
Appendix C

Getting Started with Android

Learn Java for Android Development teaches Java as a preparatory step for getting into Android app development. Because many readers have asked for a primer on developing Android apps, I created this appendix to satisfy that desire in part. Realistically, I cannot offer a comprehensive treatment of this topic in an appendix. Instead, I present only a few of the basics.

Appendix C is divided into three main sections. In the first section I discuss creating an Android app development environment in terms of SDK, Android 4.4.2 platform, and Eclipse configuration. In the second section I present app architecture basics: components; views, view groups, and event listeners; resources; manifest; and app package. In the third section I develop some Android apps.

Creating an Android App Development Environment

Before you can develop Android apps, you need to create a development environment. In this section I will show you how to install the Android SDK, install the Android 4.4.2 platform, create an Android virtual device, start and explore this device, and configure Eclipse (introduced in Chapter 1) for Android app development.

Installing the Android SDK

Google supplies the Android SDK for developing Android apps. This SDK is available for the following operating systems:

- Windows XP (32-bit), Windows Vista (32-bit/64-bit), and Windows 7 (32-bit/64-bit)
- Mac OS X 10.5.8 or later (x86 only)
- Linux (tested on Ubuntu Linux, Lucid Lynx)

There are certain requirements for the Linux operating system:

- GNU C Library (glibc) 2.7 or later is required.
- On Ubuntu Linux, version 8.04 or later is required.
- 64-bit distributions must be capable of running 32-bit applications.

If you have one of these operating systems and meet the Linux requirements (assuming that Linux is your operating system), you'll be able to install the Android SDK.

■ **Note** You should ensure that you have JDK 6 or later installed. By itself, the JRE isn't sufficient. Neither is JDK 1.4 or Gnu Compiler for Java, which may be installed on some Linux distributions. Also, you will need to install the Apache Ant (<http://ant.apache.org/>) build system.

However, you must first download the SDK. Accomplish this task by pointing your browser to Google's "Get the Android SDK" page (<http://developer.android.com/sdk/index.html>) and follow these instructions:

1. Expand the "DOWNLOAD FOR OTHER PLATFORMS" section near the bottom of the page.
2. Choose one of the packages in the "SDK Tools Only" section. For Windows, you have the option of downloading either a ZIP archive or an EXE installer.

For my 64-bit Windows 7 platform, I chose to download the EXE installer. The latest installer file at time of writing was `installer_r22.3-windows.exe`. By the time you read this appendix, the installer will probably exist at a higher version number.

Clicking on a package link takes you to a license agreement page. Agree to the license's terms and conditions by clicking the check box below the license. Doing so enables the download button located below the check box. Click this button and follow browser instructions to download the file.

After downloading the approximately 85-megabyte installer file, complete the following instructions to install the SDK:

1. Scan the installer file for viruses. (This shouldn't be needed but you never can tell.)
2. Double-click the executable to start the setup wizard.

3. Follow wizard instructions. For the destination folder, I overrode the default setting with `C:\android`, for convenient access. This directory is the *home directory*.

The setup wizard's last task gives you the opportunity to run the SDK Manager tool on exit. I chose to not run this tool because I wanted to explore the installation first.

4. Add the `tools` directory (discussed shortly) to your `PATH` environment variable so that you can access the SDK's command-line tools from anywhere in your filesystem. (I'll demonstrate command-line usage later in this appendix.)

An examination of the home directory reveals several directories and files. For example, I observed the following directories and files on my Windows platform:

- `add-ons`: This initially empty directory stores *add-ons*, which are additional SDKs from Google and other vendors. For example, the Google APIs add-on is stored here.
- `platforms`: This initially empty directory stores *Android platforms* (combinations of system images that support specific processor architectures [such as ARM EABI, Intel x86, or MIPS] and other files) in separate directories. For example, Android 4.4.2 would be stored in one directory and Android 2.3.4 would be stored in another directory.
- `tools`: This directory contains a set of platform-independent development tools such as the emulator. These tools may be updated at any time independently of Android platform releases.
- `AVD Manager.exe`: This GUI-based executable file manages Android virtual devices. I'll have more to say about this topic later in this appendix.
- `SDK Manager.exe`: This GUI-based executable file manages the SDK in terms of the packages that you can download and install. It also lets you launch AVD Manager from a menu.
- `SDK Readme.txt`: This text file welcomes you to the SDK and provides some instructions on getting started.
- `uninstall.exe`: This executable file lets you uninstall the Android SDK.

The `tools` directory contains several useful tools. For example, I observed the following tools on my Windows platform:

- **android.bat**: This tool is used to create and update Android projects; update the SDK with new platforms and other items; and create/delete/view Android virtual devices.
- **emulator.exe**: This tool is used to start a virtual device that runs one of the installed Android platforms. This virtual device includes a set of preinstalled apps (such as Browser) that you can run.
- **sqlite3.exe**: This tool is used to manage the SQLite databases created by Android apps. (I introduced SQLite in Chapter 14.)

Installing the Android 4.4 Platform

By itself, the SDK is insufficient for developing Android apps. You also need to install an Android platform. At time of writing, the latest platform is Android 4.4 (KitKat) Revision 2.

The SDK Manager tool is used to install an Android platform. Double-click the `SDK Manager.exe` filename or type this name (surrounded by double quotes) on the command line. If SDK Manager doesn't display its Android SDK Manager window, you probably need to create a `JAVA_HOME` environment variable that points to your JDK's home directory, for example, `SET JAVA_HOME = C:\Program Files\Java\jdk1.7.0_06`.

■ **Note** You can use the `android.bat` tool to display the Android SDK Manager window. Double-click this filename or specify `android` at the command line.

Figure C-1 reveals the startup Android SDK Manager window.

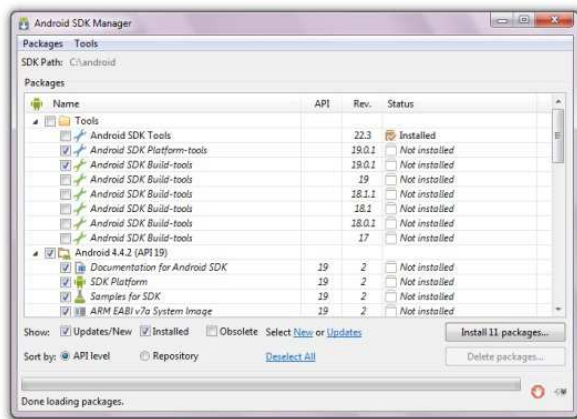


Figure C-1. Use this window to install, update, and remove Android packages and to run AVD Manager

The Android SDK Manager window presents a menu bar and a content area. The menu bar offers Packages and Tools menus:

- *Packages* provides menu items for showing a combination of updates/new packages, installed packages, obsolete packages, and archive details. It also provides menu items for sorting packages by API level or repository and reloading the list of packages that are shown in the content area.
- *Tools* provides menu items for managing Android virtual devices and add-on repositories, for specifying the proxy server and other options, and for displaying the About dialog box.

The content area reveals the path to the Android SDK. It also presents a table of information about packages, check boxes for filtering which packages to display, radio buttons for choosing how to sort packages, buttons for installing and deleting packages, and a progress bar that shows the progress while scanning repositories for package information.

The Packages table classifies packages as tools, Android platforms, and extras. Each category is associated with a check box that, when checked, selects all items in that category. Individual items can be deselected by unchecking their corresponding check boxes.

Tools are classified as SDK tools, SDK platform tools, and SDK build tools:

- *SDK tools* are the basic tools that are included in the SDK distribution file (i.e., the installer) and that are stored in the `tools` directory. This option is unchecked because the SDK tools have been installed.
- *SDK platform tools* are platform-specific tools (such as `adb.exe`) that are stored in one of the platform-specific subdirectories of the `platforms` directory. This option is checked because SDK platform tools have not been installed.
- *SDK build tools* are tools used for building apps. Examples include `aapt.exe` (Android Asset Packager Tool) and `dx.bat` (convert Java classfiles to Dalvik equivalent). Running SDK Manager for the first time results in the creation of an empty `build-tools` subdirectory of the Android home directory. This option is checked because SDK build tools have not been installed.

The only platform to be installed for this appendix is Android 4.4.2. When you run SDK Manager, you'll probably note that this platform category and all of its options are checked. Leave them checked.

Finally, you can install *extras*, which are external libraries or tools that can be included or used when building an app. For example, the Google USB driver is already checked in the Extras category. You only need to install this component when developing on a Windows platform and testing your apps on an actual Android device.

Click the Install 11 packages button (the number will differ should you install more or fewer packages). In response, you'll encounter Figure C-2's Choose Packages to Install dialog box.

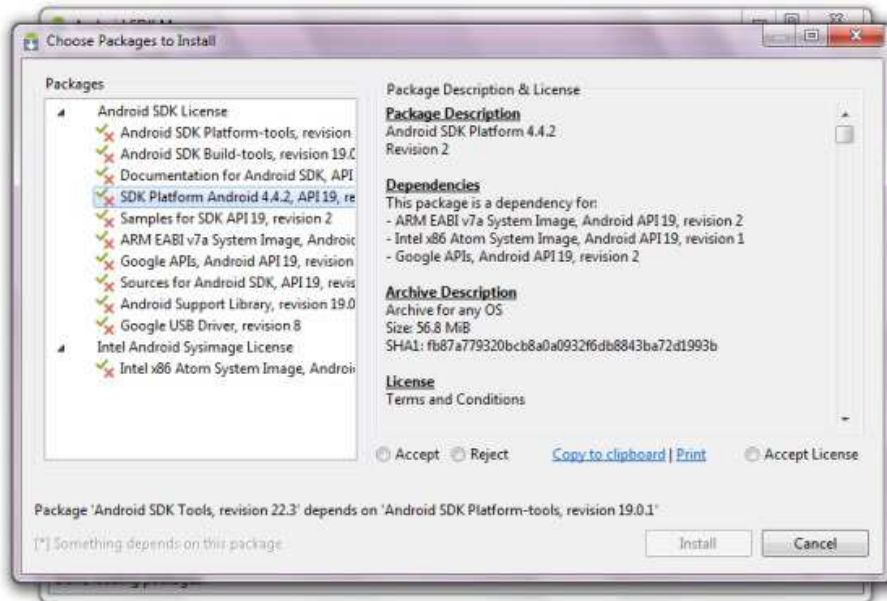


Figure C-2. The Packages list identifies those packages that can be installed

The Choose Packages to Install dialog box shows a Packages list that identifies those packages that can be installed. It displays green checkmarks and red Xs beside packages that have not yet been selected for installation. Also, it displays green checkmarks only beside those packages that have been accepted and displays red Xs only beside those packages that have been rejected.

