += part.getDisplayMessageBody();
                          (sender == null) {
                          sender = part.
                                 getDisplayOriginatingAddress();
                          }
                   }
            receivedMessage.setText(message + “\nFrom: “+sender);
            numberEntry.setText(sender);
            }
     }
};
registerReceiver(rcvIncoming, new IntentFilter(
     “android.provider.Telephony.SMS_RECEIVED”));
```

This block of code is placed within the onCreate() method of the Activity. First, the message Bundle is retrieved. In it, an array of Objects holds several byte arrays that contain PDU data—the data format that is customarily used by wireless messaging protocols. Luckily, the Android SDK can decode these with a call to the static SmsMessage.createFromPdu() utility method. From here, we can retrieve the body of the SMS message by calling getDisplayMessageBody().

The message that comes in might be longer than the limitations for an SMS. If it is, it will have been broken up in to a multipart message on the sending side. To handle this, we loop through each of the received Object parts and take the corresponding body from each, while only taking the sender address from the first.

Next, the code updates the text string in the TextView to show the user the received message. The sender address is also updated so that the recipient can respond with less typing. Finally, we register the BroadcastReceiver with the system. The IntentFilter used here, android.provider.Telephony.SMS_RECEIVED, is a well-known but undocumented IntentFilter used for this. As such, we have to use the string literal for it.

Making and Receiving Phone Calls

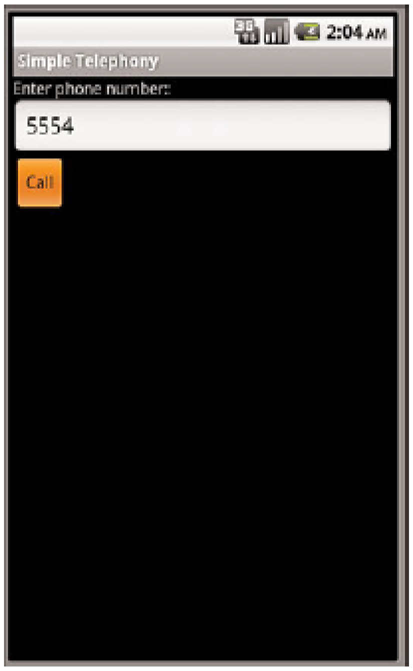
It might come as a surprise to the younger generation (they usually just text), but phones are often still used for making and receiving phone calls. Any application can be made to initiate calls and answer incoming calls; however, these abilities should be used judiciously so as not to unnecessarily disrupt the calling functionality of the user’s device.

Making Phone Calls

You’ve seen how to find out if the handset is ringing. Now let’s look at how to enable your application to make phone calls as well.

Building on the previous example, which sent and received SMS messages, we now walk through similar functionality that adds a call button to the screen to call the phone number instead of messaging it.

The Android SDK enables phone numbers to be passed to the dialer in two different ways. The first way is to launch the dialer with a phone number already entered. The user then needs to press the Send button to actually initiate the call. This method does not require any specific permissions. The second way is to actually place the call. This method requires the android.permission.CALL_PHONE permission to be added to the application’s AndroidManifest.xml file.



Let’s look at an example of how to enable an application to take input in the form of a phone number and launch the Phone dialer after the user presses a button, as shown in Figure.

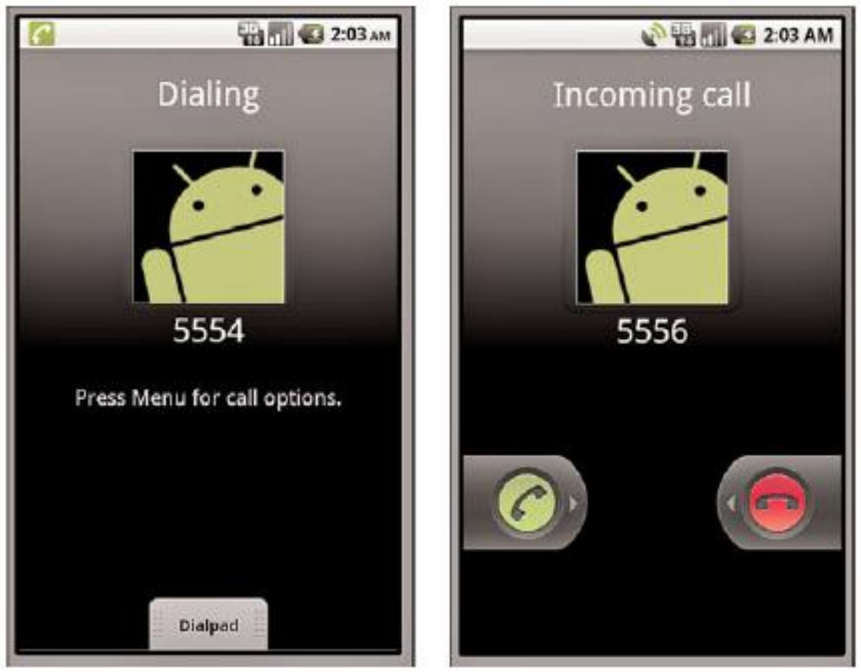
The user can enter a phone number in the EditText control and press the Call button to initiate a phone call from within the application.

We extract the phone number the user entered in the EditText field (or the most recently received SMS when continuing with the previous example).The following code demonstrates how to launch the dialer after the user presses the Call button:

```
Button call = (Button)
     findViewById(R.id.call_button);
    call.setOnClickListener(new
     View.OnClickListener() {
    public void onClick(View v) {
            Uri number = Uri.parse(“tel:” +
            numberEntry.getText().toString());
            Intent dial = new Intent(
            Intent.ACTION_DIAL, number);
                                startActivity(dial);
                                }
                         });
```

First, the phone number is requested from the EditText and tel: is prepended to it, making it a valid Uri for the Intent. Then, a new Intent is created with Intent .ACTION _DIAL to launch in to the dialer with the number dialed in already. You can also use Intent.ACTION_VIEW, which functions the same. Replacing it with Intent .ACTION_CALL, however, immediately calls the number entered. This is generally not recommended; otherwise, calls might be made by mistake. Finally, the startActivity() method is called to launch the dialer, as shown in Figure.

One emulator calling the other after the Call button is pressed within the application.



Receiving Phone Calls

Much like applications can receive and process incoming SMS messages, an application can register to answer incoming phone calls. To enable this within an application, you must implement a broadcast receiver to process intents with the action Intent.ACTION_ANSWER.

Remember, too, that if you're not interested in the call itself, but information about the incoming call, you might want to consider using the CallLog.Calls content provider (android.provider.CallLog) instead. You can use the CallLog.calls class to determine recent call information, such as

- Who called
- When they called

- Whether it was an incoming or outgoing call
- Whether or not anyone answered
- The duration of the call

Working with Notifications

Working with Notifications

Applications often need to communicate with the user even when the application isn't actively running. Applications can alert users with text notifications, vibration, blinking lights, and even audio. In this chapter, you learn how to build different kinds of notifications into your Android applications.

Notifying the User

Applications can use notifications to greatly improve the user's experience. For example:

- An email application might notify a user when new messages arrive. A news reader application might notify a user when there are new articles to read.
- A game might notify a user when a friend has signed in, or sent an invitation to play, or beat a high score.
- A weather application might notify a user of special weather alerts.
- A stock market application might notify the user when certain stock price targets are met. (Sell now! Before it's too late!)

Users appreciate these notifications because they help drive application workflow, reminding the users when they need to launch the application. However, there is a fine line between just enough and too many notifications. Application designers need to consider carefully how they should employ the use of notifications so as not to annoy the user or interrupt them without good reason. Each notification should be appropriate for the specific application and the event the user is being notified of For example, an application should not put out an emergency style notification (think flashing lights, ringing noises, and generally making a "to-do") simply to notify the user that his picture has been uploaded to a website or that new content has been downloaded.

The Android platform provides a number of different ways of notifying the user. Notifications are often displayed on the status bar at the top of the screen. Notifications may involve

- Textual information
- Graphical indicators
- Sound indicators
- Vibration of the device
- Control over the indicator light

Notifying with the Status Bar

The standard location for displaying notifications and indicators on an Android device is the status bar that runs along the top of the screen. Typically, the status bar shows information such as the current date and time. It also displays notifications (like incoming SMS messages) as they arrive—in short form along the bar and in full if the user pulls down the status bar to see the notification list. The user can clear the notifications by pulling down the status bar and hitting the Clear button.

Developers can enhance their applications by using notifications from their applications to inform the user of important events. For example, an application might want to send a simple notification to the user whenever new content has been downloaded. A simple notification has a number of important components:

- An icon (appears on status bar and full notification)
- Ticker text (appears on status bar)
- Notification title text (appears in full notification)
- Notification body text (appears in full notification)

- An intent (launches if the user clicks on the full notification)

In this section, you learn how to create this basic kind of notification.

Using the NotificationManager Service

All notifications are created with the help of the NotificationManager. The Notification Manager (within the android.app package) is a system service that must be requested. The following code demonstrates how to obtain a valid NotificationManager object using the getSystemService() method:

```
NotificationManager notifier = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
```

The NotificationManager is not useful without having a valid Notification object to use with the notify() method. The Notification object defines what information displays to the user when the Notification is triggered. This includes text that displays on the status bar, a couple of lines of text that display on the expanded status bar, an icon displayed in both places, a count of the number of times this Notification has been triggered, and a time for when the last event that caused this Notification took place.

Creating a Simple Text Notification with an Icon

You can set the icon and ticker text, both of which display on the status bar, through the constructor for the Notification object, as follows:

