Mobile Application Development
MODULE1 PART1

ANDROID: AN OPEN PLATFORM FOR MOBILE DEVELOPMENT

Android is a combination of three components:

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

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

- ➢ A hardware reference design that describes the capabilities required for a mobile device to support the software stack.
- ➢ A Linux operating system kernel that provides low-level interface with the hardware, memory management, and process control, all optimized for mobile devices.
- ➢ Open-source libraries for application development, including SQLite, WebKit, OpenGL, and a media manager.
- ➢ A run time used to execute and host Android applications, including the Dalvik virtual machine and the core libraries that provide Android-specific functionality. The run time is designed to be small and efficient for use on mobile devices.
- ➢ An application framework that agnostically exposes system services to the application layer, including the window manager and location manager, content providers, telephony, and sensors.
- ➢ A user interface framework used to host and launch applications.
- ➢ Preinstalled applications shipped as part of the stack.
- ➢ A software development kit used to create applications, including tools, plug-ins, and documentation.

NATIVE ANDROID APPLICATIONS

Android phones will normally come with a suite of generic preinstalled applications that are part of the Android Open Source Project (AOSP), including, but not necessarily limited to:

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

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

- ➢ The Android Market client for downloading third-party Android applications
- ➢ A fully-featured mobile Google Maps application including StreetView, driving directions and turn-by-turn navigation, satellite view, and traffic conditions
- ➢ The Gmail mail client
- ➢ The Google Talk instant-messaging client
- ➢ The YouTube video player

The data stored and used by many of these native applications — like contact details — are also available to third-party applications

ANDROID SDK FEATURES

.

The following list highlights some of the most noteworthy Android features:

- ➢ No licensing, distribution, or development fees or release approval processes
- ➢ Wi-Fi hardware access
- ➢ GSM, EDGE, and 3G networks for telephony or data transfer, enabling you to make or receive calls or SMS messages, or to send and retrieve data across mobile networks
- ➢ Comprehensive APIs for location-based services such as GPS
- ➢ Full multimedia hardware control, including playback and recording with the camera and

microphone

- ➢ APIs for using sensor hardware, including accelerometers and the compass
- ➢ Libraries for using Bluetooth for peer-to-peer data transfer
- ➢ IPC message passing
- ➢ Shared data stores
- ➢ Background applications and processes
- ➢ Home-screen Widgets, Live Folders, and Live Wallpaper
- ➢ The ability to integrate application search results into the system search
- ➢ An integrated open-source HTML5 WebKit-based browser
- ➢ Full support for applications that integrate map controls as part of their user interface
- ➢ Mobile-optimized hardware-accelerated graphics, including a path-based 2D graphics library and support for 3D graphics using OpenGL ES 2.0

➢ Media libraries for playing and recording a variety of audio/video or still image formats

➢ Localization through a dynamic resource framework

➢ An application framework that encourages reuse of application components and the replacement of native applications

WHY DEVELOP FOR ANDROID?

From a commercial perspective Android:

➢ Requires no certification for becoming an Android developer

➢ Provides the Android Market for distribution and monetization of your applications

➢ Has no approval process for application distribution

➢ Gives you total control over your brand and access to the user's home screen

What Does It Have That Others Don't?

Many of the features listed previously, such as 3D graphics and native database support, are also available in other mobile SDKs. Here are some of the unique features that set Android apart:

➢ **Google Map applications** Google Maps for Mobile has been hugely popular, and Android offers a Google Map as an atomic, reusable control for use in your applications. The Map View lets you display, manipulate, and annotate a Google Map within your Activities to build map-based applications using the familiar Google Maps interface.

➢ **Background services and applications** Background services let you create an application that uses an event-driven model, working silently while other applications are being used or while your mobile sits ignored until it rings, flashes, or vibrates to get your attention. Maybe it's a streaming music player, an application that tracks the stock market, alerting you to significant changes in your portfolio, or a service that changes your ringtone or volume depending on your current location, the time of day, and the identity of the caller.

- **Shared data and interprocess communication** Using Intents and Content Providers, Android lets your applications exchange messages, perform processing, and share data. You can also use these mechanisms to leverage the data and functionality provided by the native Android applications.
- **All applications are created equal** Android doesn't differentiate between native applications and those developed by third parties. This gives consumers unprecedented power to change the look and feel of their devices by letting them completely replace every native application with a third-party alternative that has access to the same underlying data and hardware.
- **Home-screen Widgets, Live Folders, Live Wallpaper, and the quick search box** Using Widgets, Live Folders, and Live Wallpaper, you can create windows into your application from the phone's home screen. The quick search box lets you integrate search results from your application directly into the phone's search functionality.

What Comes in the Box

The Android software development kit (SDK) includes everything you need to start developing, testing, and debugging Android applications. Included in the SDK download are:

- **The Android APIs** The core of the SDK is the Android API libraries that provide developer access to the Android stack. These are the same libraries used at Google to create native Android applications.
- **Development tools** So you can turn Android source code into executable Android applications, the SDK includes several development tools that let you compile and debug your applications.
- **The Android Virtual Device Manager and Emulator** The Android Emulator is a fully interactive Android device emulator featuring several alternative skins. The emulator runs within an Android Virtual Device that

simulates the device hardware configuration. Using the emulator you can see how your applications will look and behave on a real Android device. All Android applications run within the Dalvik VM, so the software emulator is an excellent environment — in fact, as it is hardware-neutral, it provides a better independent test environment than any single hardware implementation.

➢ **Full documentation** The SDK includes extensive code-level reference information detailing exactly what's included in each package and class and how to use them. In addition to

the code documentation, Android's reference documentation explains how to get started and gives detailed explanations of the fundamentals behind Android development.

➢ **Sample code** The Android SDK includes a selection of sample applications that demonstrate some of the possibilities available with Android, as well as simple programs that highlight how to use individual API features.

➢ **Online support** Android has rapidly generated a vibrant developer community. Android engineering and developer relations teams at Google. StackOverflow has also become a popular destination for Android questions.

Understanding the Android Software Stack

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

➢ Libraries Running on top of the kernel, Android includes various C/C++ core libraries such as libc and SSL, as well as:

➢ A media library for playback of audio and video media

➢ A surface manager to provide display management

➢ Graphics libraries that include SGL and OpenGL for 2D and 3D graphics

➢ SQLite for native database support

➢ SSL and WebKit for integrated web browser and Internet security

➢ Android run time Linux implementation is the Android run time. Including the core libraries and the Dalvik virtual machine, the Android run time is the engine that powers your applications and, along with the libraries, forms the basis for the application framework.

➢ Core libraries While Android development is done in Java, Dalvik is not a Java VM. The core Android libraries provide most of the functionality available in the core Java libraries as well as the Android-specific libraries.

- Dalvik virtual machine Dalvik is a register-based virtual machine that's been optimized to ensure that a device can run multiple instances efficiently. It relies on the Linux kernel for threading and low-level memory management.

➢ Application framework The application framework provides the classes used to create Android applications. It also provides a generic abstraction for hardware access and manages the user interface and application resources.

➢ Application layer All applications, both native and third-party, are built on the application layer by means of the same API libraries. The application layer runs within the Android run time, using the classes and services made available from the application framework.

- At the bottom of the layers is Linux - Linux 3.6 with approximately 115 patches.
- This provides a level of abstraction between the device hardware and it contains all the essential hardware drivers like camera, keypad, display etc.
- The kernel handles all the things that Linux is really good at such as networking and a vast array of device drivers, for interfacing to peripheral hardware.
- **android.app** − Provides access to the application model and is the cornerstone of all Android applications.
- **android.content** − Facilitates content access, publishing and messaging between applications and application components.

- **android.database** − Used to access data published by content providers and includes SQLite database management classes.
- **android.opengl** − A Java interface to the OpenGL ES 3D graphics rendering API.
- **android.os** − Provides applications with access to standard operating system services including messages, system services and inter-process communication.
- **android.text** − Used to render and manipulate text on a device display.
- **android.view** − The fundamental building blocks of application user interfaces.
- **android.widget** − A rich collection of pre-built user interface components such as buttons, labels, list views, layout managers, radio buttons etc.
- **android.webkit** − A set of classes intended to allow web-browsing capabilities to be built into applications.

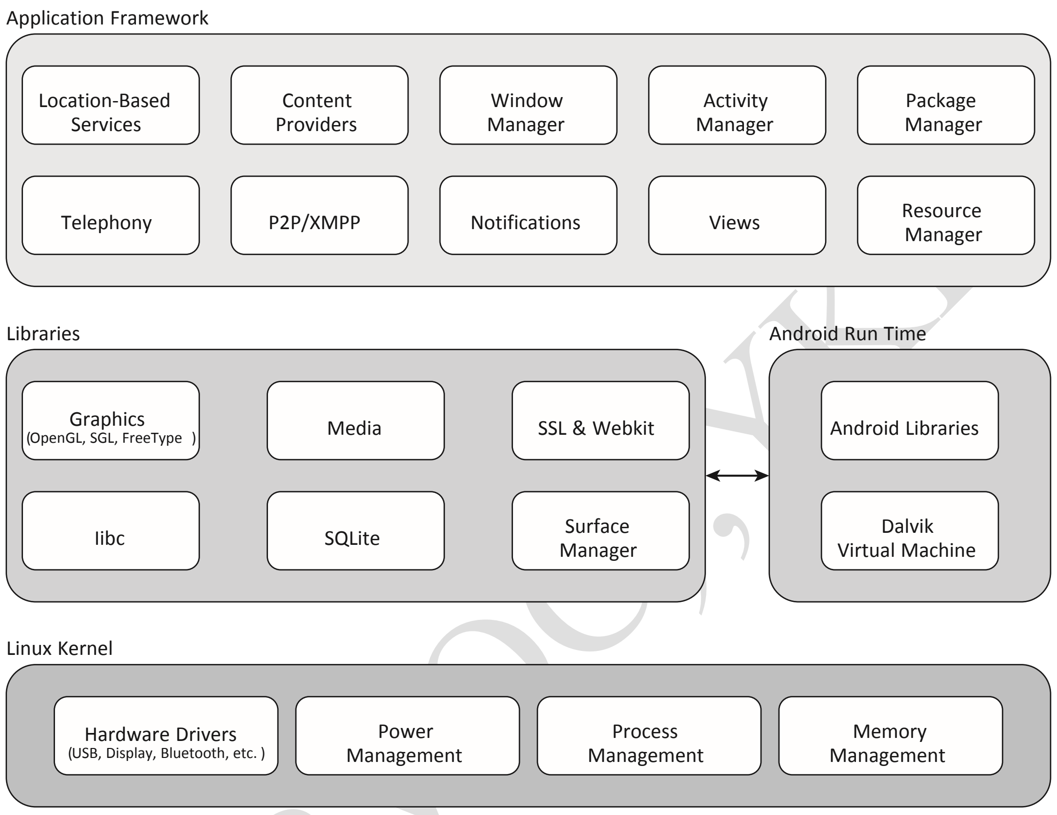


FIGURE 1-1

➢ **Application framework** the application framework provides the classes used to create Android applications. It also provides a generic abstraction for access and manages the user interface and application resources.

➢ **Application layer** All applications, both native and third-party, are built on the application layer by means of the same API libraries. The application layer runs within the Android run time, using the classes and services made available from the application framework.