For the highlighted package, Package Description & License presents a package description, a list of other packages that are dependent on this package, information on the archive that houses this package, and additional information. Click the Accept or Reject radio button to accept or reject the package.

Select Android SDK License at the top of the list and select the Accept License radio button to check every item in this section. Then click the Install button to begin installation. Android downloads and installs the chosen packages and you will see the Android SDK Manager Log dialog box, which presents messages that show what's happening. This dialog box is shown in Figure C-3.

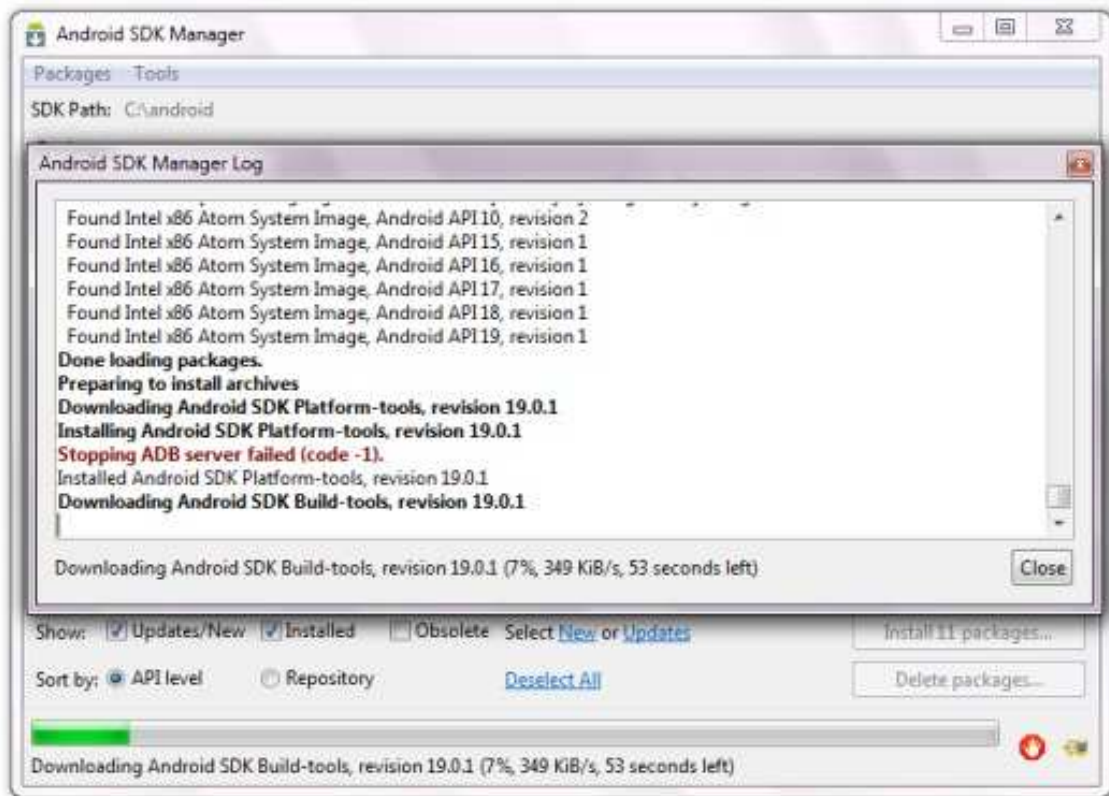


Figure C-3. The log reveals the progress of downloading and installing each package archive

Consider Figure C-3’s “Stopping ADB server failed (code -1)” message. ADB stands for *Android Debug Bridge*, which is a tool (accessed via `adb.exe`) consisting of client and server programs that let you control and interface with your Android device. This message appears because the ADB server isn’t presently running (and it doesn’t need to run at this point).

Upon completion, you should observe a “Done loading packages” message at the bottom of the Android SDK Manager Log dialog box and the Android SDK Manager window. Click the Close button on the dialog box. The Status column in the Packages table of the Android SDK Manager window should reveal which packages have been installed.

You should also observe several new subdirectories of the home directory and its directories, including the following pair:

- `platform-tools` (the latest platform tools)
- `platforms\android-19` (Android 4.4.2-specific files)

Creating an Android Virtual Device

After installing an Android platform, you need to create an *Android virtual device (AVD)*, which is a device configuration that represents an Android device. The emulator tool works with AVDs.

The AVD Manager tool lets you create and otherwise manage AVDs. You can run this tool directly or via SDK Manager (select Manage AVDs from the Tools menu). Figure C-4 shows the main window.

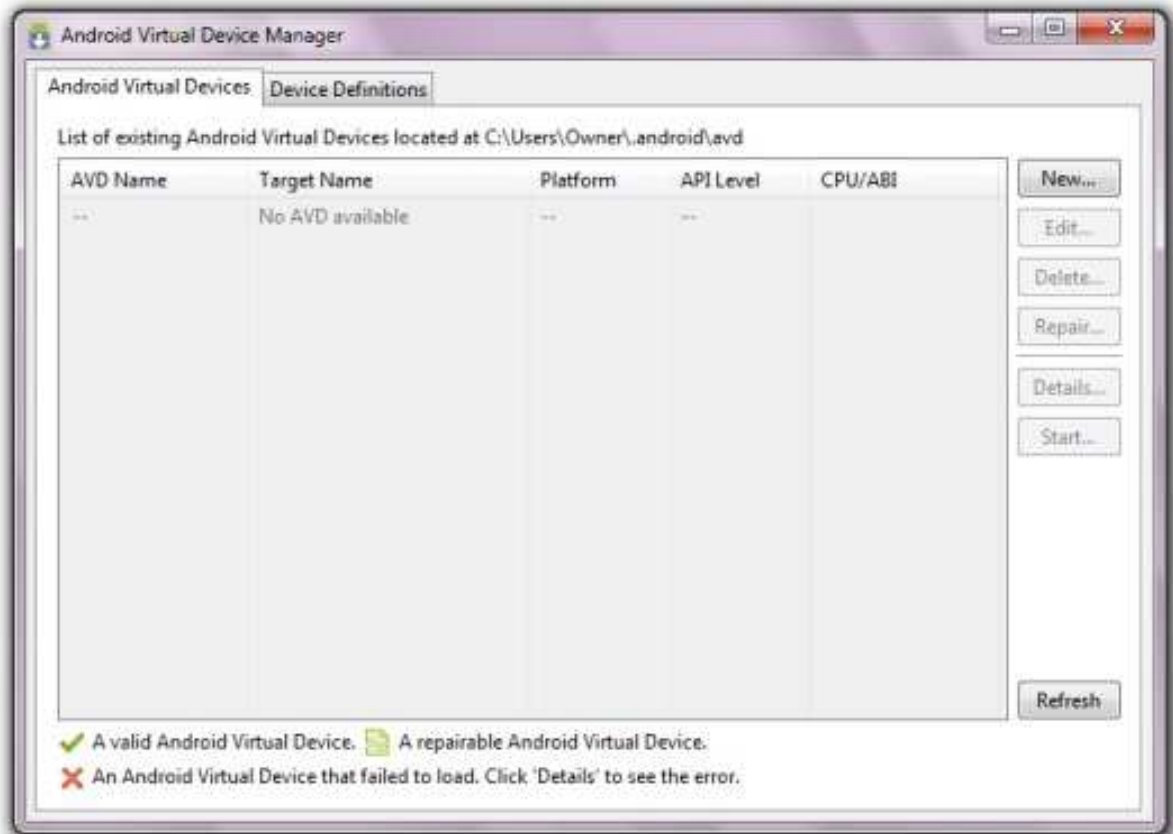


Figure C-4. No AVDs are initially installed

Click the New button. Figure C-5 shows the resulting Create new Android Virtual Device (AVD) dialog box.

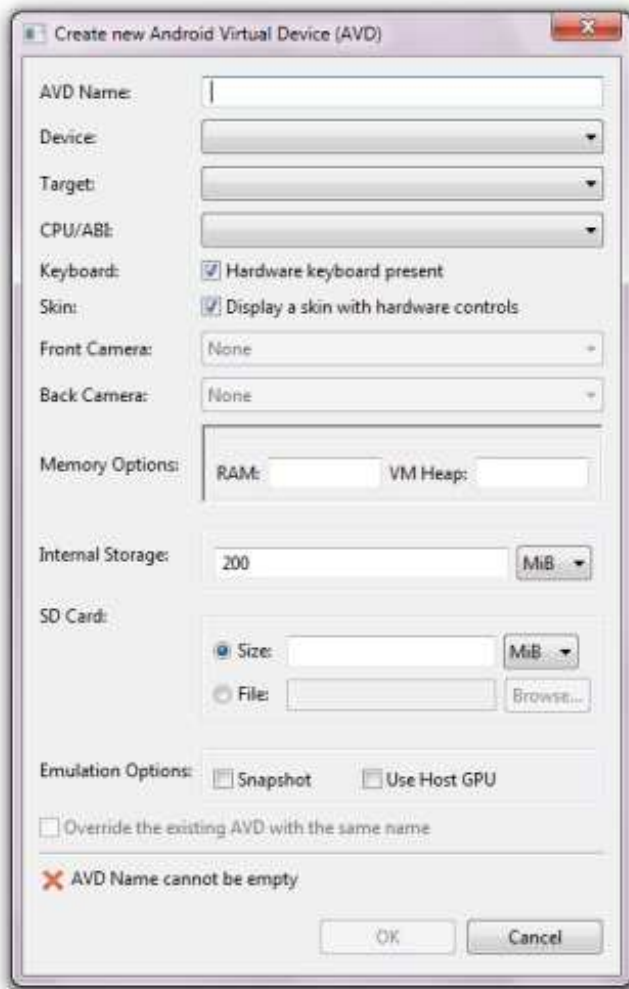


Figure C-5. An AVD consists of a name, a target platform, and other details

Figure C-5 reveals that an AVD has a mandatory name, targets a specific Android device and platform, targets a specific CPU/Application Binary Interface (such as ARM/armeabi-v7a), can emulate an SDK card, and offers other options.

Enter **MyAVD** into the AVD Name text field. Also, select 3.2" QVGA (ADP2) (320 x 480; mdpi) from the Device dropdown list box, and select Android 4.4.2 - API Level 19 from the Target dropdown list box. Finally, enter **100** into the Size text field in the SD Card section and check Use Host GPU in the Emulation Options section. This option lets you improve graphics performance. Keep the other defaults and click OK. You should observe a summary window such as the window appearing in Figure C-6.