```
Notification notify = new Notification(
R.drawable.android_32, "Hello!", System.currentTimeMillis());
```

Additionally, you can set notification information through public member variable assignment, like this:

```
notify.icon = R.drawable.android_32;
notify.tickerText = "Hello!";
notify.when = System.currentTimeMillis();
```

You need to set a couple more pieces of information before the call to the notify() method takes place. First,we need to make a call to the setLastEventInfo() method, which configures a View that displays in the expanded status bar. Here is an example:

```
Intent toLaunch = new Intent
(SimpleNotificationsActivity.this,
SimpleNotificationsActivity.class);
PendingIntent intentBack = PendingIntent.getActivity
(SimpleNotificationsActivity.this, 0, toLaunch, 0);
notify.setLatestEventInfo(SimpleNotificationsActivity.this,
"Hi there!", "This is even more text.", intentBack);
```

Next, use the notify() method to supply the notification's title and body text as well as the Intent triggered when the user clicks on the notification. In this case, we're using our own Activity so that when the user clicks on the notification, our Activity launches again.

Working with the Notification Queue

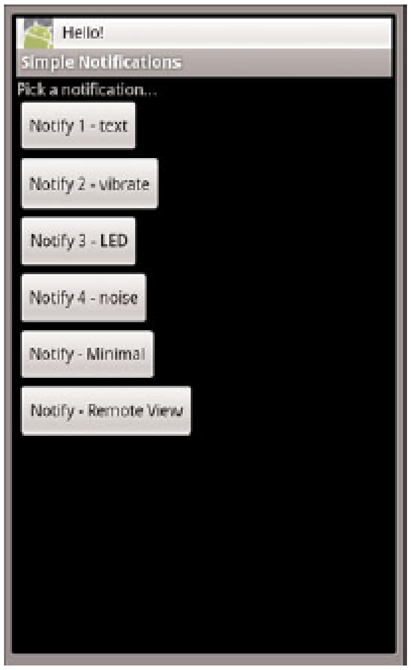
Now the application is ready to actually notify the user of the event. All that is needed is a call to the notify() method of the NotificationManager with an identifier and the Notification we configured. This is demonstrated with the following code:

```
private static final int NOTIFY_1 = 0x1001;
//  ...
notifier.notify(NOTIFY_1, notify);
```

The identifier matches up a Notification with any previous Notification instances of that type. When the identifiers match, the old Notification is updated instead of creating a new one. You might have a Notification that some file is being downloaded. You could update the Notification when the download is complete, instead of filling the Notification queue with a separate Notification, which quickly becomes obsolete. This Notification identifier only needs to be unique within your application.

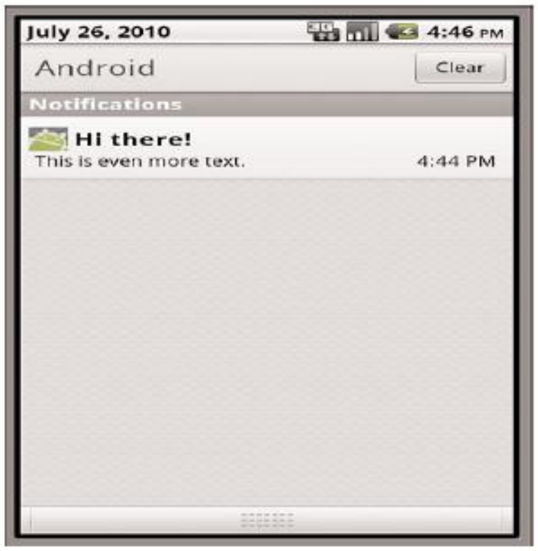
The notification displays as an icon and ticker text showing up on the status bar. This is shown at the top of Figure.

Status bar notification showing an icon and ticker text.



Shortly after the ticker text displays, the status bar returns to normal with each notification icon shown. If the users expand the status bar, they see something like what is shown in Figure.

Expanded status bar showing the icon, both text fields, and the time of the notification.



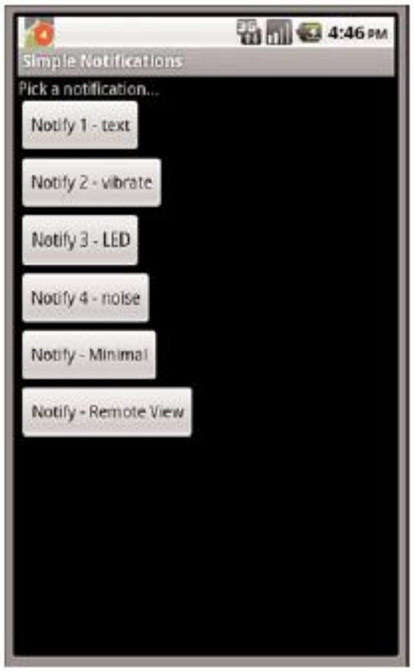
Updating Notifications

You don't want your application's notifications piling up in the notification bar. Therefore, you might want to reuse or update notifications to keep the notification list manageable. For example, there is no reason to keep a notification informing the user that the application is downloading File X when you now want to send another notification saying File X has finished downloading. Instead, you can simply update the first notification with new information.

When the notification identifiers match, the old notification is updated. When a notification with matching identifier is posted, the ticker text does not draw a second time. To show the user that something has changed, you can use a counter. The value of the number member variable of the Notification object tracks and displays this. For instance, we can set it to the number 4, as shown here: notify.number = 4;

This is displayed to the user as a small number over the icon. This is only displayed in the status bar and not in the expanded status bar, although an application could update the text to also display this information. Figure shows what this might look like in the status bar.

Status bar notification with the count of "4" showing over the icon.



Clearing Notifications

When a user clicks on the notification, the Intent assigned is triggered. At some point after this, the application might want to clear the notification from the system notifications queue. This is done through a call to the cancel() method of the NotificationManager object. For instance, the notification we created earlier could be canceled with the following call:

```
notifier.cancel(NOTIFY_1);
```

This cancels the notification that has the same identifier. However, if the application doesn't care what the user does after clicking on the notification, there is an easier way to cancel notifications. Simply set a flag to do so, as shown here:

```
notify.flags |= Notification.FLAG_AUTO_CANCEL;
```

Setting the Notification.FLAG_AUTO_CANCEL flag causes notifications to be canceled when the user clicks on them. This is convenient and easy for the application when just launching the Intent is good enough.

The Notification object is a little different from other Android objects you might have encountered. Most of the interaction with it is through direct access to its public variables instead of through helper methods. This is useful for a background application or service, discussed in Chapter "Working with Services."The Notification object can be kept around and only the values that need to be changed can be modified. After any change, the Notification needs to be posted again by calling the notify() method.

Vibrating the Phone

Vibration is a great way to enable notifications to catch the attention of a user in noisy environments or alert the user when visible and audible alerts are not appropriate (though a vibrating phone is often noisy on a hard desktop surface).Android notifications give a fine level of control over how vibration is performed. However, before the application can use vibration with a notification, an explicit permission is needed. The following XML within your application's AndroidManifest.xml file is required to use vibration:

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

Without this permission, the vibrate functionality will not work nor will there be any error. With this permission enabled, the application is free to vibrate the phone however it wants. This is accomplished by describing the vibrate member variable, which determines the vibration pattern. An array of long values describes the vibration duration. Thus, the following line of code enabled a simple vibration pattern that occurs whenever the notification is triggered:

```
notify.vibrate = new long[] {0, 200, 200, 600, 600};
```

This vibration pattern vibrates for 200 milliseconds and then stops vibrating for 200 milliseconds. After that, it vibrates for 600 milliseconds and then stops for that long. To repeat the Notification alert, a notification flag can be set so it doesn't stop until the user clears the notification.

```
notify.flags |= Notification.FLAG_INSISTENT;
```

An application can use different patterns of vibrations to alert the user to different types of events or even present counts. For instance, think about a grandfather clock with which you can deduce the time based on the tones that are played.

Blinking the Lights

Blinking lights are a great way to pass information silently to the user when other forms of alert are not appropriate. The Android SDK provides reasonable control over a multicolored indicator light, when such a light is available on the device. Users might recognize this light as a service indicator or battery level warning. An application can take advantage of this light as well, by changing the blinking rate or color of the light.

You must set a flag on the Notification object to use the indicator light. Then, the color of the light must be set and information about how it should blink. The following block of code configures the indicator light to shine green and blink at rate of 1 second on and 1 second off:

```
notify.flags |= Notification.FLAG_SHOW_LIGHTS;
notify.ledARGB = Color.GREEN;
notify.ledOnMS = 1000;
notify.ledOffMS = 1000;
```

Although you can set arbitrary color values, a typical physical implementation of the indicator light has three small LEDs in red, green, and blue. Although the colors blend reasonably well, they won't be as accurate as the colors on the screen. For instance, on the T-Mobile G1, the color white looks a tad pink.

An application can use different colors and different blinking rates to indicate different information to the user. For instance, the more times an event occurs, the more urgent the indicator light could be. The following block of code shows changing the light based on the number of notifications that have been triggered:

```
notify.number++;
notify.flags |=  Notification.FLAG_SHOW_LIGHTS;
if (notify.number < 2) {
     notify.ledARGB = Color.GREEN;
     notify.ledOnMS = 1000;
     notify.ledOffMS = 1000;
} else if (notify.number < 3) {
     notify.ledARGB = Color.BLUE;
     notify.ledOnMS = 750;
     notify.ledOffMS = 750;
} else if (notify.number < 4) {
     notify.ledARGB = Color.WHITE;
     notify.ledOnMS = 500;
     notify.ledOffMS = 500;
} else {
     notify.ledARGB = Color.RED;
     notify.ledOnMS = 50;
     notify.ledOffMS = 50;
}
```

The blinking light continues until the Notification is cleared by the user. The use of the Notification.FLAG_INSISTENT flag does not affect this as it does vibration effects.

Color and blinking rates could also be used to indicate other information. For instance, temperature from a weather service could be indicated with red and blue plus blink rate. Use of such colors for passive data indication can be useful even when other forms would work. It is far less intrusive than annoying, loud ringers or harsh, vibrating phone noises. For instance, a simple glance at the handset could tell the user some useful piece of information without the need to launch any applications or change what they are doing.

Making Noise

Sometimes, the handset has to make noise to get the user's attention. Luckily, the Android SDK provides a means for this using the Notification object. Begin by configuring the audio stream type to use when playing a sound. Generally, the most useful stream type is STREAM_NOTIFICATION. You can configure the audio stream type on your notification as follows:

```
      notify.audioStreamType = AudioManager.STREAM_NOTIFICATION;
```

Now, assign a valid Uri object to the sound member variable and that sound plays when the notification is triggered. The following code demonstrates how to play a sound that is included as a project resource:

```
notify.sound = Uri.parse(
"android.resource://com.androidbook.simplenotifications/"  +
R.raw.fallbackring);
```

By default, the audio file is played once. As with the vibration, the Notification. FLAG_ INSISTENT flag can be used to repeat incessantly until the user clears the notification. No specific permissions are needed for this form of notification

Customizing the Notification

Although the default notification behavior in the expanded status bar tray is sufficient for most purposes, developers can customize how notifications are displayed if they so choose. To do so, developers can use the RemoteViews object to customize the look and feel of a notification.

The following code demonstrates how to create a RemoteViews object and assign custom text to it:

```
RemoteViews remote =
new RemoteViews(getPackageName(), R.layout.remote);
remote.setTextViewText(R.id.text1, "Big text here!");
remote.setTextViewText(R.id.text2, "Red text down here!");
notify.contentView = remote;
```

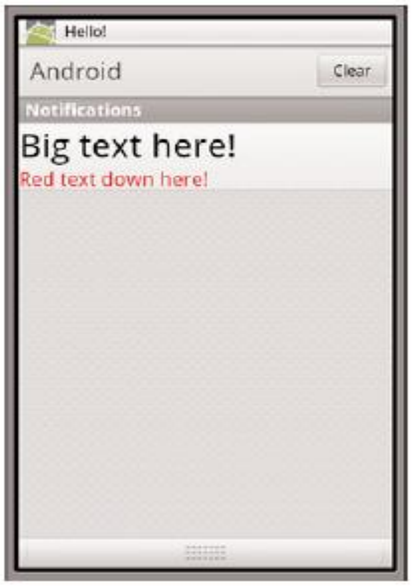
To better understand this, here is the layout file remote.xml referenced by the preceding code:

```
<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:id="@+id/text1"
```

```
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textSize="31dp"
    android:textColor="#000" />
<TextView
    android:id="@+id/text2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textSize="18dp"
    android:textColor="#f00" />
</LinearLayout>
```

This particular example is similar to the default notification but does not contain an icon. The setLatestEventInfo() method is normally used to assign the text to the default layout. In this example, we use our custom layout instead. The Intent still needs to be assigned, though, as follows:

```
    Intent toLaunch = new Intent
          (SimpleNotificationsActivity.this,
                SimpleNotificationsActivity.class);
     PendingIntent intentBack = PendingIntent.getActivity
         (SimpleNotificationsActivity.this, 0, toLaunch, 0);
         notify.contentIntent = intentBack;
         notifier.notify(NOTIFY_5, notify);
```



The end result looks something like Figure.

Custom notification showing with just two lines of text.

Using a custom notification layout can provide better control over the information on the expanded status bar. Additionally, it can help differentiate your application's notifications from other applications by providing a themed or branded appearance.

Note

The default layout includes two fields of text: an icon and a time field for when the notification was triggered. Users are accustomed to this information. An application, where feasible and where it makes sense, should try to conform to at least this level of information when using custom notifications.

Designing Useful Notifications

As you can see, the notification capabilities on the Android platform are quite robust—so robust that it is easy to overdo it and make your application tiresome for the user. Here are some tips for designing useful notifications:

- Only use notifications when your application is not in the foreground. When in the foreground, use Toast or Dialog controls.
- Allow the user to determine what types (text, lights, sound, vibration) and frequency of notifications she will receive, as well as what events to trigger notifications for.
- Whenever possible, update and reuse an existing notification instead of creating a new one.
- Clear notifications regularly so as not to overwhelm the user with dated information.
- When in doubt, generate “polite” notifications (read: quiet).
- Make sure your notifications contain useful information in the ticker, title, and body text fields and launch sensible intents.

The notification framework is lightweight yet powerful. However, some applications such as alarm clocks or stock market monitors might also need to implement their own alert windows above and beyond the notification framework provided. In this case, they may use a background service and launch full Activity windows upon certain events. In Android 2.0 and later, developers can use the WindowManager.LayoutParams class to enable activity windows to display, even when the screen is locked with a keyguard.

Working with Services

Working with Services

One important Android application component that can greatly enhance an application is a service. An Android service might be used to perform functions that do not require user input in the background, or to supply information to other applications. In this chapter, you learn how to create and interact with an Android service. You also learn how to define a remote interface using the Android Interface Definition Language (AIDL). Finally, you learn how to pass objects through this interface by creating a class that implements a Parcelable object.

Determining When to Use Services

A service within the Android Software Development Kit (SDK) can mean one of two things. First, a service can mean a background process, performing some useful operation at regular intervals. Second, a service can be an interface for a remote object, called from within your application. In both cases, the service object extends the Service class from the Android SDK, and it can be a stand-alone component or part of an application with a complete user interface.

Certainly, not all applications require or use services. However, you might want to consider a service if your application meets certain criteria, such as the following:

- The application performs lengthy or resource-intensive processing that does not require input from the user.

- The application must perform certain functions routinely, or at regular intervals, such as uploading or downloading fresh content or logging the current location.
- The application performs a lengthy operation that, if cancelled because the application exits, would be wasteful to restart. An example of this is downloading large files.
- The application needs to expose and provide data or information services (think web services) to other Android applications without the need of a user interface.

Understanding the Service Lifecycle

Before we get into the details of how to create a service, let's look at how services interact with the Android operating system. First, it should be noted that a service implementation must be registered in that application's manifest file using the <service> tag. The service implementation might also define and enforce any permissions needed for starting, stopping, and binding to the service, as well as make specific service calls.

After it's been implemented, an Android service can be started using the Context. startService() method. If the service was already running when the startService() method was called, these subsequent calls don't start further instances of the service. The service continues to run until either the Context.stopService() method is called, or the service completes its tasks and stops itself using the stopSelf() method.

To connect to a service, interested applications use the Context.bindService() method to obtain a connection. If that service is not running, it is created at that time. After the connection is established, the interested applications can begin making requests of that service, if the applications have the appropriate permissions. For example, a Magic Eight Ball application might have an underlying service that can receive yes-or-no questions and provide Yoda-style answers. Any interested application could connect to the Magic Eight Ball service, ask a question ("Will my app flourish on the Android Market?") and receive the result ("Signs point to Yes.").The application can then disconnect from the service when finished using the Context.unbindService() method.

Creating a Service

Creating an Android service involves extending the Service class and adding a service block to the AndroidManifest.xml permissions file. The GPXService class overrides the onCreate(), onStart(), onStartCommand(), and onDestroy() methods to begin with. Defining the service name enables other applications to start the service that runs in the ackground and stop it. Both the onStart() and onStartCommand() methods are essentially the same, with the exception that onStart() is deprecated in API Levels 5 and above. (The default implementation of the onStartCommand() on API Level 5 or greater is to call onStart() and returns an appropriate value such that behavior will be compatible to previous versions.) In the following example, both methods are implemented.

For this example, we implement a simple service that listens for GPS changes, displays notifications at regular intervals, and then provides access to the most recent location data via a remote interface. The following code gives a simple definition to the Service class called GPXService:

```
public class GPXService extends Service {
     public static final String GPX_SERVICE =
            “com.androidbook.GPXService.SERVICE”;
            private LocationManager location = null;
            private NotificationManager notifier = null;
            @Override
            public void onCreate() {
                   super.onCreate();
            }
            @Override
            public void onStart(Intent intent, int startId) {
                   super.onStart(intent, startId);
            }
            @Override
            public void onStartCommand(Intent intent, int flags, int
                   startId) {
                          super.onStart(intent, startId);
            }
            @Override
            public void onDestroy() {
                   super.onDestroy();
     }
}
```

You need to understand the lifecycle of a service because it’s different from that of an activity. If a service is started by the system with a call to the Context .Start Service() method, the onCreate() method is called just before the onStart() or onStartCommand() methods. However, if the service is bound to with a call to the Context.bindService() method, the onCreate() method is called just before the onBind() method. The onStart() or onStartCommand() methods are not called in this case. We talk more about binding to a service later in this chapter. Finally, when the service is finished—that is, it is stopped and no other process is bound to it—the on Destroy() method is called. Everything for the service must be cleaned up in this method.

With this in mind, here is the full implementation of the onCreate() method for the GPXService class previously introduced:

```
public void onCreate() {
      super.onCreate();
location  = (LocationManager)
     getSystemService(Context.LOCATION_SERVICE);
notifier = (NotificationManager)
getSystemService(Context.NOTIFICATION_SERVICE);
}
```

Because the object doesn’t yet know if the next call is to either of the start methods or the onBind() method, we make a couple of quick initialization calls, but no background processing is started. Even this might be too much if neither of these objects were used by the interface provided by the binder.

Because we can’t always predict what version of Android our code is running on, we can simple implement both the onStart() and onStartCommand() methods and have them call a third method that provides a common implementation. This enables us to customize behavior on later Android versions while being compatible with earlier versions. To do this, the project needs to be built for an SDK of Level 5 or higher, while having a minSdkValue of whatever earlier versions are supported. Of course, we highly recommend testing on multiple platform versions to verify that the behavior is as you expect. Here are sample implementations of the onStartCommand() and onStart() methods:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId ) {
Log.v(DEBUG_TAG, “onStartCommand() called, must be on L5 or later”);
if (flags != 0) {
Log.w(DEBUG_TAG, “Redelivered or retrying service start: “+flags);
}
doServiceStart(intent, startId);
return Service.START_REDELIVER_INTENT;
}
@Override
public void onStart(Intent intent, int startId) {
     super.onStart(intent, startId);
Log.v(DEBUG_TAG, "onStart() called, must be on L3 or L4");
doServiceStart(intent,startId);
}
```

Next, let's look at the implementation of the doStartService() method in greater detail:

```
@Override
public void doServiceStart(Intent intent, int startId) {
     super.onStart(intent, startId);
updateRate = intent.getIntExtra(EXTRA_UPDATE_RATE, -1);
if (updateRate == -1) {
     updateRate = 60000;
}
Criteria criteria = new Criteria();
criteria.setAccuracy(Criteria.NO_REQUIREMENT);
criteria.setPowerRequirement(Criteria.POWER_LOW);
location = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
String best = location.getBestProvider(criteria, true);
location.requestLocationUpdates(best,
     updateRate, 0, trackListener);
Notification notify = new
     Notification(android.R.drawable.stat_notify_more,
"GPS Tracking", System.currentTimeMillis());
notify.flags |= Notification.FLAG_AUTO_CANCEL;
Intent toLaunch = new Intent(getApplicationContext(),
     ServiceControl.class);
PendingIntent intentBack =
     PendingIntent.getActivity(getApplicationContext(),
            0, toLaunch, 0);
notify.setLatestEventInfo(getApplicationContext(),
     "GPS Tracking", "Tracking start at " +
     updateRate+"ms intervals with [" + best +
     "] as the provider.", intentBack);
     notifier.notify(GPS_NOTIFY, notify);
}
```

The background processing starts within the two start methods. In this example, though, the background processing is actually just registering for an update from another service. For more information about using location-based services and the LocationManager, see Chapter "Using Location-Based Services (LBS) APIs," and for more information on Notification calls, see Chapter "Working with Notifications."

In this case, we turn on the GPS for the duration of the process, which might affect battery life even though we request a lower power method of location determination. Keep this in mind when developing services.

The Intent extras object retrieves data passed in by the process requesting the service. Here, we retrieve one value, EXTRA_UPDATE_RATE, for determining the length of time between updates. The string for this, update-rate, must be published externally, either in developer documentation or in a publicly available class file so that users of this service know about it.

The implementation details of the LocationListener object, trackListener, are not interesting to the discussion on services. However, processing should be kept to a minimum to avoid interrupting what the user is doing in the foreground. Some testing might be required to determine how much processing a particular phone can handle before the user notices performance issues.

There are two common methods to communicate data to the user. The first is to use Notifications. This is the least-intrusive method and can be used to drive users to the application for more information. It also means the users don't need to be actively using their phone at the time of the notification because it is queued. For instance, a weather application might use notifications to provide weather updates every hour.

The other method is to use Toast messages. From some services, this might work well, especially if the user expects frequent updates and those updates work well overlaid briefly on the screen, regardless of what the user is currently doing. For instance, a background music player could briefly overlay the current song title when the song changes.

The onDestroy() method is called when no clients are bound to the service and a request for the service to be stopped has been made via a call to the Context. stop Service() method, or a call has been made to the stopSelf() method from within the service. At this point, everything should be gracefully cleaned up because the service ceases to exist.

Here is an example of the onDestroy() method:

```
@Override
public void onDestroy() {
     if (location != null) {
            location.removeUpdates(trackListener);
            location = null;
      }
     Notification notify = new
     Notification(android.R.drawable.stat_notify_more,
     "GPS Tracking", System.currentTimeMillis());
     notify.flags |= Notification.FLAG_AUTO_CANCEL;
     Intent toLaunch = new Intent(getApplicationContext(),
     ServiceControl.class);
     PendingIntent intentBack =
            PendingIntent.getActivity(getApplicationContext(),
            0, toLaunch, 0);
     notify.setLatestEventInfo(getApplicationContext(),
     "GPS Tracking", "Tracking stopped", intentBack);
     notifier.notify(GPS_NOTIFY, notify);
```

```
        super.onDestroy();
    }
```

Here, we stop updates to the LocationListener object. This stops all our background processing. Then, we notify the user that the service is terminating. Only a single call to the onDestroy() method happens, regardless of how many times the start methods are called. The system does not know about a service unless it is defined within the AndroidManifest.xml permissions file using the <service> tag. Here is the <service> tag we must add to the Android Manifest file:

```
<service
     android:enabled="true"
     android:name="GPXService">
     <intent-filter>
        <action android:name="com.androidbook.GPXService.SERVICE"/>
     </intent-filter>
</service>
```

This block of XML defines the service name, GPXService, and that the service is enabled. Then, using an Intent filter, we use the same string that we defined within the class. This is the string that is used later on when controlling the service. With this block of XML inside the application section of the manifest, the system now knows that the service exists and it can be used by other applications.

Controlling a Service

At this point, the example code has a complete implementation of a Service. Now we write code to control the service we previously defined.

```
Intent service = new Intent("com.androidbook.GPXService.SERVICE");
service.putExtra("update-rate", 5000);
startService(service);
```

Starting a service is as straightforward as creating an Intent with the service name and calling the startService() method. In this example, we also set the update-rate Intent extra parameter to 5 seconds. That rate is quite frequent but works well for testing. For practical use, we probably want this set to 60 seconds or more. This code triggers a call to the onCreate() method, if the Service isn't bound to or running already. It also triggers a call to the onStart() or onStartCommand() methods, even if the service is already running.

Later, when we finish with the service, it needs to be stopped using the following code:

```
Intent service = new Intent("com.androidbook.GPXService.SERVICE");
stopService(service);
```

This code is essentially the same as starting the service but with a call to the stopService() method.This calls the onDestroy() method if there are no bindings to it. However, if there are bindings, onDestroy() is not called until those are also terminated. This means background processing might continue despite a call to the stopService() method. If there is a need to control the background processing separate from these system calls, a remote interface is required.

Implementing a Remote Interface

Sometimes it is useful to have more control over a service than just system calls to start and stop its activities. However, before a client application can bind to a service for making other method calls, you need to define the interface. The Android SDK includes a useful tool and file format for remote interfaces for this purpose.

To define a remote interface, you must declare the interface in an AIDL file, implement the interface, and then return an instance of the interface when the onBind() method is called.

Using the example GPXService service we already built in this chapter, we now create a remote interface for it. This remote interface has a method, which can be called especially for returning the last location logged. You can use only primitive types and objects that implement the Parcelable protocol with remote service calls. This is because these calls cross process boundaries where memory can't be shared. The AIDL compiler handles the details of crossing these boundaries when the rules are followed. The Location object implements the Parcelable interface so it can be used.

Here is the AIDL file for this interface, IRemoteInterface:

```
package com.androidbook.services;
interface IRemoteInterface {
Location getLastLocation();
}
```

When using Eclipse, you can add this AIDL file, IRemoteInterface.aidl, to the project under the appropriate package and the Android SDK plug-in does the rest. Now we must implement the code for the interface. Here is an example implementation of this interface:

```
private final IRemoteInterface.Stub
mRemoteInterfaceBinder = new IRemoteInterface.Stub() {
     public Location getLastLocation() {
     Log.v("interface", "getLastLocation() called");
     return lastLocation;
     }
};
```

The service code already stored off the last location received as a member variable, so we can simply return that value. With the interface implemented, it needs to be returned from the onBind() method of the service:

```
@Override
public IBinder onBind(Intent intent) {
     // we only have one, so no need to check the intent
     return mRemoteInterfaceBinder;
}
```

If multiple interfaces are implemented, the Intent passed in can be checked within the onBind() method to determine what action is to be taken and which interface should be returned. In this example, though, we have only one interface and don't expect any other information within the Intent, so we simply return the interface.

We also add the class name of the binder interface to the list of actions supported by the intent filter for the service within the AndroidManifest.xml file. Doing this isn't required but is a useful convention to follow and allows the class name to be used. The following block is added to the service tag definition:

```
<action android:name = "com.androidbook.services.IRemoteInterface"/>
```

The service can now be used through this interface. This is done by implementing a ServiceConnection object and calling the bindService() method. When finished, the unbindService() method must be called so the system knows that the application is done using the service. The connection remains even if the reference to the interface is gone.