Android Runtime

Android Runtime provides a key component called **Dalvik Virtual Machine** which is a kind of Java Virtual Machine specially designed and optimized for Android.

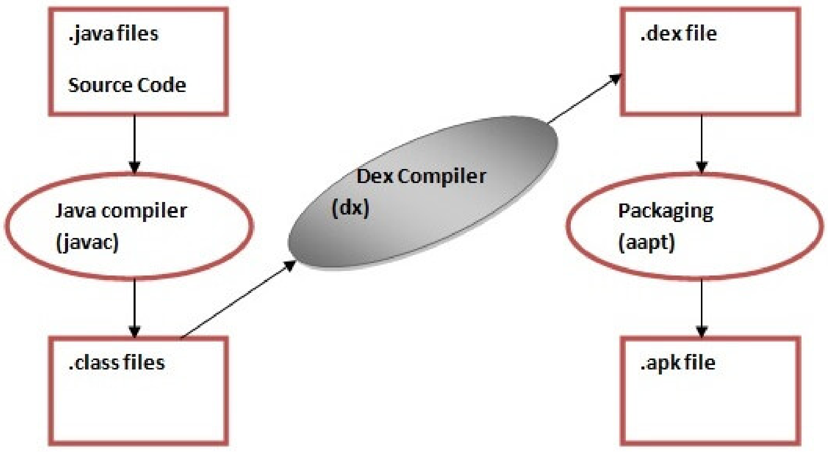
The Dalvik VM makes use of Linux core features like memory management and multi-threading, which is intrinsic in the Java language.

The Dalvik VM enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine.

The Android runtime also provides a set of core libraries which enable Android application developers to write Android applications using standard Java programming language.

Dalvik Virtual Machine | DVM

- Modern JVM provides high performance and excellent memory management.
- The **Dalvik Virtual Machine (DVM)** is an android virtual machine optimized for mobile devices.
- It optimizes the virtual machine for *memory*, *battery life* and *performance*.
- Dalvik is a name of a town in Iceland. The Dalvik VM was written by Dan Bornstein



- The Dex compiler converts the class files into the .dex file that run on the **DVM.**

- Multiple class files are converted into one dex file.
- The **javac tool** compiles the java source file into the class file.
- The **dx tool** takes all the class files of your application and generates a single .dex file. It is a platform-specific tool.
- The **Android Assets Packaging Tool (aapt)** handles the packaging process.

JVM(Java Virtual Machine)	**DVM(Dalvik Virtual Machine)**
Stack-based VM that performs arithmetic and logic operations through push and pop operands. The result of operations is stored in stack memory.	Register-based VM that uses registers located in the CPU to perform arithmetic and logic operations.
Java source code is compiled into Java bytecode format(.class file) that further translates into machine code.	Source code files are first of all compiled into Java bytecode format like JVM. Further, the **DEX compiler(dx tool)** converts the Java bytecode into Dalvik bytecode(classes.dex) file that will be used to create the **.apk file**.
More information is required to the VM for data loading and manipulation as well as method loading in the stack data structure.	Instruction size is larger as it needs to encode the source and destination register of the VM.
Compiled bytecode size is compact because the location of the operand is implicitly on the operand stack.	Compiled bytecode size is larger as each instruction needs all implicit operands.
The executable file for the device is **.jar file**.	The executable file for the device is **.apk file**.
A single instance of JVM is configured with shared processes and memory space in order to run all deployed applications.	The device runs multiple DVM instances with a separate process in shared memory space to deploy the code of each application.
Supports multiple operating systems like Linux, Windows, and Mac OS.	Support only the Android operation system.

Android Application Architecture

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

The same mechanism that lets you produce a replacement contact manager or phone dialer can let you expose your application components to let other developers create new UI front ends and functionality extensions, or otherwise build on them.

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

- **Activity Manager**
- **Views**
- **Notification Manager**
- **Content Providers**
- **Resource Manager**

Android Libraries

Android offers a number of APIs for developing your applications.

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

What You Need to Begin

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

- Microsoft Windows (XP or later)
- Mac OS X 10.4.8 or later (Intel chips only)
- Linux

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

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

Downloading and Installing the SDK

The Android SDK is completely open. There's no cost to download or use the API, and Google doesn't charge (or require review) to distribute your finished programs on the Android Market or otherwise.

The SDK is presented as a ZIP file containing only the latest version of the Android developer tools. Install it by unzipping the SDK into a new folder. (Take note of this location, as you'll need it later.)

Before you can begin development, you need to add at least one SDK Platform; do this on Windows by running the ''SDK Setup.exe'' executable, or on MacOS or Linux by running the ''android'' executable in the tools subfolder. In the screen that appears, select the ''Available Packages'' option on the left panel, and then select the SDK Platform versions you wish to install in the ''Sources, Packages, and Archives'' panel on the right. The selected platform will then be downloaded to your SDK installation folder and will contain the API libraries, documentation, and several sample applications.Android Developer Tool (ADT) plug-in.

Understanding Hello World

Let's take a step back and have a real look at your first Android application.

Activity is the base class for the visual, interactive components of your application; it is roughly equivalent to a Form in traditional desktop development. The skeleton code for an Activity-based class; note that it extends Activity, overriding the onCreate method.

```
Package com.android.helloworld;

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

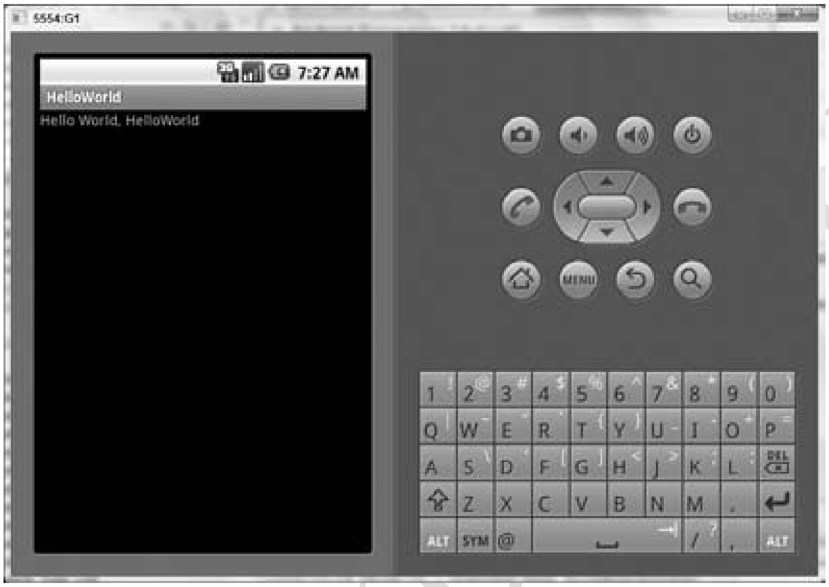


FIGURE 2-7

What's missing from this template is the layout of the visual interface. In Android, visual components are called *Views*, which are similar to controls in traditional desktop development.

The Hello World template created by the wizard overrides the onCreate method to call setContentView, which lays out the user interface by inflating a layout resource, as highlighted

below:

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

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

```
<?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="Hello World, HelloWorld"
/>
</LinearLayout>
```

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

To get access to your UI elements in code, you add identifier attributes to them in the XML definition.

You can then use the findViewById method to return a reference to each named item. The following XML snippet shows an ID attribute added to the Text View widget in the Hello World template:

```
<TextView
android:id="@+id/myTextView"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Hello World,
HelloWorld" />
```

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

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

All the properties available in code can be set with attributes in the XML layout. As well as allowing easier substitution of layout designs and individual UI elements, keeping the visual design decoupled from the application code helps keep the code more concise.

Types of Android Applications

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

- **Foreground** An application that's useful only when it's in the foreground and is effectively suspended when it's not visible. Games and map mashups are common examples.
- **Background** An application with limited interaction that, apart from when being configured, spends most of its lifetime hidden. Examples include call screening applications and SMS auto-responders.
- **Intermittent** Expects some interactivity but does most of its work in the background. Often these applications will be set up and then run silently, notifying users when appropriate. A common example would be a media player.
- **Widget** Some applications are represented only as a home-screen widget.

Complex applications are difficult to pigeonhole into a single category and usually include elements of each of these types. When creating your application you need to consider how it's likely to be used and then design it accordingly.

Foreground Applications

Applications have little control over their life cycles, and a background application with no running Services is a prime candidate for cleanup by Android's resource management. This means that you need to save the state of the application when it is no longer in the foreground, to let you present the exact same state when it is brought to the front.

It's also particularly important for foreground applications to present a slick and intuitive user experience.

Background Services and Intent Receivers

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

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

Intermittent Applications

Often you'll want to create an application that reacts to user input but is still useful when it's not active in the foreground. Chat and e-mail apps are typical examples. These applications are generally a union of visible Activities and invisible background Services.

Widgets

In some circumstances your application may consist entirely of a widget component.

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

DEVELOPING FOR MOBILE DEVICES

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

Hardware-Imposed Design Considerations

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

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

- ➢ Low processing power
- ➢ Limited RAM
- ➢ Limited permanent storage capacity
- ➢ Small screens with low resolution
- ➢ High costs associated with data transfer
- ➢ Slow data transfer rates with high latency
- ➢ Unreliable data connections
- ➢ Limited battery life

Each new generation of phones improves many of these restrictions. In particular, newer phones have dramatically improved screen resolutions and significantly cheaper data tariffs. However, given the range of devices available, it is good practice to design to accommodate the worst-case scenario.

Be Efficient

Manufacturers of embedded devices, particularly mobile devices, generally value small size and long battery life over potential improvements in processor speed. For developers, that means losing the head start traditionally afforded thanks to Moore's law (the doubling of the number of transistors placed on an integrated circuit every two years). In desktop and server hardware this usually results directly in processor performance improvements; for mobile devices it instead means smaller, more power-efficient mobiles without significant improvement in processor power.

Expect Limited Capacity

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

Android devices offer an additional restriction in that applications must be installed on the internal memory (as opposed to external SD cards). As a result, the compiled size of your application is a consideration, though more important is ensuring that your application is polite in its use of system resources.

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

Part of being polite is cleaning up after yourself. Techniques like caching are useful for limiting repetitive network lookups, but don't leave files on the file system or records in a database when they're no longer needed.

Design for Small Screens

The small size and portability of mobiles are a challenge for creating good interfaces, particularly when users are demanding an increasingly striking and information-rich graphical user experience.

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

If you're planning to include touch-screen support (and if you're not, you should be), you'll need to consider how touch input is going to affect your interface design. The time of the stylus has passed; now it's all about finger input, so make sure your Views are big enough to support interaction using a finger on the screen.

Android phones are now available with a variety of screen sizes including QVGA, HVGA, and WVGA. As display technology advances, and Android expands beyond mobile devices, screen sizes and resolutions will continue to increase. To ensure that your app looks good and behaves well on all the possible host devices it's important to design for small screens, but also make sure your UIs scale well on larger displays.

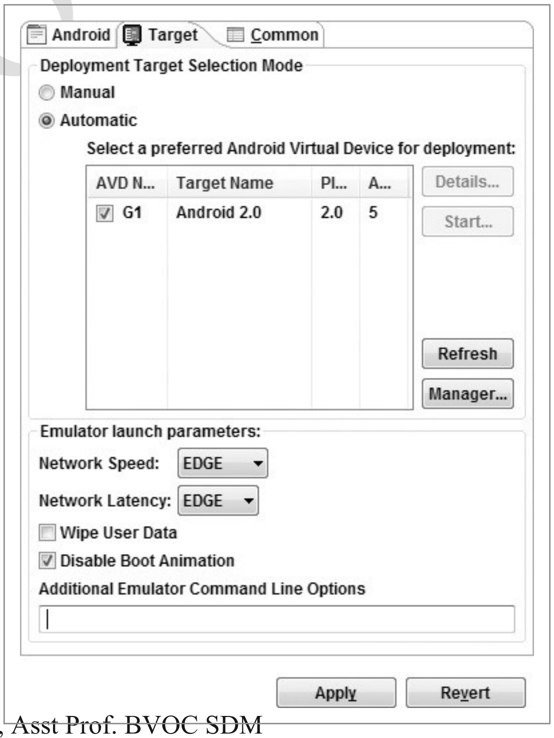
Expect Low Speeds, High Latency

The mobile Web unfortunately isn't as fast, reliable, or readily available as we'd often like, so when you're developing your Internet-based applications it's best to assume that the network connection will be slow, intermittent, and expensive. With unlimited 3G data plans and citywide Wi-Fi, this is changing, but designing for the worst case ensures that you always deliver a high-standard user experience.

This also means making sure that your applications can handle losing (or not finding) a data connection.

The Android Emulator lets you control the speed and latency of your network connection. Figure 2-8 shows the emulator's network connection speed and latency, simulating a distinctly suboptimal EDGE connection.

Experiment to ensure seamlessness and responsiveness no matter what the speed, latency, and availability of network access. In some circumstances you might find that it's better to limit the functionality of your application or reduce network lookups to cached bursts, based on the network connection(s) available.



Deployment Target Selection Mode
Manual [✓ Automatic]
Select a preferred Android Virtual Device for deployment:

AVD N...	Target Name	Pl...	A...
✓ G1	Android 2.0	2.0	5

Emulator launch parameters:
Network Speed: EDGE
Network Latency: EDGE
☐ Wipe User Data
☑ Disable Boot Animation
Additional Emulator Command Line Options

At What Cost?

If you're a mobile owner, you know all too well that some of the more powerful features on your mobile can literally come at a price. Services like SMS, some location-based services, and data transfer can sometimes incur an additional tariff
from your service provider. **FIGURE 2-8**

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

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

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

➢ Transferring as little data as possible

➢ Caching data and GPS results to eliminate redundant or repetitive lookups

➢ Stopping all data transfers and GPS updates when your activity is not visible in the foreground and if they're only being used to update the UI

➢ Keeping the refresh/update rates for data transfers (and location lookups) as low as practicable

➢ Scheduling big updates or transfers at ''off-peak'' times using alarms

➢ Respecting the user's preferences for background data transfer

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

When using the location-based services, you can select a location provider based on whether there is an associated cost. Within your location-based applications, consider giving users the choice of lower cost or greater accuracy.

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

Rather than enforcing a particular technique based on which seems cheaper, consider letting your users choose. For example, when downloading data from the Internet, you could ask users if they want to use any network available or limit their transfers to times when they're connected via Wi-Fi.

Considering the Users' Environment

While Android is already starting to expand beyond its core base as a mobile phone platform, most Android devices are still mobile phones. Remember that for most people, such a device is first and foremost a phone, secondly an SMS and email communicator, thirdly a camera, and fourthly an MP3 player. The applications you write will most likely be in the fifth category of ‘‘useful mobile tools.’’

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

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

What does this mean in terms of software design? Make sure that your application:

- **Is well behaved** Start by ensuring that your Activities suspend when they're not in the foreground. Android triggers event handlers when your Activity is suspended or resumed so you can pause UI updates and network lookups when your application isn't visible — there's no point updating your UI if no one can see it. If you need to continue updating or processing in the background, Android provides a Service class designed to run in the background without the UI overheads.
- **Switches seamlessly from the background to the foreground** With the multitasking nature of mobile devices, it's very likely that your applications will regularly move into and out of the background. It's important that they ''come to life'' quickly and seamlessly. Android's nondeterministic process management means that if your application is in the background, there's every chance it will get killed to free resources. This should be invisible to the user. You can ensure seamlessness by saving the application state and queuing updates so that your users don't notice a difference between restarting and resuming your application. Switching back to it should be seamless, with users being shown the exact UI and application state they last saw.
- **Is polite** Your application should never steal focus or interrupt a user's current activity. Use Notifications and Toasts instead to inform or remind users that their attention is requested, if your application isn't in the foreground. There are several ways for mobile devices to alert users. For example, when a call is coming in, your phone rings; when you have unread messages, the LED flashes; and when you have new voice mail, a small ''mail'' icon appears in your status bar. All these techniques and more are available through the notification mechanism.
- **Presents a consistent user interface** Your application is likely to be one of several in use at any time, so it's important that the UI you present is easy to use. Don't force users to interpret and relearn your application every time they load it. Using it should be simple,

easy, and obvious — particularly given the limited screen space and distracting user environment.

➢ **Is responsive** Responsiveness is one of the most important design considerations on a mobile device. You've no doubt experienced the frustration of a ''frozen'' piece of software; the multifunctional nature of a mobile makes this even more annoying. With the possibility of delays caused by slow and unreliable data connections, it's important that your application use worker threads and background services to keep your activities responsive and, more importantly, to stop them from preventing other applications from responding promptly.

Developing for Android

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

The Android design philosophy demands that applications be

designed for:

➢ Performance

➢ Responsiveness

➢ Seamlessness

➢ Security

Being Fast and Efficient

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

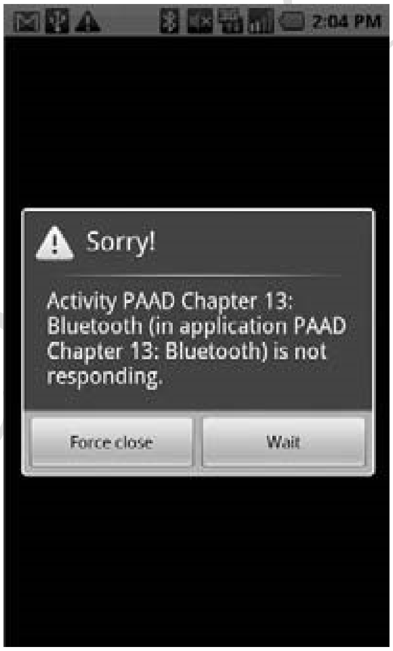
One of the keys to writing efficient Android code is not to carry over assumptions from desktop and server environments to embedded devices.

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

Being Responsive

Android takes responsiveness very seriously.

Android enforces responsiveness with the Activity Manager and Window Manager. If either service detects an unresponsive application, it will display the dreaded ''Sorry! Activity is not responding'' message — often reported by users as a *Force Close* error. This is shown in Figure 2-9.


Sorry!

Activity PAAD Chapter 13: Bluetooth (in application PAAD Chapter 13: Bluetooth) is not responding.

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

Android monitors two conditions to determine responsiveness:

➢ An application must respond to any user action, such as a key press or screen touch, within five seconds.

➢ A Broadcast Receiver must return from its onReceive handler within 10 seconds.

The most likely culprits in cases of unresponsiveness are network lookups, complex processing (such as the calculating of game moves), and file I/O

Developing Secure Applications

Android applications have access to networks and hardware, can be distributed independently, and are built on an open-source platform featuring open communication, so it shouldn't be surprising that security is a significant concern.

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

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

- ➢ Require permissions for any Services you publish or Intents you broadcast.
- ➢ Take special care when accepting input to your application from external sources such as the Internet, Bluetooth, SMS messages, or instant messaging (IM).
- ➢ Be cautious when your application may expose access to lower-level hardware to third-party applications.

Ensuring a Seamless User Experience

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

The speed and responsiveness of a mobile device shouldn't degrade the longer it's on. Android's process management helps by acting as a silent assassin, killing background applications to free resources as required.

Knowing this, your applications should always present a consistent interface, regardless of whether they're being restarted or resumed.

With an Android device typically running several third-party applications written by different developers, it's particularly important that these applications interact seamlessly. Using Intents, applications can provide functionality for each other. Knowing your application may provide, or consume, thirdparty Activities provides additional incentive to maintain a consistent look and feel.

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

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

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

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

TO-DO LIST EXAMPLE

In this example you'll be creating a new Android application from scratch. This simple example creates a new to-do list application using native Android View controls. It's designed to illustrate the basic steps involved in starting a new project.

1. Start by creating a new Android project. select **File ➪ New ➪ Project...,** then choose **Android** (as shown in Figure 2-10) before clicking **Next.**

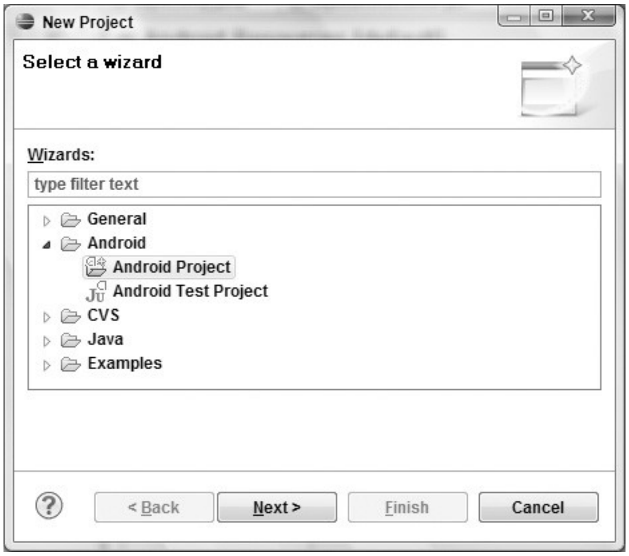
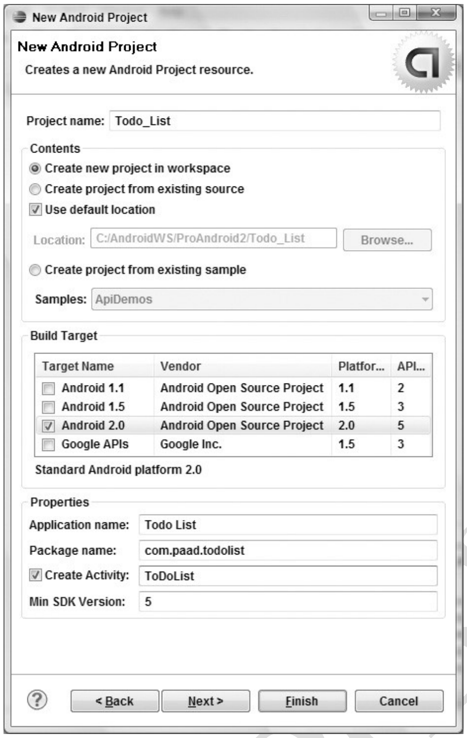

New Project
Select a wizard
Wizards:
type filter text
General
Android
Android Project
Android Test Project
CVS
Java
Examples
< Back
Next >
Finish
Cancel

FIGURE 2-10

2. In the dialog box that appears (shown in Figure 2-11), enter the details for your new project. The ''Application name'' is the friendly name of your application, and the ''Create Activity'' field lets you name your Activity. With the details entered, click **Finish** to create your new project.


New Android Project

New Android Project
Creates a new Android Project resource.

Project name: Todo_List

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

Location: C:/AndroidWS/ProAndroid2/Todo_List

- ( ) Create project from existing sample

Samples: ApiDemos

Build Target

	Target Name	Vendor	Platfor...	API...
☐	Android 1.1	Android Open Source Project	1.1	2
☐	Android 1.5	Android Open Source Project	1.5	3
✓	Android 2.0	Android Open Source Project	2.0	5
☐	Google APIs	Google Inc.	1.5	3

Standard Android platform 2.0

Properties

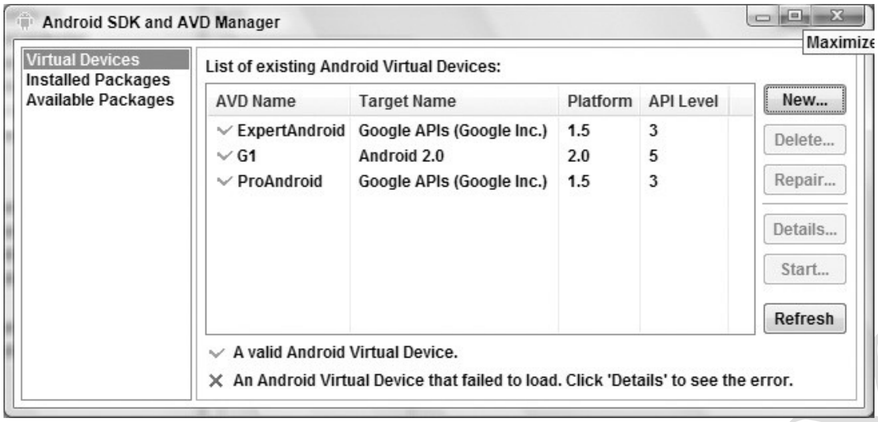
Application name:	Todo List
Package name:	com.paad.todolist
[✓] Create Activity:	ToDoList
Min SDK Version:	5

FIGURE 2-11

3. Before creating your debug and run configurations, take this opportunity to create a virtual device to test your apps with.

3.1. Select **Window ➪ Android SDK and AVD Manager.** In the resulting dialog (shown in Figure 2-12), select **Virtual Devices** from the left panel and click the **New**... button.

3.2. Enter a name for your device, and choose an SDK target and screen resolution. Set the SD Card size to larger than 8 MB: enter 12 into the text-entry box as shown in Figure 2-13.



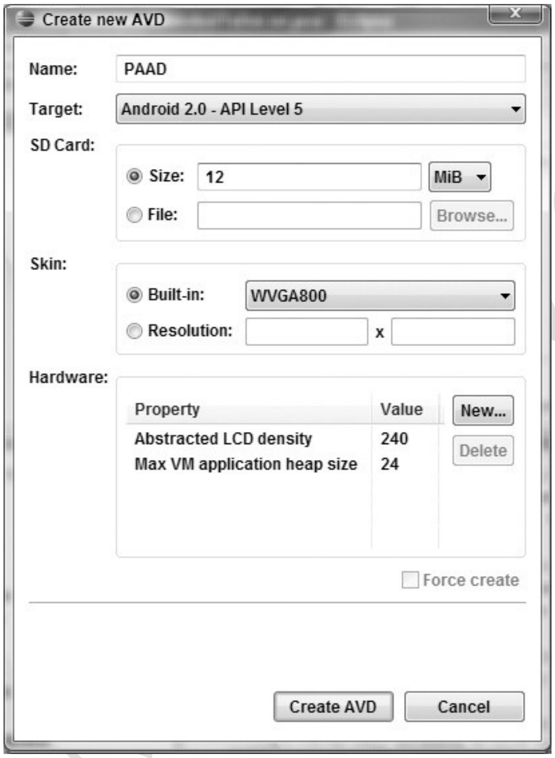
List of existing Android Virtual Devices:

AVD Name	Target Name	Platform	API Level
✓ ExpertAndroid	Google APIs (Google Inc.)	1.5	3
✓ G1	Android 2.0	2.0	5
✓ ProAndroid	Google APIs (Google Inc.)	1.5	3

✓ A valid Android Virtual Device.

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

FIGURE 2-12



Name: PAAD

Target: Android 2.0 - API Level 5

SD Card: (● Size: 12 MiB) ○ File:

Skin: (● Built-in: WVGA800) ○ Resolution: x

Hardware:

Property	Value
Abstracted LCD density	240
Max VM application heap size	24

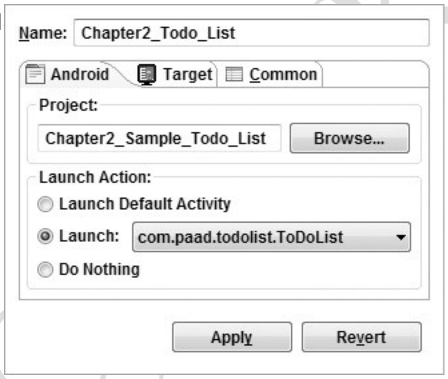
☐ Force create

FIGURE 2-13

4. Now create your debug and run configurations. Select **Run ➪ Debug Configurations**... and then **Run ➪ Run Configurations**..., creating a new

configuration for each specifying the Todo_List project and selecting the virtual device you created in Step 3. You can leave the launch action as **Launch Default Activity**, or explicitly set it to launch the new ToDoList Activity, as shown in Figure 2-14.

5. Now decide what you want to show the users and what actions they'll need to perform. Design a user interface that will make these actions as intuitive as possible.


Name: Chapter2_Todo_List

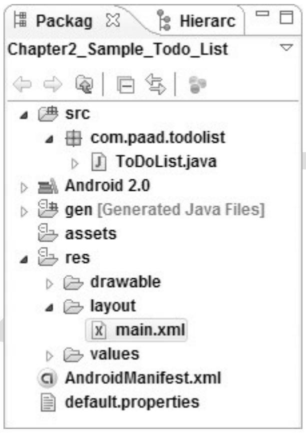
Project: Chapter2_Sample_Todo_List

Launch Action:
( ) Launch Default Activity
(●) Launch: com.paad.todolist.ToDoList
( ) Do Nothing

In this example we want to present users with a list of to-do items and a text entry box to add new ones. There's both a list and a text-entry control available from the Android libraries. The preferred method for laying out your UI is **FIGURE 2-14** using a layout resource file. Open the main.xml layout file in the res/layout project folder, as shown in Figure 2-15.

6. Modify the main layout to include a ListView and an EditText within a LinearLayout. It's important to give both the EditText and ListView an ID so you can get references to them both in code.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical" android:layout_width="fill_parent"
  android:layout_height="fill_parent">
  <EditText
    android:id="@+id/myEditText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content
    " android:text="New To Do Item"
  />
  <ListView android:id="@+id/myListView"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
  />
</LinearLayout>
```



```
Packag   Hierarc
Chapter2_Sample_Todo_List
src
  com.paad.todolist
    ToDoList.java
Android 2.0
gen [Generated Java Files]
assets
res
  drawable
  layout
    main.xml
  values
AndroidManifest.xml
default.properties
```