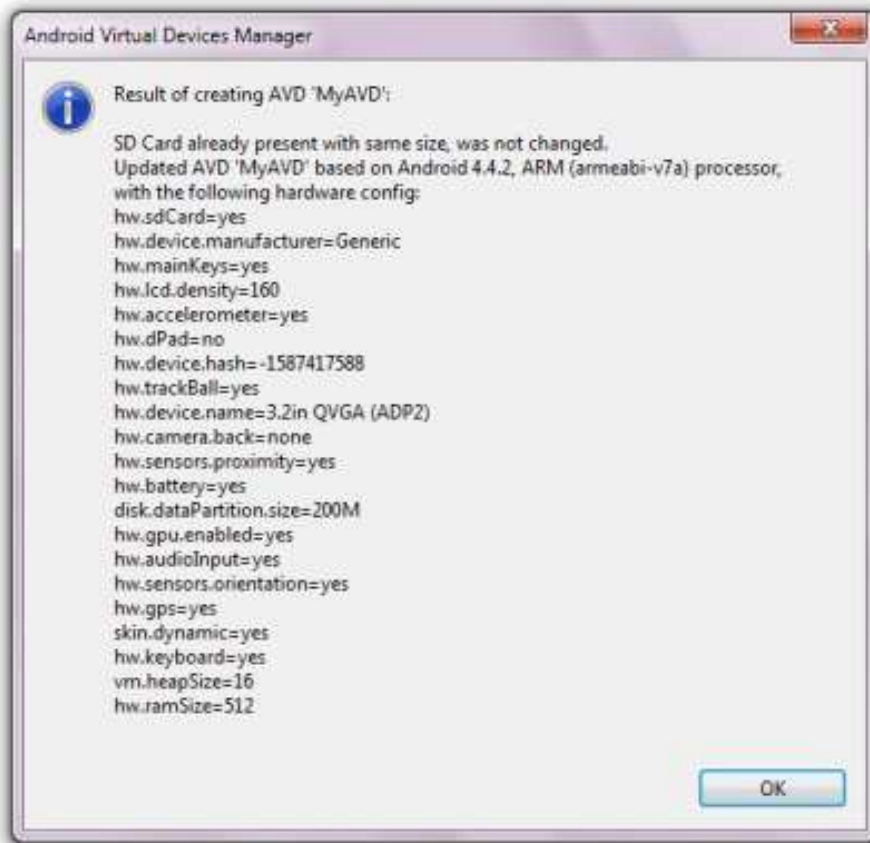


Figure C-6. The summary window reveals the AVD's hardware configuration

■ **Caution** When creating an AVD that you plan to use for testing an app, make sure that the target platform has an API level that's at least as great as the app's API level. An app typically cannot access platform APIs that are more recent than those APIs supported by the AVD's API level.

Although it's easier to use AVD Manager to create an AVD, you can also accomplish this task via the `android` tool by specifying the following syntax:

```
android create avd -n name -t targetID [-option value] ...
```

Given this syntax, *name* identifies the AVD, *targetID* is an integer ID that identifies the targeted Android platform (you can obtain this ID by executing `android list targets`), and `[-option value]` ... identifies a series of options (such as SD card size).

If you don't specify sufficient options, `android` prompts you to create a custom hardware profile. Press the Enter key when you don't want to create a custom profile and prefer to use the default hardware emulator options. For example, the following command causes an AVD named AVD1 to be created:

```
android create AVD -n AVD1 -t 1
```

This command assumes that 1 corresponds to the Android 4.4.2 platform. It also prompts to create a custom hardware profile.

■ **Note** Each AVD functions as an independent device descriptor, specifying its own private storage for user data, its own SD card, and more. When you launch the emulator tool with an AVD, this tool loads user data and SD card data from the AVD's directory. By default, the emulator stores the user data, the SD card data, and configuration information in the directory assigned to the AVD.

Starting and Exploring the Emulated Device

Perhaps the easiest way to start an emulated device is to use AVD Manager. Refer back to Figure C-4, which presents this tool's main window. If you view this window after creating the AVD in the previous section, you should observe an AVD1 entry. Highlight this entry and the Start button will no longer be disabled.

Click Start to run the emulator with the AVD1 descriptor. You should observe the Launch Options dialog box shown in Figure C-7.

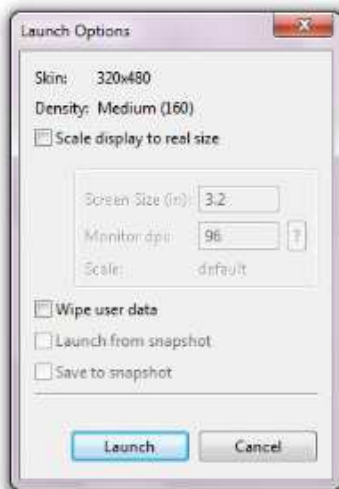


Figure C-7. The launch options dialog box lets perform launch-specific tasks prior to launching an emulated device

The Launch Options dialog box identifies the AVD's skin and screen density. It also provides unchecked check boxes for scaling the resolution of the emulator's display to match the physical device's screen size, for wiping user data, for launching from a previously saved snapshot, and for saving device state to a snapshot on device exit.

■ **Note** As you update your apps, you'll periodically package and install them on the emulator, which preserves the apps and their state data across AVD restarts in a user-data disk partition. To ensure that an app runs properly as you update it, you might need to delete the emulator's user-data partition, which is accomplished by checking Wipe user data.

Click the Launch button to launch the emulator with AVD1. AVD Manager responds by briefly displaying a Starting Android Emulator dialog box followed by the emulator window. See Figure C-8.



Figure C-8. The emulator window (with a QVGA handset skin) presents the home screen on its left and phone controls on its right

Figure C-8 shows that the emulator window is divided into a left pane, which displays the Android logo on a black background followed by the *home screen* (a special app that's a starting point for using an Android device), and a right pane, which displays phone controls.

A status bar appears above the home screen (and every app screen). The status bar presents the current time, amount of battery power remaining, and other information; it also provides access to notifications.

The home screen initially appears in locked mode. To unlock this screen drag the lock icon to its right until it touches an unlock icon (or press the MENU button). You should end up with the unlocked home screen shown in Figure C-9.



Figure C-9. The home screen now reveals the app launcher and more

The home screen presents the following items:

- *Wallpaper background:* Wallpaper appears behind everything else and can be dragged to the left or right. To change this background, press and hold down the left mouse button over the wallpaper, which causes a wallpaper-oriented pop-up menu to appear.

- *Widgets:* The Google Search widget appears near the top and the Clock widget appears upper-centered. A *widget* is a miniature app view that can be embedded in the home screen and other apps and receives periodic updates.
- *App launcher:* The app launcher (along the bottom) presents icons for launching the commonly used Phone, Contacts, Messaging, and Browser apps; it also displays a rectangular grid of all installed apps, which are subsequently launched by single-clicking their icons. Figure C–10 shows some of these icons.



Figure C–10. Drag this screen to the left to reveal more icons

The app launcher organizes apps and widgets according to the tabs near the top left of the screen. You can run apps from the APPS tab and select additional widgets to display on the home screen from the WIDGETS tab. (If you need more room for widgets on the home screen, drag its wallpaper in either direction.)

■ **Tip** The API Demos app demonstrates a wide variety of Android APIs. If you're new to Android app development, you should run the individual demos to acquaint yourself with what Android has to offer. You can view each demo's source code by accessing the source files that are located in subdirectories of the home directory's `samples/android-19` directory.

The phone controls include the following commonly-used buttons:

- The house icon phone control button takes you from wherever you are to the home screen.
- The MENU phone control button presents a menu of app-specific choices for the currently running app.
- The curved arrow phone control button takes you back to the previous activity in the activity stack—I discuss both topics later in this appendix.

While the emulated device is running, you can interact with it by using your mouse to “touch” the touchscreen and your keyboard to “press” the keys of the device's (typically onscreen) keyboard. Table C–1 shows you the mapping between AVD keys and keyboard keys:

Table C–1. Mappings between AVD Keys and Keyboard Keys

AVD Key	Keyboard Key
Home	HOME
Menu (left softkey)	F2 or Page Up
Menu (right softkey)	Shift-F2 or Page Down
Back	ESC
Call/dial button	F3
Hangup/end call button	F4
Search	F5
Power button	F7
Audio volume up button	KEYPAD_PLUS, Ctrl-F5
Audio volume down button	KEYPAD_MINUS, Ctrl-F6
Camera button	Ctrl-KEYPAD_5, Ctrl-F3

AVD Key	Keyboard Key
Switch to previous layout orientation (for example, portrait or landscape)	KEYPAD_7, Ctrl-F11
Switch to next layout orientation	KEYPAD_9, Ctrl-F12
Toggle cell networking on/off	F8
Toggle code profiling	F9 (only with the -trace startup option)
Toggle fullscreen mode	Alt-Enter
Toggle trackball mode	F6
Enter trackball mode temporarily (while key is pressed)	Delete
DPAD left/up/right/down	KEYPAD_4/8/6/2
DPAD center click	KEYPAD_5
Onion alpha increase/decrease	KEYPAD_MULTIPLY (*)/KEYPAD_DIVIDE (/)

■ **Tip** You must first disable NumLock on your development computer before you can use keypad keys.

Table C–1 refers to the -trace startup option in the context of toggle code profiling. This option lets you store profiling results in a file when starting the emulated device. For example, the following command starts the emulator for device configuration MyAVD, and it also stores profiling results in results.txt when you press F9—press F9 again to stop code profiling.

```
emulator -avd MyAVD -trace results.txt
```

Figures C–8 through C–10 display 5554:MyAVD on the title bar. Value 5554 identifies a console port for dynamically querying and otherwise controlling the environment of the emulated device.

■ **Note** Android supports up to 16 concurrently executing emulated devices. Each device is assigned an even-numbered console port number starting with 5554.

On Windows platforms, you can connect to the device’s console by specifying the telnet command according to the following command syntax:

```
telnet localhost console-port
```

For example, the following command connects to MyAVD’s console:


```
telnet localhost 5554
```

Figure C-11 shows you the resulting command window on Windows 7.