Here is an implementation of a ServiceConnection object’s two main methods, onService Connected() and onService Disconnected():

```
public void onServiceConnected(ComponentName name,
IBinder service) {
     mRemoteInterface =
            IRemoteInterface.Stub.asInterface(service);
            Log.v(“ServiceControl”, “Interface bound.”);
}
public void onServiceDisconnected(ComponentName name) {
     mRemoteInterface = null;
     Log.v(“ServiceControl”,“Remote interface no longer bound”);
}
```

When the onServiceConnected() method is called, an IRemoteInterface instance that can be used to make calls to the interface we previously defined is retrieved. A call to the remote interface looks like any call to an interface now:

```
     Location loc = mRemoteInterface.getLastLocation();
```

To use this interface from another application, you should place the AIDL file within the project and appropriate package. The call to onBind() triggers a call to the onService Connected() after the call to the service’s onCreate() method. Remember, the onStart() or onStartCommand() methods are not called in this case.

```
bindService(new Intent(IRemoteInterface.class.getName()),
this, Context.BIND_AUTO_CREATE);
```

In this case, the Activity we call from also implements the ServiceConnection interface. This code also demonstrates why it is a useful convention to use the class name as an intent filter. Because we have both intent filters and we don’t check the action on the call to the onBind() method, we can also use the other intent filter, but the code here is clearer.

When done with the interface, a call to unbindService() disconnects the interface. However, a callback to the onServiceDisconnected() method does not mean that the service is no longer bound; the binding is still active at that point, just not the connection.

Implementing a Parcelable Class

In the example so far, we have been lucky in that the Location class implements the Parcelable interface. What if a new object needs to be passed through a remote interface?

Let's take the following class, GPXPoint, as an example:

```
public final class GPXPoint {
     public int latitude;
     public int longitude;
     public Date timestamp;
     public double elevation;
     public GPXPoint() {
     }
}
```

The GPXPoint class defines a location point that is similar to a GeoPoint but also includes the time the location was recorded and the elevation. This data is commonly found in the popular GPX file format. On its own, this is not a basic format that the system recognizes to pass through a remote interface. However, if the class implements the Parcelable interface and we then create an AIDL file from it, the object can be used in a remote interface.

To fully support the Parcelable type, we need to implement a few methods and a Parcelable.Creator<GPXPoint>.The following is the same class now modified to be a Parcelable class:

```
public final class GPXPoint implements Parcelable {
    public int latitude;
    public int longitude;
    public Date timestamp;
    public double elevation;
    public static final Parcelable.Creator<GPXPoint>
    CREATOR = new Parcelable.Creator<GPXPoint>() {
          public GPXPoint createFromParcel(Parcel src) {
                return new GPXPoint(src);
          }
    public GPXPoint[] newArray(int size) {
                return new GPXPoint[size];

          }
    };
  public GPXPoint() {
  }
  private GPXPoint(Parcel src) {
    readFromParcel(src);
  }
  public void writeToParcel(Parcel dest, int flags) {
    dest.writeInt(latitude);
    dest.writeInt(longitude);
    dest.writeDouble(elevation);
    dest.writeLong(timestamp.getTime());
  }
  public void readFromParcel(Parcel src) {
    latitude = src.readInt();
    longitude = src.readInt();
    elevation = src.readDouble();
    timestamp = new Date(src.readLong());
  }
  public int describeContents() {
    return 0;
  }
 }
```

The writeToParcel() method is required and flattens the object in a particular order using supported primitive types within a Parcel. When the class is created from a Parcel, the Creator is called, which, in turn, calls the private constructor. For readability, we also created a readFromParcel() method that reverses the flattening, reading the primitives in the same order that they were written and creating a new Date object.

Now you must create the AIDL file for this class. You should place it in the same directory as the Java file and name it GPXPoint.aidl to match. You should make the contents look like the following:

```
package com.androidbook.services;
parcelable GPXPoint;
```

Now the GPXPoint class can be used in remote interfaces. This is done in the same way as any other native type or Parcelable object. You can modify the IRemote Interface .aidl file to look like the following:

```
package com.androidbook.services;
  import com.androidbook.services.GPXPoint;
  interface IRemoteInterface {
  Location getLastLocation();
  GPXPoint getGPXPoint();
  }
```

Additionally,we can provide an implementation for this method within the interface, as follows:

```
public GPXPoint getGPXPoint() {
     if (lastLocation ==  null) {
            return null;
      } else {
            Log.v("interface", "getGPXPoint() called");
            GPXPoint point = new GPXPoint();
            point.elevation = lastLocation.getAltitude();
            point.latitude = (int)(lastLocation.getLatitude()*1E6);
            point.longitude = (int)(lastLocation.getLongitude()*1E6);
            point.timestamp = new Date(lastLocation.getTime());
            return point;
     }
}
```

As can be seen, nothing particularly special needs to happen. Just by making the object Parcelable, it can now be used for this purpose.

Selling Your Android Application

Selling Your Android Application

After you've developed an application, the next logical step is to publish it so that other people can enjoy it. You might even want to make some money. There are a variety of distribution opportunities available to Android application developers. Many developers choose to sell their applications through mobile market places such as Google's Android Market. Others develop their own distribution mechanisms—for example, they might sell their applications from a website. Regardless, developers should consider which distribution options they plan to use during the application design and development process, as some distribution choices might require code changes or impose restrictions on content.

Choosing the Right Distribution Model

The application distribution methods you choose to employ depend on your goals and target users. Some questions you should ask yourself are

- Is your application ready for prime time or are you considering a beta period to iron out the kinks?
- Are you trying to reach the broadest audience, or have you developed a vertical market application? Determine who your users are, which devices they are using, and their preferred methods for seeking out and downloading applications.
- How will you price your application? Is it freeware or shareware? Are the payment models (single payment versus subscription model versus ad-driven revenue) you require available on the distribution mechanisms you want to leverage?
- Where do you plan to distribute? Verify that any application markets you plan to use are capable of distributing within those countries or regions.

- Are you willing to share a portion of your profits? Distribution mechanisms such as the Android Market take a percentage of each sale in exchange for hosting your application for distribution and collecting application revenue on your behalf.
- Do you require complete control over the distribution process or are you willing to work within the boundaries and requirements imposed by third-party market places?
- This might require compliance with further license agreements and terms.
- If you plan to distribute yourself, how will you do so? You might need to develop more services to manage users, deploy applications and collect payments. If so, how will you protect user data? What trade laws must you comply with?
- Have you considered creating a free trial version of your application? If the distribution system under consideration has a return policy, consider the ramifications. You need to ensure that your application has safeguards to minimize the number of users that buy your app, use it, and return it for a full refund. For example, a game might include safeguards such as a free trial version and a full-scale version with more game levels than could possibly be completed within the refundable time period.

Now let’s look at the steps you need to take to package and publish your application.

Packaging Your Application for Publication

There are several steps developers must take when preparing an Android application for publication and distribution. Your application must also meet several important requirements imposed by the marketplaces. The following steps are required for publishing an application:

1. Prepare and perform a release candidate build of the application.
2. Verify that all requirements for marketplace are met, such as configuring the Android manifest file properly. For example, make sure the application name and version information is correct and the debuggable attribute is set to false.
3. Package and digitally sign the application.

4. Test the packaged application release thoroughly.
5. Publish the application.

The preceding steps are required but not sufficient to guarantee a successful deployment. Developers should also

1. Thoroughly test the application on all target handsets.
2. Turn off debugging, including Log statements and any other logging.
3. Verify permissions, making sure to add ones for services used and remove any that aren’t used, regardless of whether they are enforced by the handsets.
4. Test the final, signed version with all debugging and logging turned off.

Now, let’s explore each of these steps in more detail, in the order they might be performed.

Preparing Your Code to Package

An application that has undergone a thorough testing cycle might need changes made to it before it is ready for a production release. These changes convert it from a debuggable, preproduction application into a release-ready application.

Setting the Application Name and Icon

An Android application has default settings for the icon and label. The icon appears in the application Launcher and can appear in various other locations, including marketplaces. As such, an application is required to have an icon. You should supply alternate icon drawable resources for various screen resolutions. The label, or application name, is also displayed in similar locations and defaults to the package name. You should choose a user-friendly name.

Versioning the Application

Next, proper versioning is required, especially if updates could occur in the future. The version name is up to the developer. The version code, though, is used internally by the Android system to determine if an application is an update. You should increment the version code for each new update of an application. The exact value doesn’t matter, but it must be greater than the previous version code. Versioning within the Android manifest file is discussed in Chapter “Defining Your Application Using the Android Manifest File.”

Verifying the Target Platforms

Make sure your application sets the <uses-sdk> tag in the Android manifest file correctly. This tag is used to specify the minimum and target platform versions that the application can run on. This is perhaps the most important setting after the application name and version information.

Configuring the Android Manifest for Market Filtering

If you plan to publish through the Android Market, you should read up on how this distribution system uses certain tags within the Android manifest file to filter applications available to users. Many of these tags, such as <supports-screens>, <uses-configuration>, <uses-feature>, <uses-library>, <uses-permission>, and <uses-sdk>, were discussed in Chapter “Defining Your Application Using the Android Manifest File” Set each of these settings carefully, as you don’t want to accidentally put too many restrictions on your application. Make sure you test your application thoroughly after configuring these Android manifest file settings. For more information on how Android Market filters work, see market-filters.html.

Preparing Your Application Package for the Android Market

The Android Market has strict requirements on application packages. When you upload your application to the Android Market website, the package is verified and any problems are communicated to you. Most often, problems occur when you have not properly configured your Android manifest file.

The Android Market uses the android:versionName attribute of the <manifest> tag within the Android manifest file to display version information to users. It also uses the android:versionCode attribute internally to handle application upgrades. The android:icon and android:label attributes must also be present because both are used by the Android Market to display the application name to the user with a visual icon.