FIGURE 2-15

7. With your user interface defined, open the ToDoList Activity from your project's source folder. In this example you'll make all your changes by overriding the onCreate method. Start by inflating your UI using setContentView and then get references to the ListView and EditText using findViewById.

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

  // Inflate your view
     setContentView(R.layout.main);
  // Get references to UI widgets
   ListView myListView = (ListView)findViewById(R.id.myListView);
final EditText myEditText =
(EditText)findViewById(R.id.myEditText); }
```

8. Still within onCreate, define an ArrayList of Strings to store each to-do list item. You can bind a ListView to an ArrayList using an ArrayAdapter, so create a new ArrayAdapter instance to bind the to-do item array to the ListView.

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

  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText =
  (EditText)findViewById(R.id.myEditText);

  // Create the array list of to do items
```

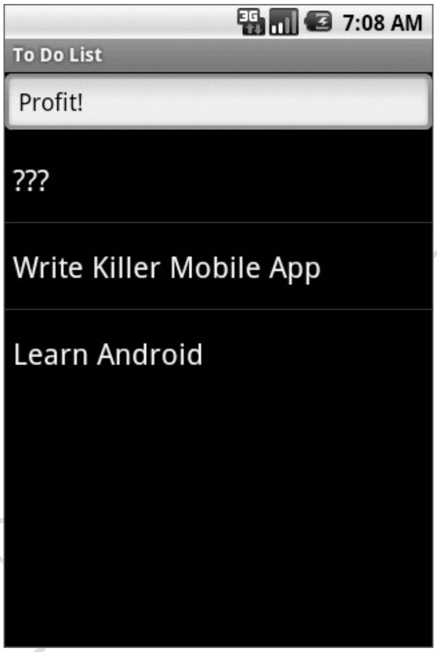
```
 final ArrayList<String> todoItems = new
ArrayList<String>();
 // Create the array adapter to bind the array to the listview
final ArrayAdapter<String> aa;
aa = new ArrayAdapter<String>(this,
android.R.layout.simple_list_item_1, todoItems);
// Bind the array adapter to the listview.
myListView.setAdapter(aa);
}
```

9. The final step to make this to-do list functional is to let users add new to-do items. Add an onKeyListener to the EditText that listens for a ‘‘D-pad center button’’ click before adding the contents of the EditText to the to-do list array and notifying the ArrayAdapter of the change. Then clear the EditText to prepare for another item.

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

  ListView myListView = (ListView)findViewById(R.id.myListView);
  final EditText myEditText =
  (EditText)findViewById(R.id.myEditText);

  final ArrayList<String> todoItems = new ArrayList<String>(); final
  ArrayAdapter<String> aa;
  aa = new ArrayAdapter<String>(this,
  android.R.layout.simple_list_item_1, todoItems);
  myListView.setAdapter(aa);

  myEditText.setOnKeyListener(new OnKeyListener() {
    public boolean onKey(View v, int keyCode, KeyEvent
    event) {
      if (event.getAction() == KeyEvent.ACTION_DOWN)
        if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER){
          todoItems.add(0,
          myEditText.getText().toString());
          aa.notifyDataSetChanged();
```