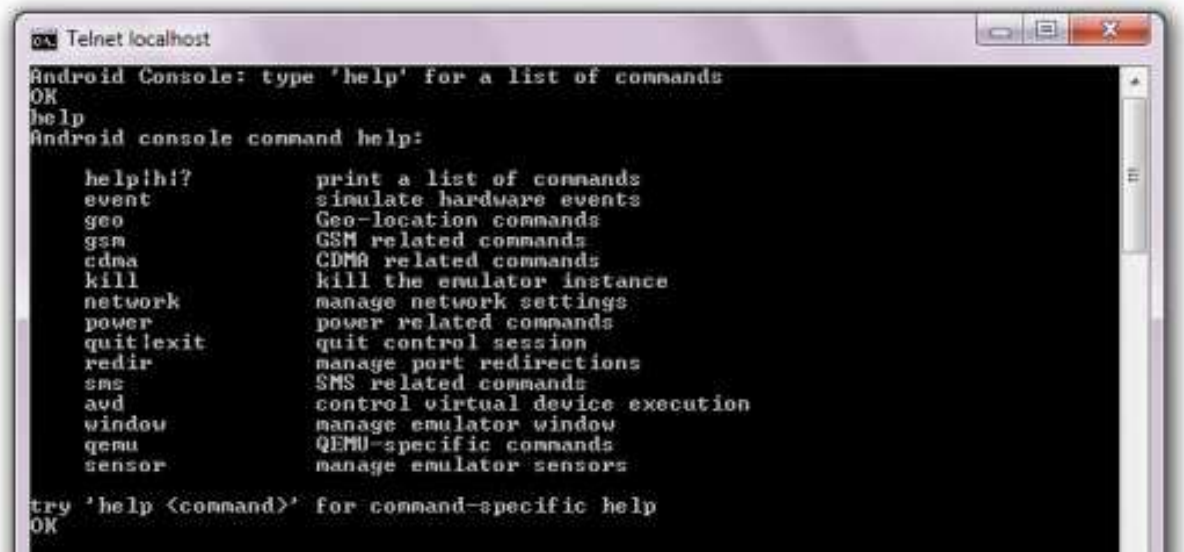


Figure C-11. Type a command name by itself for command-specific help

■ **Tip** The telnet command is disabled on Windows 7 by default (to help make this operating system more secure). To enable telnet on Windows 7, start the control panel, select Programs and Features, select Turn Windows features on or off, and (from the Windows Features dialog box) check the Telnet Client check box.

Configuring Eclipse for Android App Development

In Chapter 1 I showed you how to install Eclipse Standard 4.3.1. In this section I show you how to configure this IDE to properly support the development of Android apps. You might find it easier to develop apps with Eclipse rather than at the command line. You'll explore both approaches later in this appendix.

To develop Android apps with Eclipse, you should install the Android Development Tools (ADT) Plugin. Although you can use Eclipse without the ADT Plugin to develop these apps, it's much faster and easier to create, debug, and otherwise develop these apps with this plug-in. The ADT Plugin offers the following features:

- It gives you access to other Android development tools from inside the Eclipse IDE. For example, ADT lets you access the many capabilities of the Dalvik Debug Monitor Server (DDMS) tool, allowing you to take screenshots, manage port-forwarding, set breakpoints, and view thread and process information directly from Eclipse.
- It provides a New Project wizard that helps you quickly create and set up all of the basic files you'll need for a new Android app.
- It automates and simplifies the process of building your Android app.
- It provides an Android code editor that helps you write valid XML for your Android manifest and resource files.
- It lets you export your project into a signed APK (discussed later), which can be distributed to users.

Complete the following steps to install the latest revision of the ADT Plugin:

1. Start Eclipse and choose your workspace.
2. Select Install New Software from the Help menu.
3. Click the Add button on the resulting Install dialog box's Available Software pane.
4. On the resulting Add Repository dialog box, enter a name for the remote site (for example, **Android Plugin**) into the Name field, and enter **<https://dl-ssl.google.com/android/eclipse/>** into the Location field. Click OK.
5. You should now see Developer Tools and NDK Plugins in the list that appears in the middle of the Install dialog box. Check the check box next to these categories, which will automatically check the nested items underneath. Click Next.
6. The resulting Install Details pane lists Android DDMS, Android Development Tools, Android Hierarchy Viewer, Android Native Development Tools, Android TraceView, and Tracer for OpenGL ES. Click Next to read and accept the various license agreements, and then click Finish.
7. An Installing Software dialog box appears and takes care of installation. If you encounter a Security Warning dialog box, click OK.
8. Finally, Eclipse presents a Software Updates dialog box that prompts you to restart this IDE. Click the Yes button to restart.

■ **Tip** If you have trouble acquiring the plug-in in Step 4, try specifying `http` instead of `https` (`https` is preferred for security reasons) in the Location field.

To complete the installation of the ADT Plugin, you may have to configure this plug-in by modifying the ADT preferences in Eclipse to point to the Android SDK home directory. Accomplish this task by completing the following steps:

1. Select Preferences from the Window menu to open the Preferences dialog box. For Mac OS X, select Preferences from the Eclipse menu.
2. Select Android from the left panel.
3. If the SDK Location text field presents the SDK's home directory (such as `C:\android`), close the Preferences dialog box. You have nothing further to do.
4. If the SDK Location text field doesn't present the SDK's home directory, click the Browse button beside this text field and locate your downloaded SDK's home directory on the resulting Browse For Folder dialog box. Select this location, click OK to close this dialog box, and click Apply on the Preferences dialog box to confirm this location, which should result in a list of SDK targets (such as Android 4.4.2) appearing below the text field.

■ **Note** For more information on installing the ADT Plugin, which includes helpful information in case of difficulty, check out the "Installing the Eclipse Plugin" page (<http://developer.android.com/sdk/installing/installing-adt.html>) in Google's online Android documentation.

Learning Android App Architecture

In addition to creating a development environment, you also need to understand the architecture of Android apps before you can develop them. This architecture largely consists of components (activities, services, broadcast receivers, and content providers—along with intents); views, view groups, and event listeners; resources; the manifest; and the app package.

Learning Components

An app consists of *components* (activities, services, broadcast receivers and content providers) that run in a Linux process and that are managed by Android:

- Activities present user interface screens.
- Services perform lengthy jobs (such as playing music) in the background and don't provide user interfaces.
- Broadcast receivers receive and react to broadcasts from Android or other components.
- Content providers encapsulate data and make it available to apps.

Each component is implemented as a class that's stored in the same Java package, which is known as the *app package*. From the Android SDK perspective, each class's source file is stored under a package directory hierarchy that's situated underneath a `src` directory.

Not all of these components need to be present in an app. For example, one app might consist of activities only, whereas another app might consist of activities and a service.

■ **Note** An app's activities, services, broadcast receivers and/or content providers share a set of system resources, such as databases, preferences, a filesystem, and the Linux process.

Android communicates with activities, services, and broadcast receivers via *intents*, which are messages that describe operations to perform (such as launch an activity), or (in the case of broadcasts) provide descriptions of external events that have occurred (a device's camera being activated, for example) and are being announced. Activities, services, and broadcast receivers can also use intents to communicate among themselves.

Intents are implemented as instances of the `android.content.Intent` class. An `Intent` object describes a message in terms of some combination of the following items:

- *Action*: A string naming the action to be performed or, in the case of broadcast intents, the action that took place and is being reported. Actions are described by `Intent` constants such as `ACTION_CALL` (initiate a phone call), `ACTION_EDIT` (display data for the user to edit), and `ACTION_MAIN` (start up as the initial activity). You can also define your own action strings for activating the components in your app. These strings should include the app package as a prefix ("`com.example.project.GET_NEWSFEEDS`", for example).
- *Category*: A string that provides additional information about the kind of component that should handle the intent. For example, `CATEGORY_LAUNCHER` means that the calling activity should appear in the device's app launcher as a top-level app.

- *Component name*: A string that specifies the fully qualified name (package plus name) of a component class to use for the intent. The component name is optional. If set, the Intent object is delivered to an instance of the designated class. If not set, Android uses other information in the Intent object to locate a suitable target.
- *Data*: The uniform resource identifier of the data on which to operate (such as a person record in a contacts database).
- *Extras*: A set of key-value pairs providing additional information that should be delivered to the component handling the intent. For example, given an action for sending an email message, this information could include the message's subject, body, and so on.
- *Flags*: Bit values that instruct Android on how to launch an activity (for example, which task the activity should belong to—tasks are discussed later in this appendix) and how to treat the activity after launch (for example, whether the activity can be considered a recent activity). Flags are represented by constants in the Intent class; for example, FLAG_ACTIVITY_NEW_TASK specifies that this activity will become the start of a new task on this activity stack—the activity stack is discussed later in this appendix.
- *Type*: The MIME (<http://en.wikipedia.org/wiki/MIME>) type of the intent data. Normally, Android infers a type from the data. By specifying a type, you disable that inference.

Intents can be classified as explicit or implicit. An *explicit intent* designates the target component by its name (the previously mentioned component name item is assigned a value). Because component names are usually unknown to the developers of other apps, explicit intents are typically used for app-internal messages (such as an activity that launches another activity located in the same app). Android delivers an explicit intent to an instance of the designated target class. Only the Intent object's component name matters for determining which component should get the intent.

An *implicit intent* doesn't name a target (the component name isn't assigned a value). Implicit intents are often used to start components in other apps. Android searches for the best component (a single activity or service to perform the requested action) or components (a set of broadcast receivers to respond to the broadcast announcement) to handle the implicit intent. During the search, Android compares the contents of the Intent object to *intent filters*, manifest information associated with components that can potentially receive intents.

Filters advertise a component's capabilities and identify only those intents that the component can handle. They open up the component to the possibility of receiving implicit intents of the advertised type. If a component has no intent filters, it can receive only explicit intents. In contrast, a component with filters can receive explicit and implicit intents. Android consults an Intent object's action, category, data, and type when comparing the intent against an intent filter. It doesn't take extras and flags into consideration.

■ **Note** Android widely uses intents, which offer many opportunities to replace existing components with your own components. For example, Android provides the intent for sending an email. Your app can send this intent to activate the standard mail app, or it can register an activity that responds to the intent, replacing the standard mail app with its own activity.

This component-oriented architecture lets an app reuse the components of other apps, provided that those other apps permit reuse of their components. Component reuse reduces overall memory footprint, which is very important for devices with limited memory.

For example, you're creating a drawing app that lets users choose a color from a palette, and another app contains a suitable color chooser and permits this component to be reused. In this scenario, the drawing app can call upon that other app's color chooser to let the user select a color rather than provide its own color chooser. The drawing app doesn't contain the other app's color chooser or even link to this other app. Instead, it starts up the other app's color chooser component when needed.

■ **Note** Android starts a process when any part of the app (such as the aforementioned color chooser) is needed, and instantiates the Java objects for that part. This is why Android's apps don't have a single entry point (no C-style `main()` function, for example). Instead, apps use components that are instantiated and run as needed.