Disabling Debugging and Logging

Next, you should turn off debugging and logging. Disabling debugging involves removing the android: debuggable attribute from the <application> tag of the AndroidManifest.xml file or setting it to false. You can turn off the logging code within Java in a variety of different ways, from just commenting it out to using a build system that can do this automatically.

Verifying Application Permissions

Finally, the permissions used by the application should be reviewed. Include all permissions that the application requires, and remove any that are not used. Users appreciate this.

Packing and Signing Your Application

Now that the application is ready for publication, the file package—the .apk file—needs to be prepared for release. The package manager of an Android device will not install a package that has not been digitally signed. Throughout the development process, the Android tools have accomplished this through signing with a debug key. The debug key cannot be used for publishing an application to the wider world. Instead, you need to use a true key to digitally sign the application. You can use the private key to digitally sign the release package files of your Android application, as well as any upgrades. This ensures that the application (as a complete entity) is coming from you, the developer, and not some other source (imposters!).

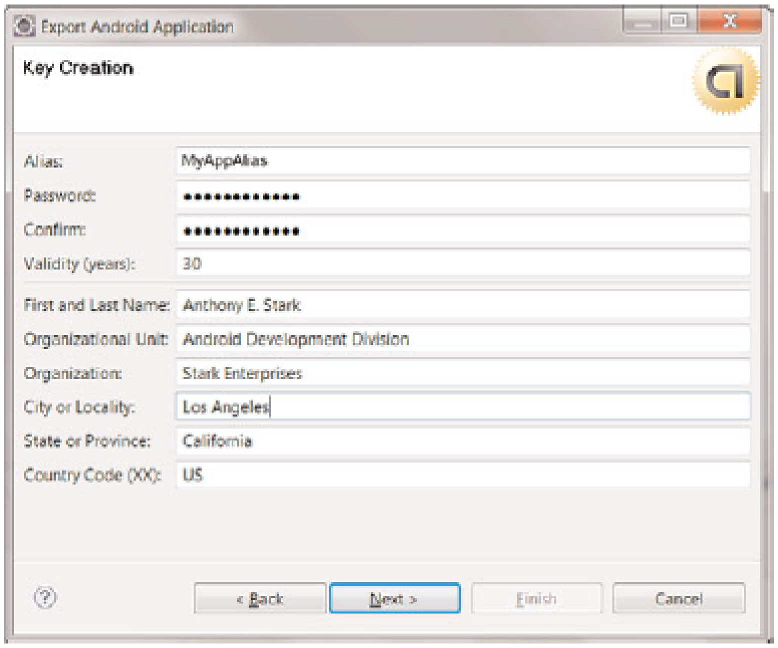
The Android Market requires that your application's digital signature validity period end after October 22, 2033.This date might seem like a long way off and, for mobile, it certainly is. However, because an application must use the same key for upgrading and applications that want to work closely together with special privilege and trust relationships must also be signed with the same key, the key could be chained forward through many applications. Thus, Google is mandating that the key be valid for the foreseeable future so application updates and upgrades are performed smoothly for users.

Although self-signing is typical of Android applications, and a certificate authority is not required, creating a suitable key and securing it properly is critical. The digital signature for Android applications can impact certain functionality. The expiry of the signature is verified at installation time, but after it's installed, an application continues to function even if the signature has expired.

You can export and sign your Android package file from within Eclipse using the Android Development plug-in, or you can use the command-line tools. You can export and sign your Android package file from within Eclipse by taking the following steps:

1. In Eclipse, right-click the appropriate application project and choose the Export option.
2. Under the Export menu, expand the Android section and choose Export Android Application.
3. Click the Next button.
4. Select the project to export (the one you right-clicked before is the default).
5. On the keystore selection screen, choose the Create New Keystore option and enter a file location (where you want to store the key) as well as a password for managing the keystore. (If you already have a keystore, choose browse to pick your keystore file, and then enter the correct password.)
6. Click the Next button.
7. On the Key Creation screen, enter the details of the key, as shown in Figure.
8. Click the Next button.
9. On the Destination and Key/Certificate Checks screen, enter a destination for the application package file then Click the Finish button.

Exporting and signing an Android application in Eclipse.



Alias:	MyAppAlias
Password:	•••••••••••••
Confirm:	••••••••••••
Validity (years):	30
First and Last Name:	Anthony E. Stark
Organizational Unit:	Android Development Division
Organization:	Stark Enterprises
City or Locality:	Los Angeles
State or Province:	California
Country Code (XX):	US

You have now created a fully signed and certified application package file. The application package is ready for publication.

Testing the Release Version of Your Application Package

Now that you have configured your application for production, you should perform a full final testing cycle paying special attention to subtle changes to the installation process. An important part of this process is to verify that you have disabled all debugging features and logging has no negative impact on the functionality and performance of the application.

Certifying Your Android Application

If you're familiar with other mobile platforms, you might be familiar with the many strict certification programs found on platforms, such as the TRUE BREW or Symbian Signed programs. These programs exist to enforce a lower bound on the quality of an application.

As of this writing, Android does not have any certification or testing requirements. It is an open market with only a few content guidelines and rules to follow. This does not mean, however, that certification won't be required at some point or that certain distribution means won't require certification.

Typically, certification programs require rigorous and thorough testing, certain usability conventions must be met, and various other constraints that might be goo common practice or operator-specific rules are enforced. The best way to prepare for any certification program is to incorporate its requirements into the design of your specific project. Following best practices for Android development and developing efficient, usable, dynamic, and robust applications always pay off in the end—whether your application requires certification.

Distributing Your Applications

Now that you've prepared your application for publication, it's time to get your application out to users—for fun and profit. Unlike other mobile platforms, most Android distribution mechanisms support free applications and price plans.

Selling Your Application on the Android Market

The Android Market is the primary mechanism for distributing Android applications at this time. This is where your typical user purchases and downloads applications. As of this writing, it's available to most, but not all, Android devices. As such, we show you how to check your package for preparedness, sign up for a developer account, and submit your application for sale on the Android Market.

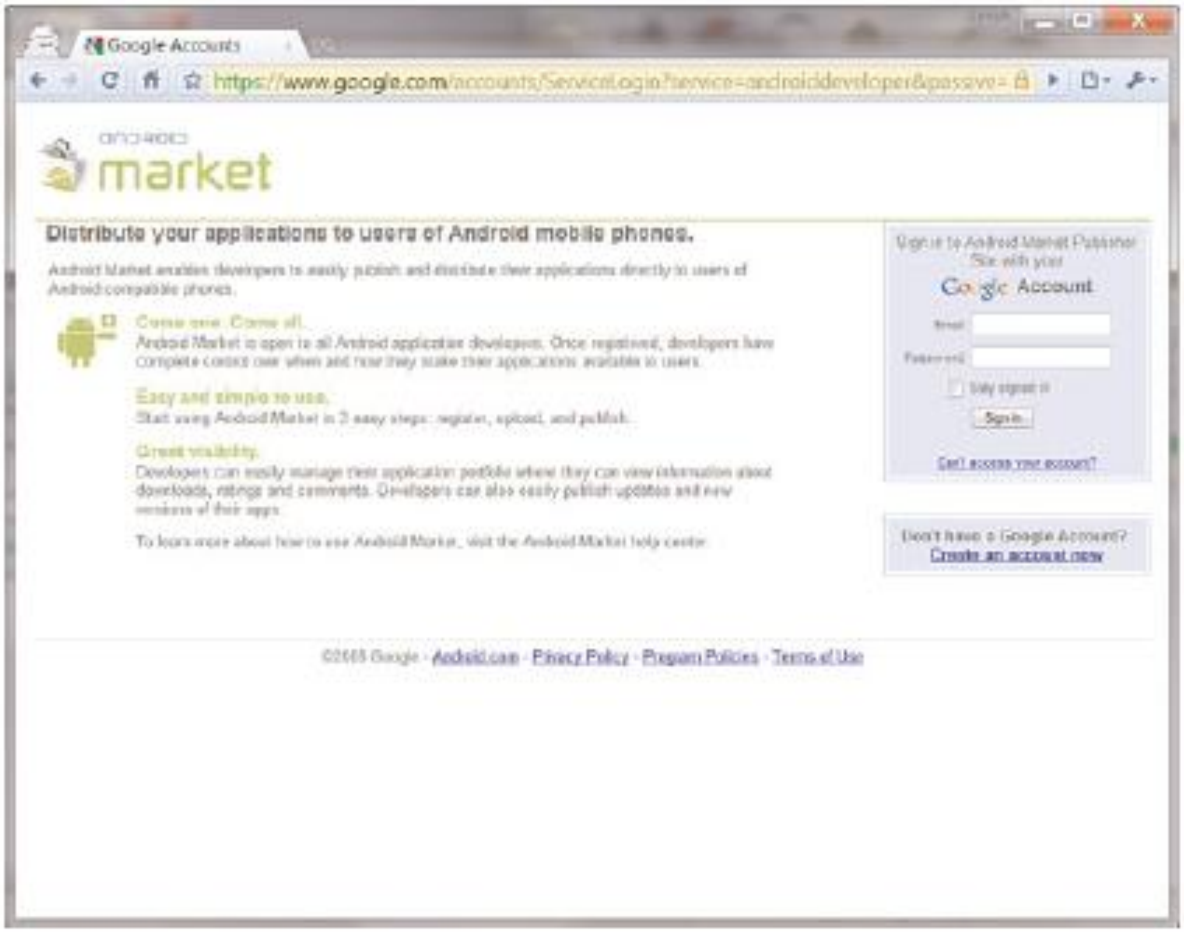
Signing Up for a Developer Account on the Android Market

To publish applications through the Android Market, you must register as a developer. This accomplishes two things. It verifies who you are to Google and signs you up for a Google Checkout account, which is used for billing of Android applications.