```
myEditText.setText("");
return true;
```

```
          } return false;
        }
      });
    }
```



10. Run or debug the application and you'll see a text entry box above a list, as shown in Figure 2-16.

11. You've now finished your first "real" Android application.

As it stands, this to-do list application isn't spectacularly useful. It doesn't save to-do list items between sessions, you can't edit or remove an item from the list, and typical tasklist items like due dates and task priority aren't recorded or displayed.

ANDROID DEVELOPMENT TOOLS

The Android SDK includes several tools and utilities to help you create, test, and debug your projects.

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

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

computer. Use the emulator to test and debug your Android applications.

- **Dalvik Debug Monitoring Service (DDMS)** Use the DDMS perspective to monitor and control the Dalvik virtual machines on which you're debugging your applications.
- **Android Asset Packaging Tool (AAPT)**Constructs the distributable Android package files(.apk).
- **Android Debug Bridge (ADB)** A client-server application that provides a link to a running emulator. It lets you copy files, install compiled application packages (.apk), and run shell commands.

The following additional tools are also available:

- **SQLite3** A database tool that you can use to access the SQLite database files created and used by Android.
- **Traceview**A graphical analysis tool for viewing the trace logs from your Android application.
- **MkSDCard** Creates an SD card disk image that can be used by the emulator to simulate an external storage card.
- **Dx** Converts Java .class bytecode into Android .dex bytecode.
- **activityCreator** A script that builds Ant build files that you can then use to compile your Android applications without the ADT plug-in.
- **layoutOpt** A tool that analyzes your layout resources and suggests improvements and optimizations.

The Android Virtual Device and SDK Manager

The Virtual Device and SDK Manager is a tool used to create and manage the virtual devices that will host instances of your emulator. You can use the same tool both to see which version of the SDK you have installed and to install new SDKs when they are released.

Android Virtual Devices

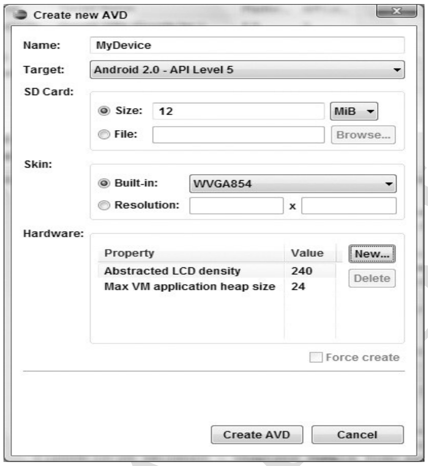
Android Virtual Devices are used to simulate the software builds and hardware specifications available on different devices. This lets you test your application on a variety of hardware platforms without needing to buy a variety of phones.

Each virtual device is configured with a name, a target build of Android (based on the SDK version it supports), an SD Card capacity, and screen resolution, as shown in the ‘‘Create new AVD’’ dialog in Figure 2-17.

Each virtual device also supports a number of specific hardware settings and restrictions that can be added in the form of NVPs in the hardware table. These additional settings include:

➢ Maximum virtual machine heap size

➢ Screen pixel density



Create new AVD

Name: MyDevice
Target: Android 2.0 - API Level 5
SD Card:
(●) Size: 12 MiB
( ) File: Browse...
Skin:
(●) Built-in: WVGA854
( ) Resolution: x
Hardware:

Property	Value
Abstracted LCD density	240
Max VM application heap size	24

FIGURE 2-17

➢ SD Card support

➢ The existence of DPad, touchscreen, keyboard, and trackball hardware

➢ Accelerometer and GPS support

➢ Available device memory

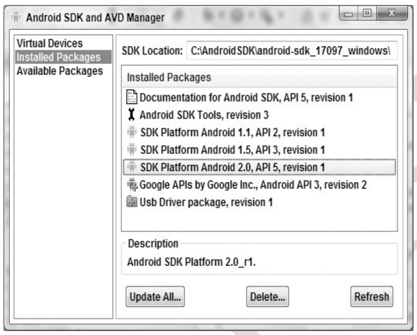
- ➢ Camera hardware (and resolution)
- ➢ Support for audio recording

Different hardware settings and screen resolutions will present alternative user-interface skins to represent the different hardware configurations. This simulates a variety of mobile device types. To complete the illusion, you can create a custom skin for each virtual device to make it look like the device it is emulating.

SDK Manager

Use the installed and available package tabs to manage your SDK installations.

Installed Packages, shown in Figure 2-18, displays the SDK platforms, documentation, and tools you have available to use in your development environment. When updating to a new version you can simply click the **Update All**... button to have the manager update your SDK installation with the latest version of each component.


Android SDK and AVD Manager

Virtual Devices
Installed Packages
Available Packages

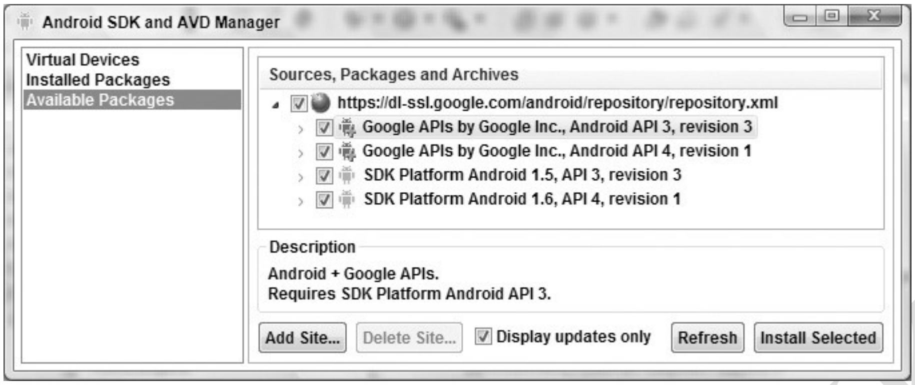
SDK Location: C:\AndroidSDK\android-sdk_17097_windows\

Installed Packages
- Documentation for Android SDK, API 5, revision 1
- Android SDK Tools, revision 3
- SDK Platform Android 1.1, API 2, revision 1
- SDK Platform Android 1.5, API 3, revision 1
- SDK Platform Android 2.0, API 5, revision 1
- Google APIs by Google Inc., Android API 3, revision 2
- Usb Driver package, revision 1

Description
Android SDK Platform 2.0_r1.

FIGURE 2-18

Alternatively, Available Packages checks the Android SDK repository for any source, packages, and archives available but not yet installed on your system. Use the checkboxes, as shown in Figure 2-19, to select additional SDK packages to install.



Android SDK and AVD Manager

Virtual Devices
Installed Packages
Available Packages

Sources, Packages and Archives

- [✓] https://dl-ssl.google.com/android/repository/repository.xml
  - [✓] Google APIs by Google Inc., Android API 3, revision 3
  - [✓] Google APIs by Google Inc., Android API 4, revision 1
  - [✓] SDK Platform Android 1.5, API 3, revision 3
  - [✓] SDK Platform Android 1.6, API 4, revision 1

Description
Android + Google APIs.
Requires SDK Platform Android API 3.

FIGURE 2-19

The Android Emulator

The emulator is the perfect tool for testing and debugging your applications.

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

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

The ADT plug-in integrates the emulator into Eclipse so that it's launched automatically within the selected AVD when you run or debug your projects. If you aren't using the plug-in or want to use the emulator outside of Eclipse, you can telnet into the emulator and control it from its console.

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

Dalvik Debug Monitor Service (DDMS)

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

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

The Android Debug Bridge (ADB)

The *Android debug bridge (ADB)* is a client-service application that lets you connect with an Android Emulator or device. It's made up of three components: a daemon running on the emulator, a service that runs on your development hardware, and client applications (like the DDMS) that communicate with the daemon through the service.

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

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

WHAT MAKES AN ANDROID APPLICATION?

Android applications consist of loosely coupled components, bound by an application manifest that describes each component and how they all interact, as well as the application metadata including its hardware and platform requirements.

The following six components provide the building blocks for your applications:

- **Activities** Your application's presentation layer. Every screen in your application will be an extension of the Activity class. Activities use Views to form graphical user interfaces that display information and respond to user actions. In terms of desktop development, an Activity is equivalent to a Form.
- **Services** The invisible workers of your application. Service components run in the background, updating your data sources and visible Activities and triggering Notifications. They're used to perform regular processing that needs to continue even when your application's Activities aren't active or visible.
- **Content Providers** Shareable data stores. Content Providers are used to manage and share application databases. They're the preferred means of sharing data across application boundaries. This means that you can configure your own Content Providers to permit access from other applications and use Content Providers exposed by others to access their stored data. Android devices include several native Content Providers that expose useful databases like the media store and contact details.
- **Intents** An inter-application message-passing framework. Using Intents, you can broadcast messages system-wide or to a target Activity or Service, stating your intention to have an action performed. The system will then determine the target(s) that will perform any actions as appropriate.
- **Broadcast Receivers** Intent broadcast consumers. If you create and register a Broadcast Receiver, your application can listen for broadcast Intents that match specific filter criteria. Broadcast

Receivers will automatically start your application to respond to an incoming Intent, making them perfect for creating event-driven applications.

- **Widgets** Visual application components that can be added to the home screen. A special variation of a Broadcast Receiver, widgets let you create dynamic, interactive application components for users to embed on their home screens.
- **Notifications** A user notification framework. Notifications let you signal users without stealing focus or interrupting their current Activities. They're the preferred technique for getting a user's attention from within a Service or Broadcast Receiver. For example, when a device receives a text message or an incoming call, it alerts you by flashing lights, making sounds, displaying icons, or showing messages.

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

INTRODUCING THE APPLICATION MANIFEST

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

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

The manifest also offers attributes to specify application metadata (like its icon or theme), and additional top-level nodes can be used for security settings, unit tests, and defining hardware and platform support requirements, as described below.

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

Use the versionCode attribute to define the current application version as an integer. This value is used internally to compare application versions. Use the versionName attribute to specify a public version number that is displayed to users.

A typical manifest node is shown in the following XML snippet:

```
<manifest
          xmlns:android=http://schemas.android.com/apk/res/androi
          d
package="com.my_domain.my_app"
android:versionCode="1"        android:versionName="0.9
          Beta">
          [ ... manifest nodes ... ]
</manifest>
```

The <manifest> tag includes nodes that define the application components, security settings, test classes, and requirements that make up your application. The following list gives a summary of the available <manifest> node tags, and an XML snippet demonstrating how each one is used:

- uses-sdk This node lets you define a minimum, maximum, and target SDK version that must be available on a device in order for your application to function properly. Using a combination of minSDKVersion,maxSDKVersion,andtargetSDKVersion attributes you can restrict which devices your application can run on, based on the SDK version supported by the installed platform.

  The minimum SDK version specifies the lowest version of the SDK that includes the APIs you have used in your application. If you fail to specify a minimum version one will be assumed and your application will crash if it attempts to access APIs that aren't available on the host device.

  The maximum SDK version lets you define an upper limit you are willing to support. Your application will not be visible on the Market for devices running a higher platform release. It's good practice *not* to set the maximum SDK value unless you know your application will definitely not work on newer platform releases.

The target SDK version attribute lets you specify the platform against which you did your development and testing. Setting a target SDK version tells the system that there is no need to apply any forward- or backward- compatibility changes to support that particular version.