Activities in Depth

An *activity* is a component that presents a user interface screen with which the user interacts. For example, Android's Contacts app includes an activity for entering a new contact, its Phone app includes an activity for dialing a phone number, and its Calculator app includes an activity for performing basic calculations (see Figure C-12).

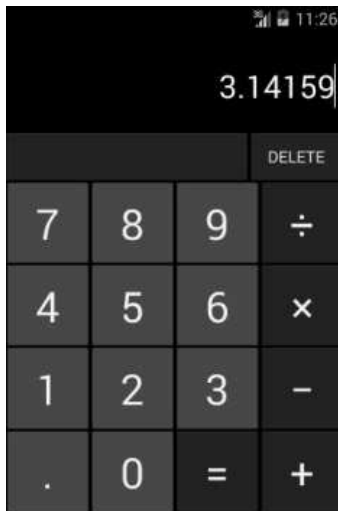


Figure C-12. The main activity of Android's Calculator app lets the user perform basic calculations

Although an app can include a single activity, it's more typical for apps to include multiple activities. For example, the Calculator app also includes an "advanced panel" activity that lets the user calculate square roots, perform trigonometry, and carry out other advanced mathematical operations.

Activities are described by subclasses of the `android.app.Activity` class, which is an indirect subclass of the `android.content.Context` class.

■ **Note** `Context` is an abstract class whose methods let apps access global information about their environments (such as their resources and filesystems), and let apps perform contextual operations, such as launching activities and services, broadcasting intents, and opening private files.

Activity subclasses override various Activity *life cycle callback methods* that Android calls during the life of an activity. For example, the `SimpleActivity` class in Listing C-1 extends `Activity` and overrides its `void onCreate(Bundle bundle)` and `void onDestroy()` life cycle callback methods.

Listing C-1. A Skeletal Activity

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

public class SimpleActivity extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
```

```
    super.onCreate(savedInstanceState); // Always call superclass method first.  
    System.out.println("onCreate(Bundle) called");  
}  
  
@Override  
public void onDestroy()  
{  
    super.onDestroy(); // Always call superclass method first.  
    System.out.println("onDestroy() called");  
}  
}
```

The overriding `onCreate(Bundle)` and `onDestroy()` methods in Listing C–1 first invoke their superclass counterparts, a pattern that must be followed when overriding the `void onStart()`, `void onRestart()`, `void onResume()`, `void onPause()`, and `void onStop()` life cycle callback methods.

- `onCreate(Bundle)` is called when the activity is first created. This method is used to create the activity's user interface, create background threads as needed, and perform other global initialization. `onCreate()` is passed an `android.os.Bundle` object containing the activity's previous state, if that state was captured (via `void onSaveInstanceState(Bundle outState)`); otherwise, the null reference is passed. Android always calls the `onStart()` method after calling `onCreate(Bundle)`. All meaningful activities override `onCreate(Bundle)`.
- `onStart()` is called just before the activity becomes visible to the user. Android calls the `onResume()` method after calling `onStart()` when the activity comes to the foreground, and calls the `onStop()` method after `onStart()` when the activity becomes hidden.
- `onRestart()` is called after the activity has been stopped, just prior to it being started again. Android always calls `onStart()` after calling `onRestart()`.
- `onResume()` is called just before the activity starts interacting with the user. At this point the activity has the focus and user input is directed to the activity. Android always calls the `onPause()` method after calling `onResume()`, but only when the activity must be paused.
- `onPause()` is called when Android is about to resume another activity. This method is typically used to save unsaved changes, stop animations that might be consuming processor cycles, and so on. It should perform its job quickly because the next activity won't be resumed until it returns. Android calls `onResume()` after calling `onPause()` when the activity starts interacting with the user, and calls `onStop()` when the activity becomes invisible to the user. Many activities implement `onPause()` to commit data changes and otherwise prepare to stop interacting with the user.

- `onStop()` is called when the activity is no longer visible to the user. This may happen because the activity is being destroyed or because another activity (either an existing one or a new one) has been resumed and is covering the activity. Android calls `onRestart()` after calling `onStop()`, when the activity is coming back to interact with the user, and it calls the `onDestroy()` method when the activity is going away.
- `onDestroy()` is called before the activity is destroyed unless memory is tight and Android is forced to kill the activity's process. In this scenario, `onDestroy()` is never called. If `onDestroy()` is called, it will be the final call that the activity ever receives. Android can kill the process hosting the activity at any time after `onPause()`, `onStop()`, or `onDestroy()` returns. An activity is in a killable state from the time `onPause()` returns until the time `onResume()` is called. The activity won't again be killable until `onPause()` returns.

Figure C–13 illustrates an activity's life cycle in terms of these seven methods.

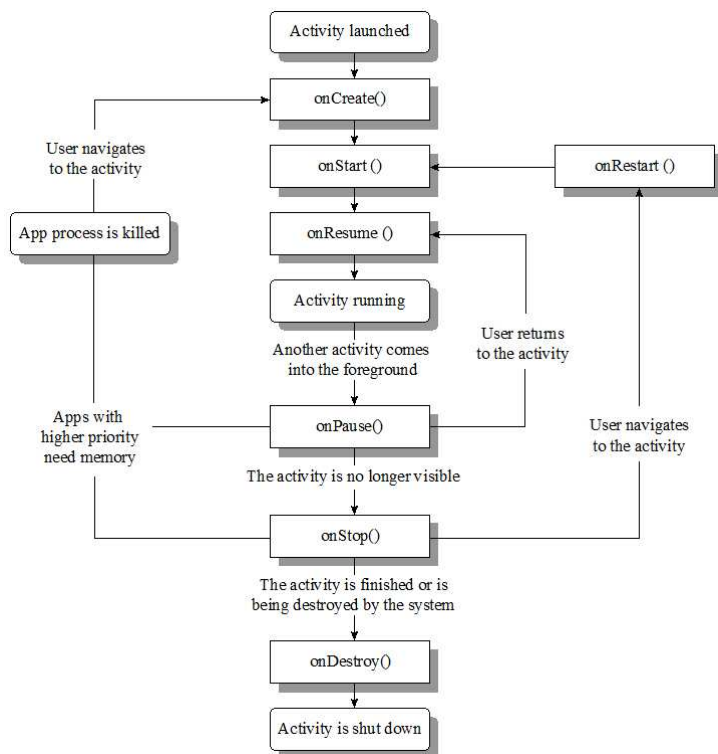


Figure C–13. The life cycle of an activity reveals that there's no guarantee of `onDestroy()` being called

Figure C–13 reveals that an activity is started by calling `startActivity()`. More specifically, the activity is started by creating an `Intent` object describing an explicit or implicit intent and by passing this object to `Context`'s `void startActivity(Intent intent)` method (launch a new activity; no result is returned when it finishes).

Alternatively, the activity could be started by calling `Activity`'s `void startActivityForResult(Intent intent, int requestCode)` method. The specified `int` result is returned to `Activity`'s `void onActivityResult(int requestCode, int resultCode, Intent data)` callback method as an argument.

■ **Note** The responding activity can look at the intent that caused it to be launched by calling `Activity`'s `Intent getIntent()` method. Android calls the activity's `void onNewIntent(Intent intent)` method (also located in the `Activity` class) to pass any subsequent intents to the activity.

Suppose that you've created an app named `SimpleActivity`, and that this app consists of `SimpleActivity` (described in Listing C–1) and `SimpleActivity2` classes. Now suppose that you want to launch `SimpleActivity2` from `SimpleActivity`'s `onCreate(Bundle)` method. The following example shows you how to start `SimpleActivity2`:

```
Intent intent = new Intent(SimpleActivity.this, SimpleActivity2.class);
SimpleActivity.this.startActivity(intent);
```

The first line creates an `Intent` object that describes an explicit intent. It initializes this object by passing the current `SimpleActivity` instance's reference and `SimpleActivity2`'s `Class` instance to the `Intent(Context packageContext, Class<?> cls)` constructor.

The second line passes this `Intent` object to `startActivity(Intent)`, which is responsible for launching the activity described by `SimpleActivity2.class`. If `startActivity(Intent)` was unable to find the specified activity (which shouldn't happen), it would throw an `android.content.ActivityNotFoundException` instance.

Figure C–13 also reveals that `onDestroy()` might not be called before the app is terminated. As a result, you should not count on using this method as a place for saving data. For example, if an activity is editing a content provider's data, those edits should typically be committed in `onPause()`.

■ **Note** `onDestroy()` is usually implemented to free system resources (such as threads) that were acquired in `onCreate(Bundle)`.

The seven life cycle callback methods define an activity's entire life cycle and describe the following three nested loops:

- The *entire lifetime* of an activity is defined as everything from the first call to `onCreate(Bundle)` through to a single final call to `onDestroy()`. An activity performs all of its initial setup of “global” state in `onCreate(Bundle)`, and it releases all remaining resources in `onDestroy()`. For example, if the activity has a thread running in the background to download data from the network, it might create that thread in `onCreate(Bundle)` and stop the thread in `onDestroy()`.
- The *visible lifetime* of an activity is defined as everything from a call to `onStart()` through to a corresponding call to `onStop()`. During this time, the user can see the activity onscreen, although it might not be in the foreground and interacting with the user. Between these two methods, the activity can maintain system resources that are needed to show itself to the user. For example, it can register a broadcast receiver in `onStart()` to monitor for changes that impact its user interface, and it can unregister this object in `onStop()` when the user can no longer see what the activity is displaying. The `onStart()` and `onStop()` methods can be called multiple times as the activity alternates between being visible to and being hidden from the user.
- The *foreground lifetime* of an activity is defined as everything from a call to `onResume()` through to a corresponding call to `onPause()`. During this time, the activity is in front of all other activities onscreen and is interacting with the user. An activity can frequently transition between the resumed and paused states; for example, `onPause()` is called when the device goes to sleep or when a new activity is started, and `onResume()` is called when an activity result or a new intent is delivered. The code in these two methods should be fairly lightweight.

ACTIVITIES, TASKS, AND THE ACTIVITY STACK

Android refers to a sequence of related activities as a *task* and provides an *activity stack* (also known as *history stack* or *back stack*) to remember this sequence. The activity starting the task is the initial activity pushed onto the stack and is known as the *root activity*. This activity is typically the activity selected by the user via the device’s app launcher. The activity that’s currently running is located at the top of the stack.

When the current activity starts another, the new activity is pushed onto the stack and takes focus (becomes the running activity). The previous activity remains on the stack but is stopped. When an activity stops, the system retains the current state of its user interface.

When the user presses the device’s BACK control, the current activity is popped from the stack (the activity is destroyed), and the previous activity resumes operation as the running activity (the previous state of its user interface is restored).

Activities in the stack are never rearranged, only pushed and popped from the stack. Activities are pushed onto the stack when started by the current activity, and they are popped off the stack when the user leaves them by pressing the BACK control. As such, the stack operates as a “last in, first out” object structure.

Each time the user presses BACK, an activity in the stack is popped off to reveal the previous activity. This continues until the user returns to the home screen or to whichever activity was running when the task began. When all activities are removed from the stack, the task no longer exists.

Check out the “Tasks and Back Stack” section in Google’s online Android documentation to learn more about activities and tasks. You’ll find this documentation located at <http://developer.android.com/guide/components/tasks-and-back-stack.html>.

Services In Depth

A *service* is a component that runs in the background for an indefinite period of time and that doesn’t provide a user interface. As with an activity, a service runs on the process’s main thread; it must spawn another thread to perform a time-consuming operation. Services are classified as local or remote:

- A *local service* runs in the same process as the rest of the app. Such services make it easy to implement background tasks.
- A *remote service* runs in a separate process. Such services let you perform interprocess communications.

■ **Note** A service isn’t a separate process, although it can be specified to run in a separate process. Also, a service isn’t a thread. Instead, a service lets the app tell Android about something it wants to be doing in the background (even when the user isn’t directly interacting with the app), and lets the app expose some of its functionality to other apps.

Consider a service that plays music in response to a user’s music choice via an activity. The user selects the song to play via this activity, and a service is started in response to the selection. The service plays the music on another thread to prevent an “Application Not Responding” dialog box from appearing.

■ **Note** The rationale for using a service to play the music is that the user expects the music to keep playing even after the activity that initiated the music leaves the screen.

Services are described by subclasses of the abstract `android.app.Service` class, which is an indirect subclass of `Context`.

Service subclasses override various Service life cycle callback methods that Android calls during the life of a service. For example, the `SimpleService` class in Listing C-2 extends `Service` and also overrides the `void onCreate()` and `void onDestroy()` life cycle callback methods.

Listing C-2. A Skeletal Service, Version 1