To sign up for an Android Market developer account, you need to follow these steps:

1. Go to the Android Market sign-up website, as shown in Figure.
2. Sign in with the Google Account you want to use. (At this time, you cannot change the associated Google Account, but you can change the contact email addresses for applications independently.)
3. Enter your developer information, including your name, email address, and website,as shown in Figure.
4. Confirm your registration payment (as of this writing, $25 USD). Note that Google Checkout is used for registration payment processing.
5. Signing up and paying to be an Android Developer also creates a mandatory Google Checkout Merchant account for which you also need to provide information. This account is used for payment processing purposes.
6. Agree to link your credit card and account registration to the Android Market Developer Distribution Agreement. The basic agreement (U.S. version) is available for review Always print out the actual agreement you sign as part of the registration process, in case it changes in the future.

The Android Market publisher sign-up page.


Google Accounts
https://www.google.com/accounts/ServiceLogin?service=androiddeveloper&passive=

Distribute your applications to users of Android mobile phones.

Android Market enables developers to easily publish and distribute their applications directly to users of Android-compatible phones.

Come one. Come all.
Android Market is open to all Android application developers. Once registered, developers have complete control over when and how they make their applications available to users.

Easy and simple to use.
Start using Android Market in 3 easy steps: register, upload, and publish.

Great visibility.
Developers can easily manage their application portfolio where they can view information about downloads, ratings and comments. Developers can also easily publish updates and new versions of their apps.

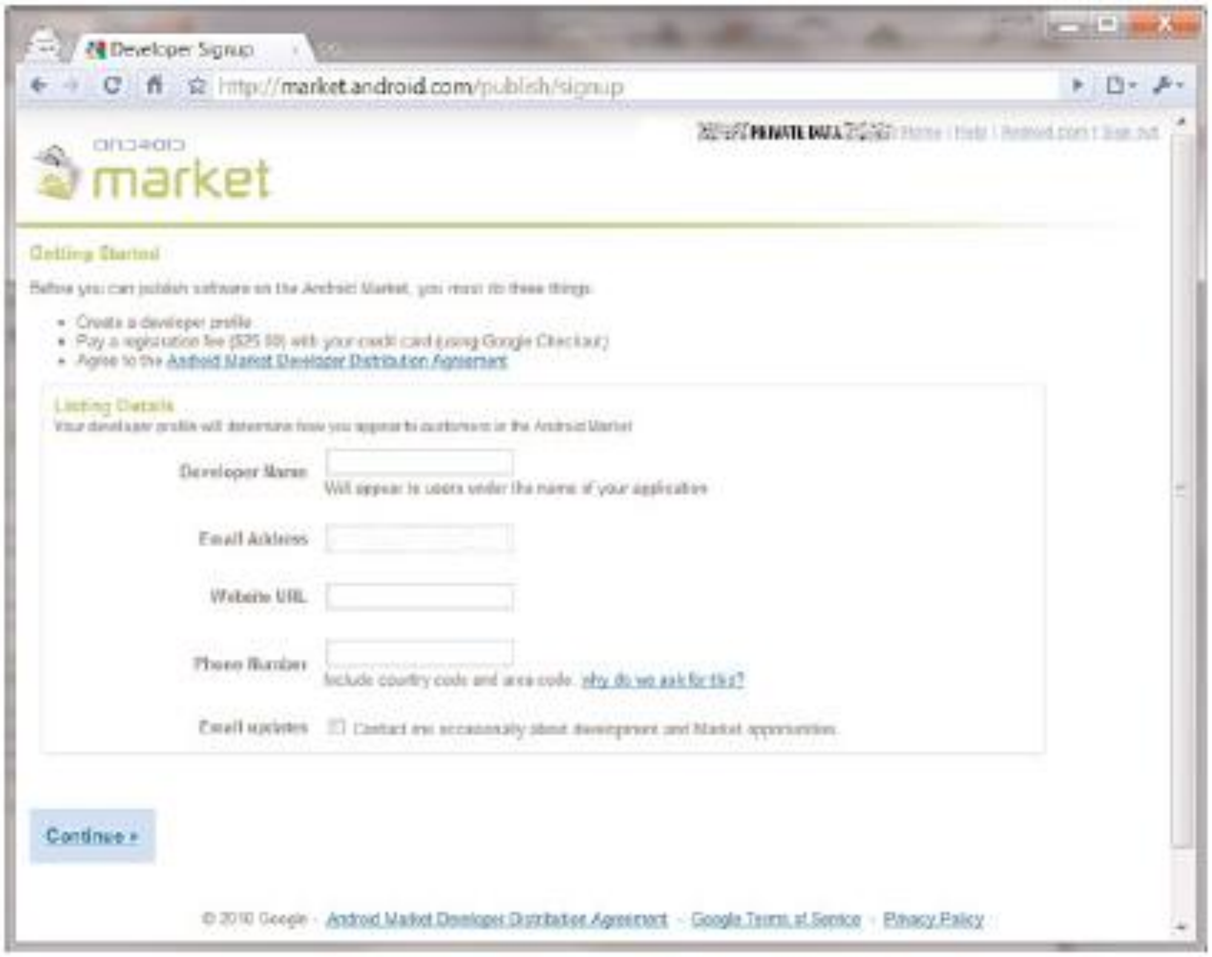
To learn more about how to use Android Market, visit the Android Market help center.

Sign in to Android Market Publisher Site with your
Google Account
Email
Password
Stay signed in
Sign in
Can't access your account?

Don't have a Google Account?
Create an account now

©2008 Google · Android.com · Privacy Policy · Program Policies · Terms of Use

The Android Market publisher profile page.


Developer Signup

http://market.android.com/publish/signup

PRIVATE DATA Home | Help | Android.com | Sign out

Getting Started

Before you can publish software on the Android Market, you must do three things:

- Create a developer profile
- Pay a registration fee ($25.00) with your credit card (using Google Checkout)
- Agree to the Android Market Developer Distribution Agreement

Listing Details

Your developer profile will determine how you appear to customers in the Android Market

Developer Name: Will appear to users under the name of your application

Email Address:

Website URL:

Phone Number: Include country code and area code. why do we ask for this?

Email updates: Contact me occasionally about development and Market opportunities.

Continue »

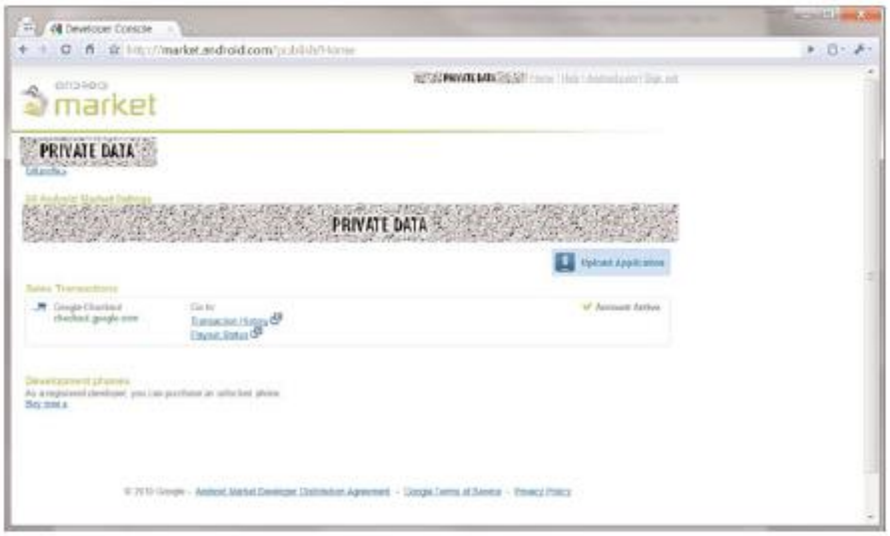
© 2010 Google - Android Market Developer Distribution Agreement - Google Terms of Service - Privacy Policy

When you successfully complete these steps, you are presented with the home screen of the Android Market, which also confirms that the Google Checkout Merchant account was created.

Uploading Your Application to the Android Market

Now that you have an account registered for publishing applications through Android Market and a signed application package, you are ready to upload it for publication. From the main page of the Android Market website, sign in with your developer account information. After you are logged in, you see a webpage with your developer account information, as shown in Figure.

Android Market developer application listings.