```
<uses-sdk android:minSdkVersion="4"
android:targetSdkVersion="5"> </uses-sdk>
```

- uses-configuration Use uses-configuration nodes to specify each combination of input mechanisms supported by your application. You can specify any combination of input devices that include:
  - reqFiveWayNav Specify true for this attribute if you require an input device capable of navigating up, down, left, and right and of clicking the current selection. This includes both trackballs and D-pads.
  - reqHardKeyboard If your application requires a hardware keyboard specify true.
  - reqKeyboardType Lets you specify the keyboard type as one of nokeys, qwerty, twelvekey, or undefined.
  - reqNavigation Specify the attribute value as one of nonav, dpad, trackball, wheel, or undefined as a required navigation device.
  - reqTouchScreen Select one of notouch, stylus, finger, or undefined to specify the required touchscreen input.

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

```
<uses-configuration android:reqTouchScreen=["finger"]
                    android:reqNavigation=["trackball"]
                    android:reqHardKeyboard=["true"]
                    android:reqKeyboardType=["qwerty"/>
```

```
<uses-configuration android:reqTouchScreen=["finger"]
                    android:reqNavigation=["trackball"]
                    android:reqHardKeyboard=["true"]
                    android:reqKeyboardType=["twelvekey"]/>
```

- uses-feature One of the advantages of Android is the wide variety of hardware platforms it runs on. Use multiple uses-feature nodes to specify each of the hardware features your application requires. This will prevent your application from being installed on a device that does not include a required hardware feature. You can require support for any hardware that is optional on a compatible device. Currently optional hardware features include:
  - android.hardware.camera For applications that require camera hardware.
  - android.hardware.camera.autofocus If you require an autofocus camera.

  As the variety of platforms on which Android is available increases, so too will the optional hardware.

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

```
<uses-feature android:glEsVersion=" 0x00010001"
              android:name="android.hardware.camera" />
```

- supports-screens After the initial round of HVGA hardware, 2009 saw the introduction of WVGA and QVGA screens to the Android device menagerie.

  Exact dimensions will vary depending on hardware, but in general the supported screen sizes match resolutions as follows:
  - smallScreens Screens with a resolution smaller than traditional HVGA — typically QVGA screens.
  - normalScreens Used to specify typical mobile phone screens of at least HVGA, including WVGA and WQVGA.

- ➢ largeScreens Screens larger than normal. In this instance a large screen is considered to be significantly larger than a mobile phone display.
- ➢ anyDensity Set to true if your application can be scaled to accommodate any screen resolution.

As of SDK 1.6 (API level 4), the default value for each attribute is true. Use this node to specify screen sizes you do not support.

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

➢ application

A manifest can contain only one application node. It uses attributes to specify the metadata for your application (including its title, icon, and theme). During development you should include a debuggable attribute set to true to enable debugging — though you may wish to disable this on your release builds.

The <application> node also acts as a container that includes the Activity, Service, Content Provider, and Broadcast Receiver tags used to specify the application components. You can also define your own implementation of the Application class.

```
<application android:icon="@drawable/icon"
        android:theme="@style/my_theme"
        android:name="MyApplication"
        android:debuggable="true">
        [ ... application nodes ... ]
</application>
```

- ➢ activity An <activity> tag is required for every Activity displayed by your application. Using the android:name attribute to specify the Activity class name.

Each Activity node supports <intent-filter> child tags that specify which Intents launch the Activity.

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

➢ service As with the activity tag, create a new service tag for each Service class used in your application. Service tags also support <intent-filter> child tags to allow late runtime binding.

```
<service android:enabled="true"

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

➢ provider Provider tags specify each of your application's Content Providers.

```
<provider

android:permission="com.paad.MY_PERMISSION"
android:name=".MyContentProvider"

android:enabled="true"
android:authorities="com.paad.myapp.MyContentProvider">
</provider>
```

➢ receiver By adding a receiver tag, you can register a Broadcast Receiver without having to launch your application first. Broadcast Receivers are like global event listeners that, once registered, will execute whenever a

matching Intent is broadcast by the system or an application. By registering a Broadcast Receiver in the manifest you can make this process entirely autonomous. If a matching Intent is broadcast, your application will be started automatically and the registered Broadcast Receiver will be run.

```
<receiver android:enabled="true"
          android:label="MyIntentReceiver"
          android:name=".MyIntentReceiver">
</receiver>
```

- uses-permission As part of the security model, uses-permission tags declare the permissions you've determined your application needs to operate properly. The permissions you include will be presented to the user before installation commences. Permissions are required for many of the native Android services, particularly those with a cost or security implication (such as dialing, receiving SMS, or using the location-based services).

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

- permission Third-party applications can also specify permissions before providing access to shared application components. Before you can restrict access to an application component, you need to define a permission in the manifest. Use the permission tag to create a permission definition.

  Application components can then require permissions by adding the android:permission attribute. Other applications will then need to include a uses-permission tag in their manifests to use these protected components.

  Within the permission tag, you can specify the level of access the permission will permit
  (normal, dangerous, signature, signatureOrSystem), a label, and an external resource containing the description that explains the risks of granting the specified permission.

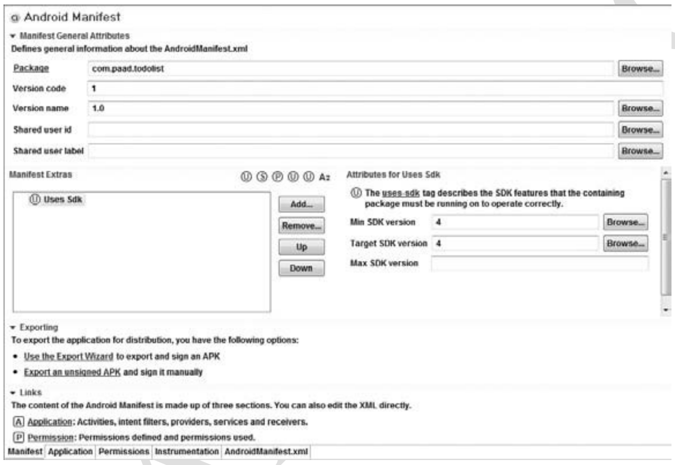
```
<permission
android:name="com.paad.DETONATE_DEVICE"
android:protectionLevel="dangerous" android:label="Self
Destruct"
```

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

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

```
<instrumentation android:label="My Test"
android:name=".MyTestClass"
android:targetPackage="com.paad.aPackage">
</instrumentation>
```

The ADT New Project Wizard automatically creates a new manifest file when it creates a new project.


Android Manifest

▾ Manifest General Attributes

Defines general information about the AndroidManifest.xml

Package	com.paad.todolist	Browse...
Version code	1	
Version name	1.0	Browse...
Shared user id		Browse...
Shared user label		Browse...

Manifest Extras

Uses Sdk

Attributes for Uses Sdk

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

Min SDK version	4	Browse...
Target SDK version	4	Browse...
Max SDK version		

▾ Exporting

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

▾ Links

The content of the Android Manifest is made up of three sections. You can also edit the XML directly.

A Application: Activities, intent filters, providers, services and receivers.

P Permission: Permissions defined and permissions used.

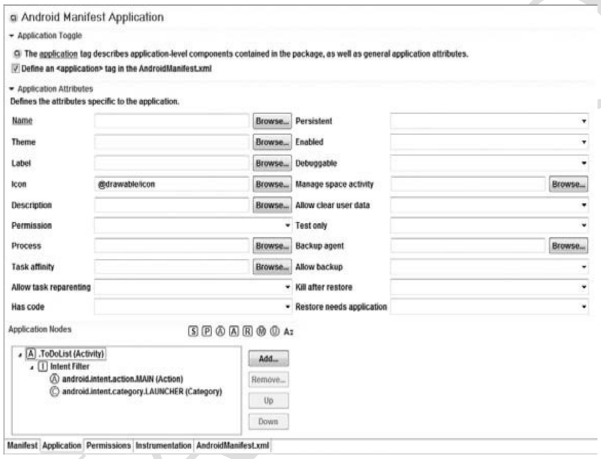
FIGURE 3-1

THE ANDROID APPLICATION LIFE CYCLE

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

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

Android aggressively manages its resources, doing whatever it takes to ensure that the device remains responsive. This means that processes (and their hosted applications) will be killed, without warning in some cases, to free resources for higher-priority applications — generally those interacting directly with the user at the time



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		Persistent	
Theme		Enabled	
Label		Debuggable	
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	

```
.ToDoList (Activity)
  Intent Filter
    android.intent.action.MAIN (Action)
    android.intent.category.LAUNCHER (Category)
```

FIGURE 3-2

UNDERSTANDING APPLICATION PRIORITY AND PROCESS STATES

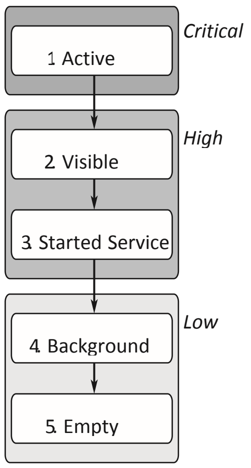
The order in which processes are killed to reclaim resources is determined by the priority of the hosted applications. An application's priority is equal to its highest-priority component.

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

second application, the secondary application will have at least as high a priority as the application it supports.

It's important to structure your application correctly to ensure that its priority is appropriate for the work it's doing. If you don't, your application could be killed while it's in the middle of something important.

The following list details each of the application states shown in Figure 3-3, explaining how the state is determined by the application components comprising it:



➢ **Active processes** Active (foreground) processes have application components interacting with the user. These are the processes Android is trying to keep responsive by reclaiming resources. There are generally very few of these processes, and they will be killed only as a last resort.

Active processes

➢ Activities in an ''active'' state; that is, those in the foreground responding to user events.

➢ Broadcast Receivers executing onReceive event handlers.

➢ Services executing onStart, onCreate, or onDestroy event handlers.

➢ Running Services that have been flagged to run in the foreground.

➢ **Visible processes** Visible but inactive processes are those hosting ‘‘visible’’ Activities. As the name suggests, visible Activities are visible, but they aren’t in the foreground or responding to user events. This happens when an Activity is only partially obscured (by a non-full-screen or transparent Activity). There are generally very few visible processes, and they’ll be killed only under extreme circumstances to allow active processes to continue.

➢ **Started Service processes** Processes hosting Services that have been started. Services support ongoing processing that should continue without a visible interface. Because background Services don’t interact directly with the user, they receive a slightly lower priority than visible Activities. They are still considered foreground

processes and won’t be killed unless resources are needed for active or visible processes.

➢ **Background processes** Processes hosting Activities that aren’t visible and that don’t have any running Services. There will generally be a large number of background processes that Android will kill using a last-seen-first-killed pattern in order to obtain resources for foreground processes.

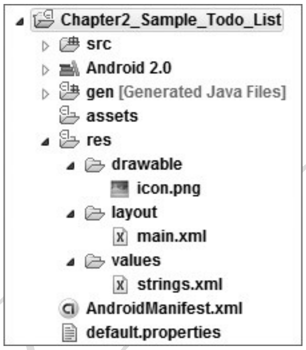
➢ **Empty processes** To improve overall system performance, Android will often retain an application in memory after it has reached the end of its lifetime. Android maintains this cache to improve the start-up time of applications when they’re relaunched. These processes are routinely killed as required.

Creating Resources

Application resources are stored under the res/ folder of your project hierarchy. In this folder each of the available resource types are stored in a subfolder containing those resources.

If you start a project using the ADT wizard, it will create a res folder that contains subfolders for the values, drawable-ldpi,drawable-mdpi, drawable-hdpi, and layout resources that contain the default layout, application icon, and string resource definitions respectively, as shown in Figure 3-4.

Note that three Drawable resource folders are created with three different icons, one each for low, medium, and high DPI displays.



```
Chapter2_Sample_Todo_List
  src
  Android 2.0
  gen [Generated Java Files]
  assets
  res
    drawable
      icon.png
    layout
      main.xml
    values
      strings.xml
  AndroidManifest.xml
  default.properties
```

FIGURE 3-4

Nine primary resource types have different folders: simple values, Drawables, layouts, animations, styles, menus, searchables, XML, and raw resources. When your application is built, these resources will be compiled as efficiently as possible and included in your application
package.

This process also generates an R class file that contains references to each of the resources you include in your

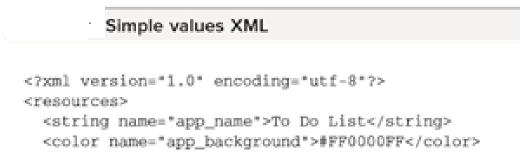
project.