```
import android.app.Service;

public class SimpleService extends Service
{
    @Override
    public void onCreate()
    {
        System.out.println("onCreate() called");
    }

    @Override
    public void onDestroy()
    {
        System.out.println("onDestroy() called");
    }

    @Override
    public IBinder onBind(Intent intent)
    {
        System.out.println("onBind(Intent) never called");
        return null;
    }
}
```

`onCreate()` is called when the service is initially created, and `onDestroy()` is called when the service is being removed. Because it's abstract, the `IBinder onBind(Intent intent)` life cycle callback method (described later in this section) must always be overridden, even if only to return `null`, which indicates that this method is ignored.

■ **Note** Service subclasses typically override `onCreate()` and `onDestroy()` to perform initialization and cleanup. Unlike `Activity`'s `onCreate(Bundle)` and `onDestroy()` methods, `Service`'s `onCreate()` method isn't repeatedly called and its `onDestroy()` method is always called.

A service's lifetime happens between the time `onCreate()` is called and the time `onDestroy()` returns. As with an activity, a service initializes in `onCreate()` and cleans up in `onDestroy()`. For example, a music playback service could create the thread that plays music in `onCreate()` and stop the thread in `onDestroy()`.

Local Services

Local services are typically started via Context's `ComponentName` `startService(Intent intent)` method, which returns an `android.content.ComponentName` instance that identifies the started service component, or the null reference when the service doesn't exist. Furthermore, `startService(Intent)` results in the life cycle shown in Figure C-14.

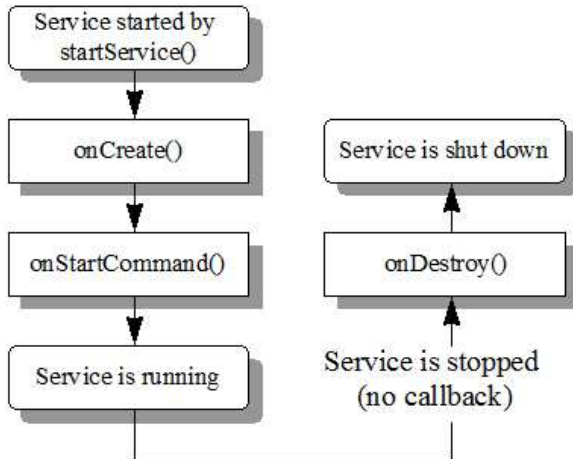


Figure C-14. The life cycle of a service that's started by `startService(Intent)` features a call to `onStartCommand(Intent, int, int)`

The call to `startService(Intent)` results in a call to `onCreate()`, followed by a call to `onStartCommand(Intent intent, int flags, int startId)`. This latter life cycle callback method, which replaces the deprecated `void onStart(Intent intent, int startId)` method, is called with the following arguments:

- `intent` is the `Intent` object passed to `startService(Intent)`.
- `flags` can provide additional data about the start request but is often set to 0.
- `startID` is a unique integer that describes this start request. A service can pass this value to Service's boolean `stopSelfResult(int startId)` method to stop itself.

`onStartCommand(Intent, int, int)` processes the `Intent` object, and typically it returns the constant `Service.START_STICKY` to indicate that the service is to continue running until explicitly stopped. At this point, the service is running and will continue to run until one of the following events occurs:

- Another component stops the service by calling Context's boolean `stopService(Intent intent)` method. Only one `stopService(Intent)` call is needed no matter how often `startService(Intent)` was called.
- The service stops itself by calling one of Service's overloaded `stopSelf()` methods or by calling Service's `stopSelfResult(int)` method.

After `stopService(Intent)`, `stopSelf()`, or `stopSelfResult(int)` has been called, Android calls `onDestroy()` to let the service perform cleanup tasks.

■ **Note** When a service is started by calling `startService(Intent)`, `onBind(Intent)` isn't called.

Listing C-3 presents a skeletal service class that could be used in the context of the `startService(Intent)` method.

Listing C-3. A Skeletal Service, Version 2

```
import android.app.Service;

public class SimpleService extends Service
{
    @Override
    public void onCreate()
    {
        System.out.println("onCreate() called");
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId)
    {
        System.out.println("onStartCommand(Intent, int, int) called");
        return START_STICKY;
    }

    @Override
    public void onDestroy()
    {
        System.out.println("onDestroy() called");
    }

    @Override
    public IBinder onBind(Intent intent)
    {
        System.out.println("onBind(Intent) never called");
        return null;
    }
}
```

The following example, which is assumed to be located in the `onCreate()` method of Listing C-1's `SimpleActivity` class, employs `startService(Intent)` to start an instance of Listing C-3's `SimpleService` class via an explicit intent:

```
Intent intent = new Intent(SimpleActivity.this, SimpleService.class);
SimpleActivity.this.startService(intent);
```

Remote Services

Remote services are started via Context's boolean `bindService(Intent service, ServiceConnection conn, int flags)` method, which connects to a running service (creating the service if necessary) and which returns true when successfully connected. `bindService(Intent, ServiceConnection, int)` results in Figure C-15's life cycle.

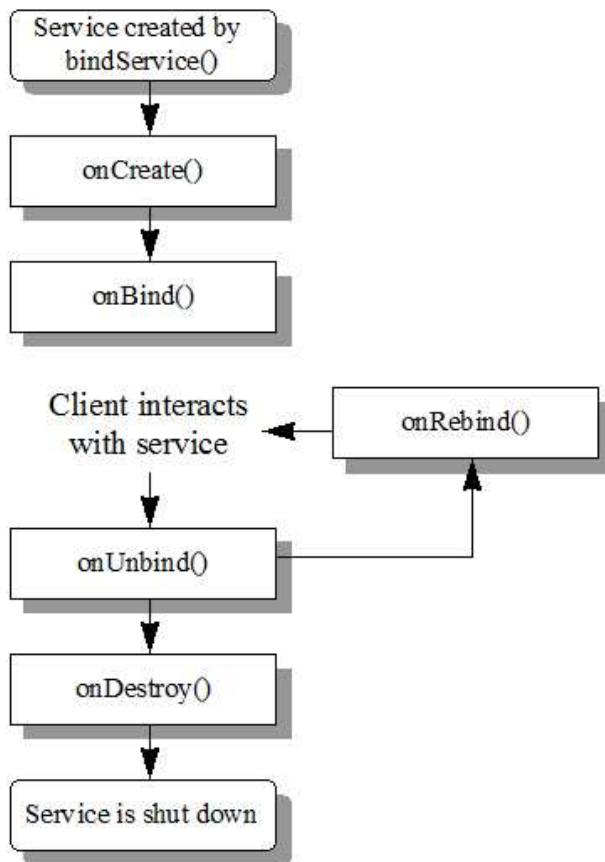


Figure C-15. The life cycle of a service started by `bindService(Intent, ServiceConnection, int)` doesn't include a call to `onStartCommand(Intent, int, int)`

The call to `bindService(Intent, ServiceConnection, int)` results in a call to `onCreate()` followed by a call to `onBind(Intent)`, which returns the *communications channel* (an instance of a class that implements the `android.os.IBinder` interface) that clients use to interact with the service.

The client interacts with the service as follows:

1. The client subclasses `android.content.ServiceConnection` and overrides this class's abstract `void onServiceConnected(ComponentName className, IBinder service)` and `void onServiceDisconnected(ComponentName name)` methods to receive information about the service as the service is started and stopped. When `bindService(Intent, ServiceConnection, int)` returns true, the former method is called when a connection to the service has been established; the `IBinder` argument passed to this method is the same value returned from `onBind(Intent)`. The latter method is called when a connection to the service has been lost.

Lost connections typically occur when the process hosting the service has crashed or has been killed. The `ServiceConnection` instance itself isn't removed—the binding to the service will remain active, and the client will receive a call to `onServiceConnected(ComponentName, IBinder)` when the service is next running.

2. The client passes the `ServiceConnection` subclass object to `bindService(Intent, ServiceConnection, int)`.

A client disconnects from a service by calling Context's `void unbindService(ServiceConnection conn)` method. This component no longer receives calls as the service is restarted. When no other components are bound to the service, the service is allowed to stop at any time.

Before the service can stop, Android calls the service's boolean `onUnbind(Intent intent)` life cycle callback method with the `Intent` object that was passed to `unbindService(ServiceConnection)`. Assuming that `onUnbind(Intent)` doesn't return true, which tells Android to call the service's `void onRebind(Intent intent)` life cycle callback method each time a client subsequently binds to the service, Android calls `onDestroy()` to destroy the service.

Listing C-4 presents a skeletal service class that could be used in the context of the `bindService(Intent, ServiceConnection, int)` method.

Listing C-4. A Skeletal Service, Version 3