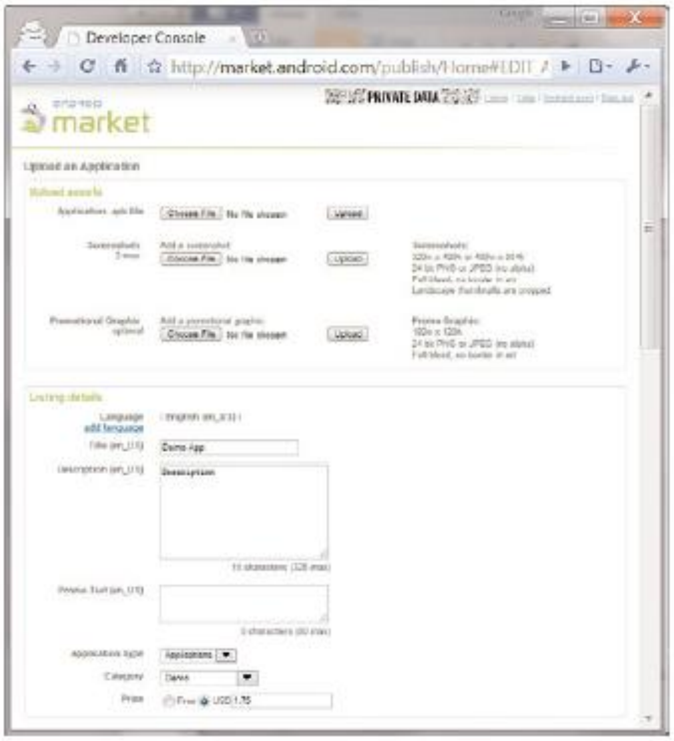


From this page, you can configure developer account settings, see your payment transaction history, and manage your published applications. In order to publish a new application, press the Upload Application button on this page. A form is presented for uploading the application package.

Let's look at some of the important fields you must enter on this form:

- The application package file (.apk)
- Promotional screenshots and graphics for the market listing
- The application title and description in several languages
- Application type and category

Android Market application upload form.



Developer Console

http://market.android.com/publish/Home#EDIT

PRIVATE DATA

market

Upload an Application

Upload assets

Application .apk file — Choose File No file chosen — Upload

Screenshots 2 max — Add a screenshot — Choose File No file chosen — Upload

Screenshots:
320w x 480h or 480w x 854h
24 bit PNG or JPEG (no alpha)
Full bleed, no border in art
Landscape thumbnails are cropped

Promotional Graphic optional — Add a promotional graphic — Choose File No file chosen — Upload

Promo Graphic:
180w x 120h
24 bit PNG or JPEG (no alpha)
Full bleed, no border in art

Listing details

Language: (English (en_US)) add language

Title (en_US): Demo App

Description (en_US): Description

11 characters (325 max)

Promo Text (en_US):

0 characters (80 max)

Application type: Applications

Category: Games

Price: Free / USD 1.75

- Copy protection information—Choosing this option might help prevent the application from being copied from the device and distributed without your knowledge or permission. This option is likely to change in the near future, as Google adds a new licensing service.
- Locations to distribute to—Choose the countries where the application should be published.
- Support contact information—This option defaults to the information you provided for the developer account. You can change it on an app-by-app basis, though, which allows for great support flexibility when you're publishing multiple applications.
- Consent—You must click the checkboxes to agree to the terms of the current (at the time you click) Android Content Guidelines, as well as the export laws of the United States, regardless of your location or nationality.

Publishing Your Application on the Android Market

Finally, you are ready to press the Publish button. Your application appears in the Android Market almost immediately. After publication, you can see statistics including ratings, reviews, downloads, and active installs in the Your Android Market Listings section of the main page on your developer account. These statistics aren't updated as frequently as the publish action is, and you can't see review details directly from the listing. Clicking on the application listing enables you to edit the various fields.

Understanding the Android Market Application Return Policy

Although it is a matter of no small controversy, the Android Market has a 24-hour refund policy on applications. That is to say, a user can use an application for 2 hours and then return it for a full refund. As a developer, this means that sales aren't final until after the first 24 hours. However, this only applies to the first download and first return. If a particular user has already returned your application and wants to "try it again," he or she must make a final purchase—and can't retur it a second time. Although this limits abuse, you should still be aware that if your application has limited reuse appeal or if all its value can come from just a few hours (or less) of use, you might find that you have a return rate that's too high and need to pursue other methods of monetization.

Upgrading Your Application on the Android Market

You can upgrade existing applications from the Market from the developer account page. Simply upload a new version of the same application using the Android manifest file tag, android:versionCode. When you publish it, users receive an Update Available notification, prompting them to download the upgrade.

Removing Your Application from the Android Market

You can also use the unpublish action to remove the application from the Market from the developer account. The unpublish action is also immediate, but the application entry on the Market application might be cached on handsets that have viewed or downloaded the application.

Using Other Developer Account Benefits

In addition to managing your applications on the Android Market, an additional benefit to have a registered Android developer account is the ability to purchase development versions of Android handsets. These handsets are useful for general development and testing but might not be suitable for final testing on actual target handsets because some functionality might be limited, and the firmware version might be different than that found on consumer handsets.

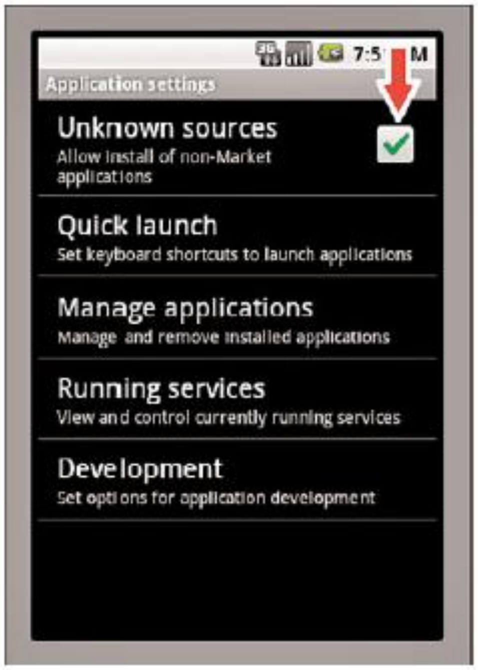
Selling Your Application on Your Own Server

You can distribute Android applications directly from a website or server. This method is most appropriate for vertical market applications, content companies developing mobile marketplaces, and big brand websites wanting to drive users to their branded Android applications. It can also be a good way to get beta feedback fro end users.

Although self-distribution is perhaps the easiest method of application distribution, it might also be the hardest to market, protect, and make money. The only requirement for self-distribution is to have a place to host the application package file.

The downside of self-distribution is that end users must configure their devices to allow packages from unknown sources. This setting is found under the Application section of the device Settings application, as shown in Figure. This option is not available on all consumer devices in the market. Most notably, Android devices on U.S. carrier AT&T can only install applications from the Android Market—no third-party sources are allowed.

Settings application showing required check box for downloading from unknown sources.



Application settings

Unknown sources
Allow install of non-Market applications ✓

Quick launch
Set keyboard shortcuts to launch applications

Manage applications
Manage and remove installed applications

Running services
View and control currently running services

Development
Set options for application development

After that, the final step the user must make is to enter the URL of the application package in to the web browser on the handset and download the file (or click on a link to it).When downloaded, the standard Android install process occurs, asking the user to confirm the permissions and, optionally, confirm an update or replacement of an existing application if a version is already installed.

Selling Your Application Using Other Alternatives

The Android Market is not the only consolidated market available for selling Android applications. Android is an open platform, which means there is nothing preventing a handset manufacturer or an operator (or even you) from running an Android market website or building another Android application that serves as a market. Many of the mobile focused stores, such as Handango, have been adding Android applications to their offerings.

Here are a few alternate marketplaces where you might consider distributing your Android applications:

- **Handango** distributes mobile applications across a wide range of devices with various billing models .
- **SlideME** is an Android-specific distribution community for free and commercial applications using an on-device store .
- **AndAppStore** is an Android-specific distribution for free applications using an ondevice store .
- **MobiHand** distributes mobile applications for a wide range of devices for free and commercial applications.

This list is not complete, nor do we endorse any of these markets. That said, we feel it is important to demonstrate that there are a number of alternate distribution mechanisms available to developers. Application requirements vary by store. Third-party application stores are free to enforce whatever rules they want on the applications they accept, so read the fine print carefully. They might enforce content guidelines, require additional technical support, and enforce digital signin requirements. Only you and your team can determine which are suitable for your specific needs.

Protecting Your Intellectual Property

You've spent time, money, and effort to build a valuable Android application. Now you want to distribute it but perhaps you are concerned about reverse engineering of trade secrets and software piracy. As technology rapidly advances, it's impossible to perfectly protect against both.

If you're accustomed to developing Java applications, you might be familiar with code obfuscation tools. These are designed to strip easy-to-read information from compiled Java byte codes making the decompiled application more difficult to understand. For Android, though, applications are compiled for the Dalvik virtual machine. As such, existing Java tools might not work directly and might need to be updated. Some tools, such as ProGuard , support Android applications because they can run after the jar file is created and before it's converted to the final package file used with Android.

Android Market supports a form of copy protection via a check box when you publish your application. The method that this uses isn't well documented currently. However, you can also use your own copy protection methods or those available through other markets if this is a huge concern for you or your company.

Billing the User

Unlike some other mobile platforms you might have used, Android does not currently provide built-in billing APIs that work directly from within applications or charge directly to the users' cell phone bill. Instead, Android Market uses Google checkout for processing payments. When an application is purchased, the user owns it (although any paid application can be returned within 24 hours for a full refund).

Billing Recurring Fees or Content-Specific Fees

If your application requires a service fee and sells other goods within the application (that is, ringtones, music, e-books, and more), the application developer must develop a custom billing mechanism. Most Android devices can leverage the Internet, so using online billing services and APIs—Paypal, Google, and Amazon, to name few—are likely to be the common choice. Check with your preferred billing service to make sure it specifically allows mobile use and that the billing methods your application requires are available, feasible, and legal for your target users.

Leveraging Ad Revenue

Another method to make money from users is to have an ad-supported mobile business model. This is a relatively new model for use within applications because many older application distribution methods specifically disallowed it. However, Android has no specific rules against using advertisements within applications. This shouldn’t come as too much of a surprise, considering the popularity of Google’s AdSense.