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

Creating Simple Values

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

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



Simple values XML

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">To Do List</string>
  <color name="app_background">#FF0000FF</color>
```

```
    <dimen name="default_border">5px</dimen>
    <array name="string_array">
      <item>Item 1</item>
      <item>Item 2</item>
      <item>Item 3</item>
    </array>
    <array name="integer_array">
      <item>3</item>
      <item>2</item>
      <item>1</item>
    </array>
  </resources>
```

This example includes all the simple value types. By convention, resources are separated into different files for each type; for example, res/values/strings.xml would contain only string resources.

The following sections detail the options for defining simple resources.

Strings

Externalizing your strings helps maintain consistency within your application and makes it much easier to create localized versions.

String resources are specified with the <string> tag, as shown in the following XML snippet.

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

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

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

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

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

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

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

Colors

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

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

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

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

Dimensions

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

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

- ➢ px (screen pixels)
- ➢ in (physical inches)
- ➢ pt (physical points)
- ➢ mm (physical millimeters)
- ➢ dp (density-independent pixels relative to a 160-dpi screen)
- ➢ sp (scale-independent pixels)

These alternatives let you define a dimension not only in absolute terms, but also using relative scales that account for different screen resolutions and densities to simplify scaling on different hardware.

The following XML snippet shows how to specify dimension values for a large font size and a standard border:

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

Styles and Themes

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

You can easily change the appearance of your application by simply specifying a different style as the theme in your project manifest.

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

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="StyleName">
      <item name="attributeName">value</item>
  </style>
</resources>
```

Styles support inheritance using the parent attribute on the <style> tag, making it easy to create simple variations.

The following example shows two styles that can also be used as a theme: a base style that sets several text properties and a second style that modifies the first to specify a smaller font.

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

  <style name="SmallText" parent="BaseText">
    <item name="android:textSize">8sp</item>
  </style>
</resources>
```

Drawables

Drawable resources include bitmaps and NinePatch (stretchable PNG) images. They also include complex composite Drawables, such as LevelListDrawables and StateListDrawables that can be defined in XML.

All Drawables are stored as individual files in the res/drawable folder. The resource identifier for a Drawable resource is the lowercase file name without an extension.

Layouts

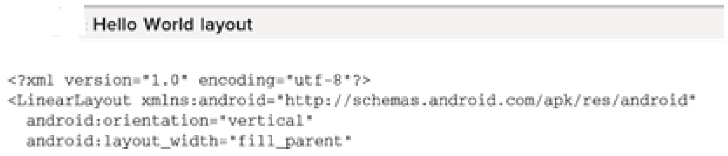
Layout resources let you decouple your presentation layer by designing user interface layouts in XML rather than constructing them in code.

The most common use of a layout is for defining the user interface for an Activity. Once defined in
XML, the layout is ‘‘inflated’’ within an Activity using setContentView, usually within the onCreate method. You can also reference layouts from within other layout resources, such as layouts for each row in a List View.

Using layouts to create your screens is best-practice UI design in Android. The decoupling of the layout from the code lets you create optimized layouts for different hardware configurations, such as varying screen sizes, orientation, or the presence of keyboards and touchscreens.

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

layout container for a Text View that displays the ‘‘Hello World’’ greeting.



Hello World layout

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

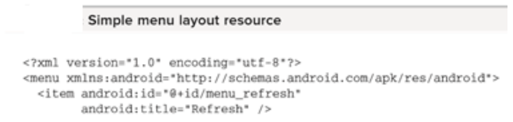
```
android:layout_height="fill_parent">
        <TextView
          android:layout_width="fill_parent"
          android:layout_height="wrap_conten
          t" android:text="Hello World!"/>


     </LinearLayout>
```

Menus

Create menu resources to further decouple your presentation layer by designing your menu layouts in XML rather than constructing them in code.

Menu resources can be used to define both Activity and context menus within your applications, and provide the same options you would have when constructing your menus in code. Once defined in XML, a menu is ‘‘inflated’’ within your application via the inflate method of the MenuInflator Service, usually within the onCreateOptionsMenu method.

Each menu definition is stored in a separate file, each containing a single menu, in the res/menu folder. The file name then becomes the resource identifier. Using XML to define your menus is best-practice design in Android.



Simple menu layout resource

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

```
   <item
 android:id="@+id/menu_settings"
 android:title="Settings" /> </menu>
```

Using Resources

As well as the resources you create, Android supplies several system resources that you can use in your applications. The resources can be used directly from your application code and can also be referenced from within other resources (e.g., a dimension resource might be referenced in a layout definition).

Android will automatically select the most appropriate value for a given resource identifier based on the current hardware and device settings.

Using Resources in Code

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

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

The value of these variables is a reference to the corresponding resource's location in the resource table, *not* an instance of the resource itself.

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

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

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

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

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

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

```
Resources myResources = getResources();

CharSequence styledText =

myResources.getText(R.string.stop_message);
```

```
Drawable icon =

myResources.getDrawable(R.drawable.app_icon); int

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

float borderWidth =

myResources.getDimension(R.dimen.standard_border);

Animation tranOut;

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

String[] stringArray; stringArray =

myResources.getStringArray(R.array.string_array);

 int[] intArray =

myResources.getIntArray(R.array.integer_array);
```

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

```
AnimationDrawable rocket;
rocket =
(AnimationDrawable)myResources.getDrawable(R.drawable.frame_by
_frame);
```

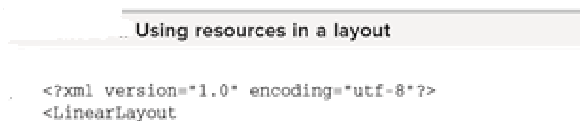
Referencing Resources within Resources

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

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

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

layout that uses color, dimension, and string resources.



.. Using resources in a layout

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

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:padding="@dimen/standard_border">
    <EditText
    android:id="@+id/myEditText"
    android:layout_width="fill_parent"
    android:layout_height="wrap_conte
    nt"
    android:text="@string/stop_mes
    sage"
    android:textColor="@color/opaq
    ue_blue" />
  </LinearLayout>
```

Using System Resources

The native Android applications externalize many of their resources, providing you with various strings, images, animations, styles, and layouts to use in your applications.

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

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

To access system resources in XML specify Android as the package name, as shown in this XML snippet.

```
<EditText android:id="@+id/myEditText"
  android:layout_width="fill_parent"
  android:layout_height="wrap_content
  "
  android:text="@android:string/httpEr
  rorBadUrl"
  android:textColor="@android:color/d
  arker_gray"
/>
```

Referring to Styles in the Current Theme

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

The following example shows a snippet of the preceding code but uses the current theme's text color rather than an external resource.

```
<EditText
android:id="@+id/myEditText"
android:layout_width="fill_parent"
android:layout_height="wrap_conte
nt"
android:text="@string/stop_messa
ge"
android:textColor="?android:tex
tColor" />
```

This technique lets you create styles that will change if the current theme changes, without your having to modify each individual style resource.

INTRODUCING THE ANDROID APPLICATION CLASS

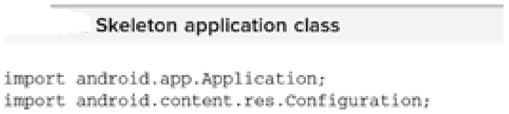
Extending the Application class with your own implementation enables you to do three things:

1. Maintain application state
2. Transfer objects between application components
3. Manage and maintain resources used by several application components

When your Application implementation is registered in the manifest, it will be instantiated when your application process is created. As a result your Application implementation is by nature a singleton and should be implemented as such to provide access to its methods and member variables.

Extending and Using the Application Class

the skeleton code for extending the Application class and implementing it as a singleton.


Skeleton application class

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

```
public class MyApplication extends Application {

  private static MyApplication singleton;

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


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

Once created, you must register your new Application class in the manifest’s <application> node, as shown in the following snippet:

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

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

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

This is a particularly effective technique for transferring objects between your loosely coupled application components, or for maintaining application state or shared resources.

Overriding the Application Life Cycle Events

The Application class also provides event handlers for application creation and termination, low available memory, and configuration changes.

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

- onCreate Called when the application is created. Override this method to initialize your application singleton and create and initialize any application state variables or shared resources.
- onTerminate Can be called when the application object is terminated. Note that there is no guarantee of this method handler's being called. If the application is terminated by the kernel in order to free resources for other applications, the process will be terminated without warning and without a call to the application object's onTerminate handler.
- onLowMemory Provides an opportunity for well-behaved applications to free additional memory when the system is running low on resources. This will generally only be called when background processes have already been terminated and the current foreground applications are still low on memory. Override this handler to clear caches or release unnecessary resources.
- onConfigurationChanged Unlike with Activities, your application object is not killed and restarted for configuration changes. Override this handler if it is necessary to handle configuration changes at an application level.

A CLOSER LOOK AT ANDROID ACTIVITIES

To create user interface screens you extend the Activity class, using Views to provide the UI and allow user interaction.

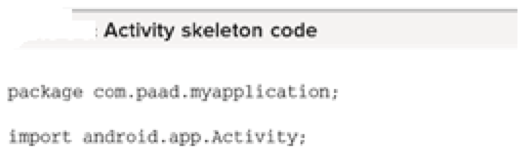
Each Activity represents a screen (similar to a Form) that an application can present to its users. The more complicated your application, the more screens you are likely to need.

Create a new Activity for every screen you want to display. Typically this includes at least a primary interface screen that handles the main UI functionality of your application. This primary interface is often supported by secondary Activities for entering information, providing different perspectives on your data, and supporting additional functionality. To move between screens start a new Activity (or return from one).

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

Creating an Activity

Extend Activity to create a new Activity class. Within this new class you must define the user interface and implement your functionality. The basic skeleton code for a new Activity is shown



Activity skeleton code

```
package com.paad.myapplication;

import android.app.Activity;
```

```
import android.os.Bundle; public class

MyActivity extends Activity {

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

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

Views are the user interface controls that display data and provide user interaction. Android provides several layout classes, called *View Groups*, that can contain multiple Views to help you design your user interfaces.

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

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

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

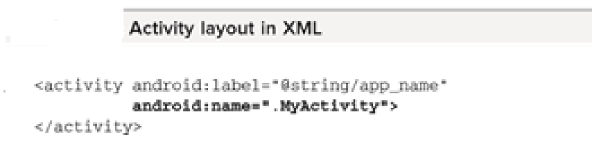
Usually you'll want to use a more complex UI design. You can create a layout in code using layout View Groups, or you can use the standard Android convention of passing a resource ID for a layout defined in an external resource, as shown in the following snippet:

```
@Override
```

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

In order to use an Activity in your application you need to register it in the manifest. Add new
<activity> tags within the <application> node of the manifest; the <activity> tag includes attributes for metadata such as the label, icon, required permissions, and themes used by the Activity. An Activity without a corresponding <activity> tag can't be displayed.

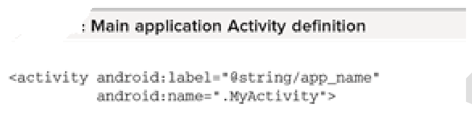
The XML shows how to add a node for the MyActivity class .



Activity layout in XML

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

Within the <activity> tag you can add <intent-filter> nodes that specify the Intents your Activity will listen for and react to.