```
import android.app.Service;

public class SimpleService extends Service
{
    public class SimpleBinder extends Binder
    {
        SimpleService getService()
        {
```

```

        return SimpleService.this;
    }
}

private final IBinder binder = new SimpleBinder();

@Override
public IBinder onBind(Intent intent)
{
    return binder;
}

@Override
public void onCreate()
{
    System.out.println("onCreate() called");
}

@Override
public void onDestroy()
{
    System.out.println("onDestroy() called");
}
}

```

Listing C–4 first declares a `SimpleBinder` inner class that extends the `android.os.Binder` class. `SimpleBinder` declares a single `SimpleService` `getService()` method that returns an instance of the `SimpleService` subclass.

■ **Note** `Binder` works with the `IBinder` interface to support a remote procedure call mechanism for communicating between processes. Although this example assumes that the service is running in the same process as the rest of the app, `Binder` and `IBinder` are still required.

Listing C–4 next instantiates `SimpleBinder` and assigns the instance's reference to the private `binder` field. This field's value is returned from the subsequently overriding `onBind(Intent)` method.

Let's assume that the `SimpleActivity` class in Listing C–1 declares a private `SimpleService` field named `ss` (`private SimpleService ss;`). Continuing, let's assume that the following example is contained in `SimpleActivity`'s `onCreate(Bundle)` method:

```

ServiceConnection sc = new ServiceConnection()
{
    @Override
    public void onServiceConnected(ComponentName className, IBinder service)
    {
        ss = ((SimpleService.SimpleBinder) service).getService();
        System.out.println("Service connected");
    }
}

```

```

@Override
public void onServiceDisconnected(ComponentName className)
{
    ss = null; System.out.println("Service disconnected");
}
};
bindService(new Intent(SimpleActivity.this, SimpleService.class), sc,
            Context.BIND_AUTO_CREATE);

```

This example first creates an object from an anonymous subclass of `ServiceConnection`. The overriding `onServiceConnected(ComponentName, IBinder)` method uses the service argument to call `SimpleBinder`'s `getService()` method and save the result.

Although it must be present, the overriding `onServiceDisconnected(ComponentName)` method should never be called because `SimpleService` runs in the same process as `SimpleActivity`.

The example next passes the `ServiceConnection` subclass object, along with an intent identifying `SimpleService` as the intent's target and `Context.BIND_AUTO_CREATE` (create a persistent connection) to `bindService(Intent, ServiceConnection, int)`.

■ **Note** This example used `bindService(Intent, ServiceConnection, int)` to start a local service, but it's more typical to use this method to start a remote service.

A service can be started with `startService(Intent)` and have components bound to it with `bindService(Intent, ServiceConnection, int)`. Android keeps the service running until all components have unbound and/or the service stops itself or is stopped by another component (or by Android when memory is low and it must recover system resources).

Broadcast Receivers in Depth

A *broadcast receiver* is a component that receives and reacts to broadcasts. Many broadcasts originate in system code; for example, an announcement is made to indicate that the time zone has been changed or the battery power is low.

Apps can also initiate broadcasts. For example, an app may want to let other apps know that some data has finished downloading from the network to the device and is now available for them to use.

Broadcast receivers are described by classes that subclass the abstract `android.content.BroadcastReceiver` class and override `BroadcastReceiver`'s abstract `void onReceive(Context context, Intent intent)` method. For example, Listing C-5's `SimpleBroadcastReceiver` class extends `BroadcastReceiver` and overrides this method.

Listing C-5. A Skeletal Broadcast Receiver

```
public class SimpleBroadcastReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        System.out.println("onReceive(Context, Intent) called");
    }
}
```

You start a broadcast receiver by creating an `Intent` object and passing this object to any of `Context`'s broadcast methods (such as `Context`'s overloaded `sendBroadcast()` methods), which broadcast the message to all interested broadcast receivers.

The following example, which is assumed to be located in the `onCreate()` method of Listing C-1's `SimpleActivity` class, starts an instance of Listing C-5's `SimpleBroadcastReceiver` class:

```
Intent intent = new Intent(SimpleActivity.this, SimpleBroadcastReceiver.class);
intent.putExtra("message", "Hello, broadcast receiver!");
SimpleActivity.this.sendBroadcast(intent);
```

`Intent`'s `Intent putExtra(String name, String value)` method is called to store the message as a key/value pair. As with `Intent`'s other `putExtra()` methods, this method returns a reference to the `Intent` object so that method calls can be chained together.

Content Providers In Depth

A *content provider* is a component that makes a specific set of an app's data available to other apps. The data can be stored in the Android filesystem, in an SQLite database, or in any other manner that makes sense.

Content providers are preferable to directly accessing raw data because they decouple component code from raw data formats. This decoupling prevents code breakage when formats change.

Content providers are described by classes that subclass the abstract `android.content.ContentProvider` class and override `ContentProvider`'s abstract methods (such as `String getType(Uri uri)`). For example, the `SimpleContentProvider` class in Listing C-6 extends `ContentProvider` and overrides these methods.

Listing C-6. A Skeletal Content Provider

```
public class SimpleContentProvider extends ContentProvider
{
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs)
    {
        System.out.println("delete(Uri, String, String[]) called");
    }
}
```

```

        return 0;
    }

    @Override
    public String getType(Uri uri)
    {
        System.out.println("getType(Uri) called");
        return null;
    }

    @Override
    public Uri insert(Uri uri, ContentValues values)
    {
        System.out.println("insert(Uri, ContentValues) called");
        return null;
    }

    @Override
    public boolean onCreate()
    {
        System.out.println("onCreate() called");
        return false;
    }

    @Override
    public Cursor query(Uri uri, String[] projection, String selection,
                        String[] selectionArgs, String sortOrder)
    {
        System.out.println("query(Uri, String[], String, String[], String) called");
        return null;
    }

    @Override
    public int update(Uri uri, ContentValues values, String selection,
                     String[] selectionArgs)
    {
        System.out.println("update(Uri, ContentValues, String, String[]) called");
        return 0;
    }
}

```

Clients don't instantiate `SimpleContentProvider` and call these methods directly. Rather, they instantiate a subclass of the abstract `android.content.ContentResolver` class and call its methods (such as `Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)`).

■ **Note** A `ContentResolver` instance can talk to any content provider; it cooperates with the provider to manage any interprocess communication that's involved.

Learning Views, View Groups, and Event Listeners

An activity's user interface is based on *views* (user interface components), *view groups* (views that group together related views), and *event listeners* (objects that listen for events originating from views or view groups).

■ **Note** Android refers to views as *widgets*. Don't confuse widget in this context with the widgets that are shown on the Android home screen. Although the same term is used, user interface widgets and home screen widgets are different. User interface widgets are components; home screen widgets are miniature views of running apps.

Views are described by subclasses of the concrete `android.view.View` class and are analogous to Java Swing components. The `android.widget` package contains various `View` subclasses, such as `Button`, `EditText`, and `TextView` (the parent of `EditText`).

View groups are described by subclasses of the abstract `android.view.ViewGroup` class (which subclasses `View`) and are analogous to Java Swing containers. The `android.widget` package contains various subclasses, such as `LinearLayout`.

■ **Note** Because `ViewGroup` is a subclass of `View`, view groups are a kind of view. This arrangement lets you nest view groups within view groups to achieve screens of arbitrary complexity. Don't overdo it, however; users typically don't want to navigate screens that are overly complex.

Event listeners are described by nested interface members of `View` and `ViewGroup` (and various subclasses). For example, `View.OnClickListener` declares a `void onClick(View v)` method that's invoked when a clickable view (such as a button) is clicked.

To cement your understanding of views, view groups, and event listeners, consider the following `onCreate(Bundle)` method, which uses `Button`, `EditText`, and `LinearLayout` to create a screen where the user enters text and subsequently clicks the button to display this text via a pop-up message:

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    LinearLayout layout = new LinearLayout(this);
    final EditText et = new EditText(this);
    et.setEms(10);
    layout.addView(et);
    Button btnOK = new Button(this);
    btnOK.setText("OK");
    layout.addView(btnOK);
}
```

```

View.OnClickListener ocl;
ocl = new View.OnClickListener()
{
    @Override
    public void onClick(View v)
    {
        Toast.makeText(class.this, et.getText(),
                        Toast.LENGTH_SHORT).show();
    }
};
btnOK.setOnClickListener(ocl);
setContentView(layout);
}

```

After calling its superclass counterpart, `onCreate(Bundle)` instantiates `LinearLayout`. This container arranges its children in a single column or row. Its default behavior is to arrange the children in a row.

Keyword `this` is passed to `LinearLayout`'s constructor, a practice followed by other widget constructors. The current context object referenced by `this` lets a widget load and access resources (discussed later in this appendix) when necessary.

Next, `EditText` is instantiated and its inherited (from `TextView`) `void setEms(int ems)` method is called to set the widget's width to 10 *ems* (a relative measurement unit; one em equals the height of the capital letter "M" in the default font size).

At this point, `LinearLayout`'s inherited (from its `ViewGroup` parent) `void addView(View child)` method is called to add the `EditText` widget instance to the `LinearLayout` widget container instance.

Having finished with `EditText`, `onCreate(Bundle)` instantiates `Button`, invokes its inherited (from `TextView`) `void setText(CharSequence text)` method to set the button label to OK, and adds the `Button` instance to the `LinearLayout` instance.

`onCreate(Bundle)` now instantiates an anonymous class that implements the `View.OnClickListener` interface, overriding `onClick(View)` to display a *toast* (a message that pops up on the surface of the window for a short period of time).

■ **Caution** The code fragment demonstrates a problem where efficiency is concerned. Consider the approach to creating the click listener that's subsequently attached to the button. This approach is inefficient because it requires that a new object (an instance of an anonymous class that implements the `View.OnClickListener` interface) be created each time `onCreate(Bundle)` is called. (This method is called each time the device orientation changes.) A more efficient approach makes `ocl` an instance field and instantiates the anonymous class only when `ocl` doesn't contain the null reference. However, an even better solution exists. This solution is presented later in this appendix where resources are discussed.

The toast is created by invoking the `android.widget.Toast` class's `Toast.makeText(Context context, CharSequence text, int duration)` factory method, where the value passed to `context` is a reference to the current activity (abstracted here via `class.this`) and `duration` is one of `Toast.LENGTH_SHORT` or `Toast.LENGTH_LONG`. After the `Toast` instance has been created, `Toast`'s `void show()` method is called to display the toast for the specified period of time. (The message fades in when this method is called and fades out after the duration expires.)