: Main application Activity definition

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

```
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category
  android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

The Activity Life Cycle

Activity Stacks

The state of each Activity is determined by its position on the Activity stack, a last-in–first-out collection of all the currently running Activities. When a new Activity starts, the current foreground screen is moved to the top of the stack. If the user navigates back using the Back button, or the foreground Activity is closed, the next Activity on the stack moves up and becomes active. This process is illustrated in Figure 3-6.

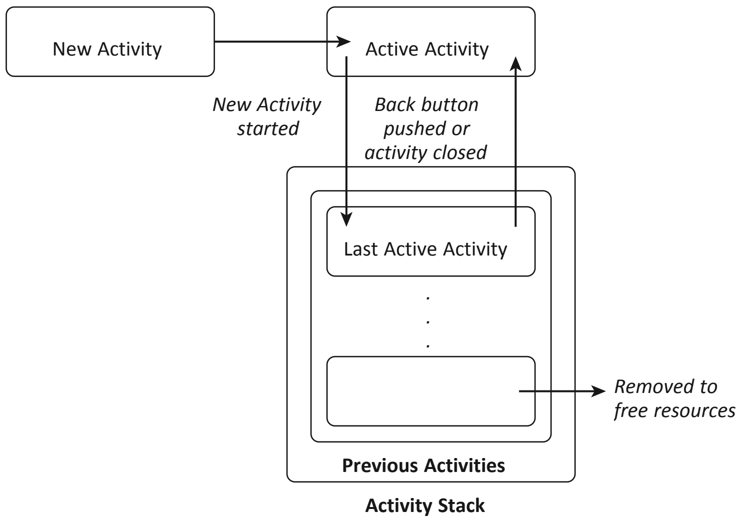


FIGURE 3-6

An application's priority is influenced by its highest priority Activity. When the Android memory manager is deciding which application to terminate to free resources, it uses this stack to determine the priority of applications based on their Activities.

Activity States

As Activities are created and destroyed they move in and out of the stack shown in Figure 3-6. As they do so, they transition through four possible states:

- **Active** When an Activity is at the top of the stack it is the visible, focused, foreground Activity that is receiving user input. Android will attempt to keep it alive at all costs,

  killing Activities further down the stack as needed, to ensure that it has the resources it needs. When another Activity becomes active, this one will be paused.

- **Paused** In some cases your Activity will be visible but will not have focus; at this point it's paused. This state is reached if a transparent or non-full-screen Activity is active in front of it. When paused, an Activity is treated as if it were active; however, it doesn't receive user input events. In extreme cases Android will kill a paused Activity to recover resources for the active Activity. When an Activity becomes totally obscured, it is stopped.

- **Stopped** When an Activity isn't visible, it ''stops.'' The Activity will remain in memory, retaining all state information; however, it is now a candidate for termination when the system requires memory elsewhere. When an Activity is stopped it's important to save data and the current UI state. Once an Activity has exited or closed, it becomes inactive.

- **Inactive** After an Activity has been killed, and before it's been launched, it's inactive. Inactive Activities have been removed from the Activity stack and need to be restarted before they can be displayed and used.

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

Monitoring State Changes

To ensure that Activities can react to state changes, Android provides a series of event handlers that are fired when an Activity transitions through its full, visible, and active lifetimes. Figure 3-7 summarizes these lifetimes in terms of the Activity states described in the previous section.

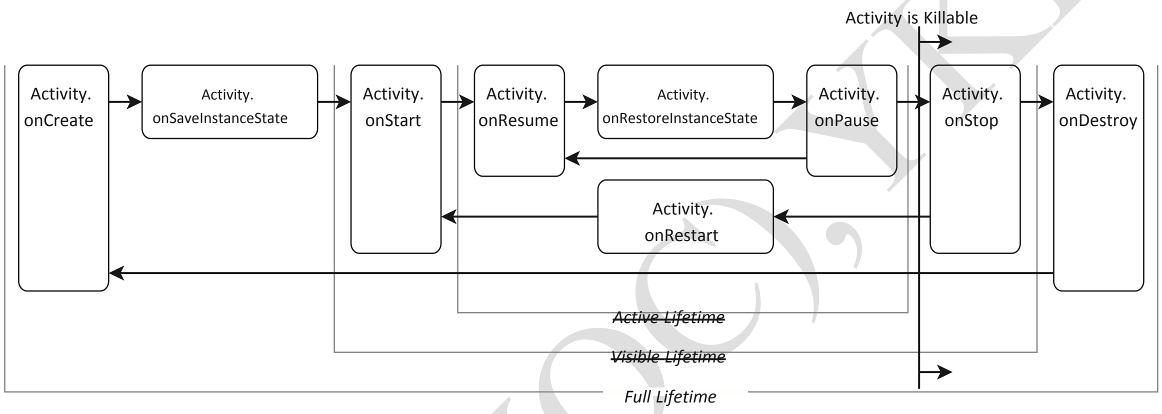
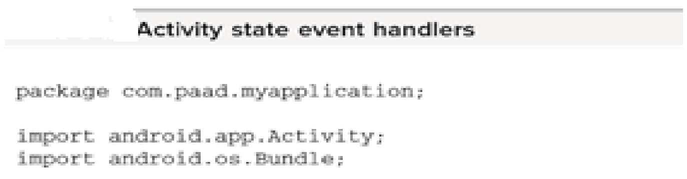


FIGURE 3-7

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


Activity state event handlers

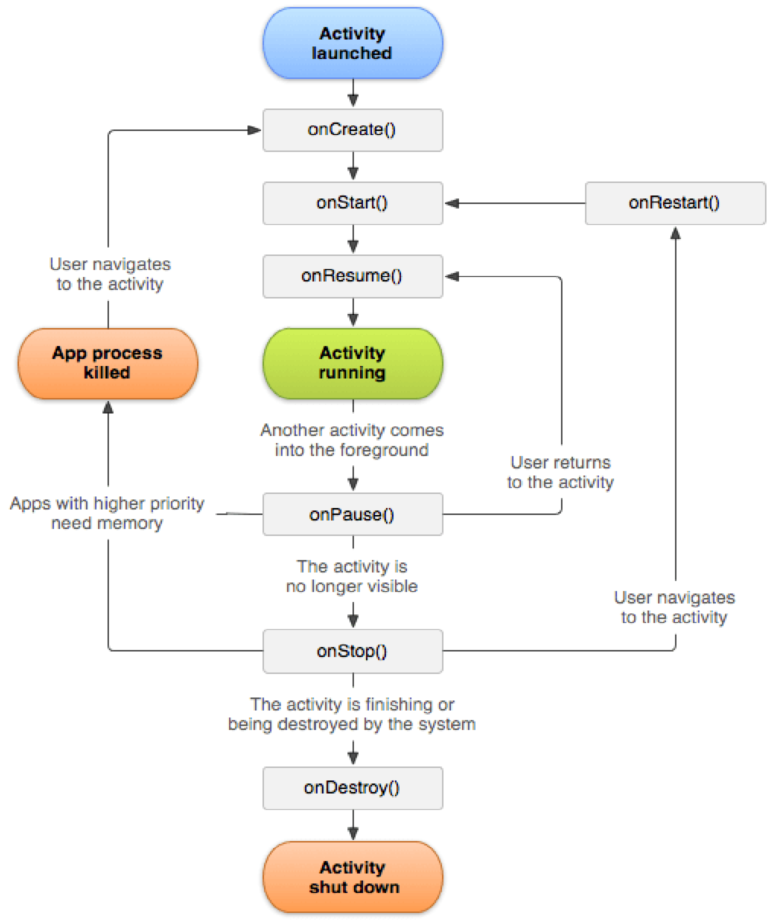
```
package com.paad.myapplication;

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

```
public class MyActivity extends Activity {
// Called at the start of the full lifetime.
@Override public void onCreate(Bundle
savedInstanceState) {
super.onCreate(savedInstanceState); // Initialize activity.
}

// Called after onCreate has finished, use to restore UI state
@Override
```

```
public void onRestoreInstanceState(Bundle savedInstanceState) {
super.onRestoreInstanceState(savedInstanceState); // Restore UI
state from the savedInstanceState. // //This bundle has also been
passed to onCreate.
}

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

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

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

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

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

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

          // Called at the end of the visible lifetime.
```

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

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

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


Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Understanding Activity Lifetimes

Within an Activity's full lifetime, between creation and destruction, it will go through one or more iterations of the active and visible lifetimes. Each transition will trigger the method handlers described previously.

The Full Lifetime

The full lifetime of your Activity occurs between the first call to onCreate and the final call to onDestroy. It's possible, in some cases, for an Activity's process to be terminated *without* the onDestroy method being called.

Use the onCreate method to initialize your Activity: inflate the user interface, allocate references to class variables, bind data to controls, and create Services and threads. The onCreate method is passed a Bundle object containing the UI state saved in the last call to onSaveInstanceState. You should use this Bundle to restore the user interface to its previous state, either within the onCreate method or by overriding onRestoreInstanceState.

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

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

The Visible Lifetime

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

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

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

The onStart/onStop methods are also used to register and unregister Broadcast Receivers that are being used exclusively to update the user interface.

The Active Lifetime

The active lifetime starts with a call to onResume and ends with a corresponding call to onPause.

An active Activity is in the foreground and is receiving user input events. Your Activity is likely to go through several active lifetimes before it's destroyed, as the active lifetime will end when a new Activity is displayed, the device goes to sleep, or the Activity loses focus. Try to keep code in the onPause and onResume methods relatively fast and lightweight to ensure that your application remains responsive when moving in and out of the foreground.

Immediately before onPause, a call is made to onSaveInstanceState. This method provides an opportunity to save the Activity's UI state in a Bundle that will be passed to the onCreate and onRestoreInstanceState methods. Use onSaveInstanceState to save the UI state (such as checkbox states, user focus, and entered but uncommitted user input) to ensure that the Activity can present the same UI when it next becomes active. You can safely assume that

during the active lifetime onSaveInstanceState and onPause will be called before the process is terminated.

Most Activity implementations will override at least the onPause method to commit unsaved changes, as it marks the point beyond which an Activity may be killed without warning. Depending on your application architecture you may also choose to suspend threads, processes, or Broadcast Receivers while your Activity is not in the foreground.

The onResume method can be very lightweight. You will not need to reload the UI state here as this is handled by the onCreate and onRestoreInstanceState methods when required. Use onResume to reregister any Broadcast Receivers or other processes you may have suspended in onPause.

Android Activity Classes

The Android SDK includes a selection of Activity subclasses that wrap up the use of common user interface widgets. Some of the more useful ones are listed here:

- **MapActivity** Encapsulates the resource handling required to support a MapView widget within an Activity.

- **ListActivity** Wrapper class for Activities that feature a ListView bound to a data source as the primary UI metaphor, and exposing event handlers for list item selection.
- **ExpandableListActivity**Similar to the List Activity but supporting an ExpandableListView
- **TabActivity** Enables you to embed multiple Activities or Views within a single screen using a tab widget to switch among them.

Method	Description
onCreate	called when activity is first created.
onStart	called when activity is becoming visible to the user.
onResume	called when activity will start interacting with the user.
onPause	called when activity is not visible to the user.
onStop	called when activity is no longer visible to the user.
onRestart	called after your activity is stopped, prior to start.
onDestroy	called before the activity is destroyed.