Following listener creation, `onCreate(Bundle)` invokes `Button`'s inherited (from `View`) `void setOnClickListener(View.OnClickListener l)` method to register the previously created listener object with the button.

Finally, `onCreate(Bundle)` invokes `Activity`'s `void setContentView(View view)` method. This method is used to install the `LinearLayout` instance into the activity's view hierarchy so that the `edittext` and `button` widgets can be displayed in a single row.

Although you can create user interfaces by instantiating widget classes, there are advantages to using resources for this task. For example, you can separate the user interface description from the code that creates and manipulates it.

Learning Resources

Resources are images, strings, and other entities that support apps. Developers store them in external files to maintain them independently of code. Also, separating resources from the code that relies on them makes it easier to adapt an app to run on multiple devices and support multiple *locales* (geographical, political, or cultural regions).

Android supports the following resource types:

- *Animation*: A simulation of movement specified as a *property animation* (an animation in which an object's property value(s) [for example, background color or alpha value] is/are modified over a period of time) or a *view animation* (an animation in which a series of transformations [for example, rotation or fading] are performed on a single image—a *tween animation*—or an animation in which images are successively shown—a *frame animation*)
- *Color State List*: A list of colors mapped to widget states (such as a button's pressed, focused, and neither pressed nor focused states)
- *Drawable*: A graphic to be drawn on the screen and retrieved via an appropriate API (a bitmap, for example)
- *Layout*: An arrangement for a screen's widgets (such as in a linear fashion)

- *Menu*: An app menu, such as *options menu* (an activity's primary collection of menu items) or *context menu* (a floating menu)
- *String*: a text item, an array of text items, or a pluralistic text item with optional styling and formatting
- *Style*: the format and look for a user interface, ranging from a single widget (such as a button) to an activity or app
- *Additional*: Boolean value, color value, dimension value, unique identifier, integer value, integer array, typed array, and raw/asset

■ **Note** A *style* is a collection of properties that specify the look and format for a view or a window. A *theme* is a style applied to an entire app or activity rather than an individual view.

Classifying Resources

Android classifies resources as default, alternative, or platform. Resources in the first two categories are supplied by the developer and organized as files in subdirectories of the app project's `res` directory. These files must not be placed directly in `res`. Doing so will result in an error when the app is being built.

Default Resources

Default resources are used when no alternative resources exist that match the current device configuration. For example, the same layout resource is used to arrange widgets on a device with a small screen and on a device with a large screen. A second example is strings of English text that are used regardless of the device's locale setting.

Default resources are stored in the following subdirectories of the `res` directory:

- `anim` stores XML files that define tween animations.
- `animator` stores XML files that define property animations. Property animation XML files can be saved in the `anim` directory as well. However, `animator` is preferred for property animations to distinguish between both types.
- `color` stores XML files that define state lists of colors.

- `drawable` stores either bitmap files (`.png`, `.9.png`, `.jpg`, `.gif`) or XML files that are compiled into bitmap files, *nine-patches* (resizable bitmaps), state lists, shapes, frame animation drawables, or other drawables.
- `layout` stores XML files that define user interface layouts.
- `menu` stores XML files that define different kinds of app menus (such as a context menu).
- `raw` stores arbitrary files in their raw form where the original filenames no longer exist. To preserve their filenames (and file hierarchy), save these files in the `assets` directory, which is at the same level as `res`.
- `values` stores XML files that define simple values such as strings, integers, or colors. Each file in this directory can define multiple resources, whereas XML files in other directories define single resources. Because each resource is defined with its own XML element, you can name these files whatever you want and place different resource types in the same file. However, it's clearer to place unique resource types in different files and adopt the following filename conventions: `arrays.xml` for resource arrays (that is, typed arrays), `colors.xml` for color values, `dimens.xml` for dimension values, `strings.xml` for string values, and `styles.xml` for styles.
- `xml` stores arbitrary XML files, including various configuration files, such as a *searchable configuration* (an XML-based configuration file that supports search with assistance from Android, in which search queries are delivered to an activity and search suggestions are provided).

Resources stored in these directories define an app's default design and content. They're used by the current Android device unless overridden by alternative resources.

Alternative Resources

Alternative resources are resources that are used with a specific device configuration. For example, a layout resource that's optimal for landscape mode replaces the default portrait-oriented layout resource when the device is switched to landscape mode. A second example is strings of French text replacing default English-oriented strings when the device's locale setting is changed to French.

As with default resources, alternative resources are stored in specific subdirectories of the `res` directory. The name of each alternative directory begins with the name of a default resource directory and continues with a hyphen followed by a configuration qualifier name. For example, `layout-land` identifies the directory for storing landscape-oriented layout files.

The following list identifies a few of the configuration qualifier names that can be appended to default resource directory names:

- *Language and region:* The language is defined by a two-letter “ISO 639-1” (http://en.wikipedia.org/wiki/ISO_639-1) language code and is optionally followed by a two-letter “ISO 3166-1-alpha-2” (http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2) region code that’s preceded by lowercase letter *r*. These codes aren’t case sensitive. The prefix *r* is used to distinguish the region portion; you cannot specify a region without a language. Examples include *en*, *fr*, *en-rUS*, *en-rGB*, *fr-rFR*, and *fr-rCA*.
- *Platform version:* The API level supported by the device is indicated by a numeric code beginning with lowercase letter *v*. Examples include *v1* for API Level 1 (devices with Android 1.0 or higher) and *v4* for API Level 4 (devices with Android 1.6 or higher).
- *Screen orientation:* Either *port* for portrait (vertical) orientation or *land* for landscape (horizontal) orientation is specified.
- *Screen pixel density:* Specify *ldpi* for low-density screens (approximately 120 dpi [dots per inch]), *mdpi* for medium-density (on traditional HVGA) screens (approximately 160 dpi), *hdpi* for high-density screens (approximately 240 dpi), *xhdpi* for extra high-density screens (approximately 320 dpi), *nodpi* for bitmap resources that are not to be scaled to match the device’s screen pixel density, or *tvdpi* for screens somewhere between *mdpi* and *hdpi* (approximately 213 dpi). The *xhdpi* qualifier was added in API Level 8 and the *tvdpi* qualifier was added in API Level 13.
- *Screen size:* Specify *small* for screens whose sizes are similar to the low-density QVGA screen (minimum layout size is 320x426 dp [density-independent pixel] units), *normal* for screens whose sizes are similar to the medium-density HVGA screen (minimum layout size is approximately 320x476 dp units), *large* for screens whose sizes are similar to the medium-density VGA screen (minimum layout size is approximately 480x640 dp units), and *xlarge* for screens whose sizes are much larger than the traditional medium-density HVGA screen (minimum layout size is approximately 720x960 dp units). Extra-large screen devices are most likely tablet-oriented devices. Support for screen size configuration qualifiers was added in API Level 4. Support for the *xlarge* qualifier wasn’t added until API Level 9.

SUPPORTING A WIDE VARIETY OF SCREENS

Android has been designed to support screens of various orientations, sizes, and densities, as is explained in Google's "Supporting Multiple Screens" document at http://developer.android.com/guide/practices/screens_support.html. Also, Google discusses Android's various units of measurement in its document on the dimension resource at <http://developer.android.com/guide/topics/resources/more-resources.html#Dimension>. Because this appendix refers to density-independent pixels and scale-independent pixels, these two units of measurement are defined below.

A *density-independent pixel* (*dip* or *dp*) is an abstract unit (that is, a virtual pixel) that's based on the physical density of the screen. This unit is relative to a 160 dpi screen, so one dp is one pixel on a 160 dpi screen. The ratio of dp-to-pixel will change with the screen density but not necessarily in direct proportion. Use this unit when defining layout to express layout dimensions or position in a density-independent way. For example, specify `android:padding="5dip"` (or `android:padding="5dp"`) instead of `android:padding="5px"` in your XML file to state that you want five density-independent pixels (instead of five density-dependent pixels) of padding around a view.

A *scale-independent pixel* (*sip* or *sp*) is similar to a dp but is scaled by the user's font size preference. Use this unit when specifying font sizes in your resources, so they will be adjusted for both the screen density and the user's preference. For example, you would specify `android:textSize="15sp"` instead of `android:textSize="15px"` in your XML file to set the size of a `<TextView>` element based on the user's font size.

You can often append multiple qualifiers to a default resource directory name, provided that you place a hyphen between each qualifier and its predecessor. For example, `drawable-fr-port` refers to an alternative drawable resource to be used only when the device is set to the French locale and portrait orientation.

Android requires you to adhere to the following rules when using configuration qualifier names:

- When specifying multiple configuration qualifier names, they must be specified in the order shown in Table 2 of Google's "Providing Resources" (<http://developer.android.com/guide/topics/resources/providing-resources.html>) document; otherwise, the associated resources will be ignored. For example, `drawable-land-mdpi` is correct, whereas `drawable-mdpi-land` is incorrect.
- You cannot nest alternative resources. For example, you cannot specify `res/drawable/drawable-fr`. Instead, you would specify `res/drawable-fr`.

- You cannot specify multiple values for a qualifier type. For example, you cannot specify a `drawable-rCA-rFR` directory to store the same drawable files for Canada and France. Instead, you must create separate `drawable-rCA` and `drawable-rFR` directories where each directory contains the appropriate files or aliases (which are discussed later in this appendix).

You must supply your app's default and alternative resources. You can also leverage the various platform resources that Google provides for use in your apps.

Platform Resources

Google has standardized several resources (for example, styles, themes, and layouts) that it makes available for your own use. These *platform resources* are accessible via the `android` package's `R` class and its various subclasses (such as `R.anim` and `R.layout`).

Accessing Resources

After creating your app's default and alternative resources (whose file names follow the aforementioned conventions, and which are stored in appropriately named subdirectories of `res`), you will want to access them from code and/or other XML files. You might also want to access platform resources.

Code-Based Access

The Android Asset Packager Tool (`aapt`) generates a file named `R.java` under the app project's package hierarchy within the project's `gen` directory. This Java source file stores resource IDs (as names and integer values) for all resources organized under the `res` directory. It reveals that resource ID names consist of the following two parts:

- *Resource type*: Each resource belongs to a type such as `drawable`, `string`, and `layout`. The type is `id` for XML-based resources that are defined via elements whose `android:id` attributes identify them via the `@+id/resource name` syntax. For example, `<TextView android:id="@+id/msg" />` defines the XML `<TextView>` element `msg`.
- *Resource name*: Each resource has a name, which is a filename (excluding the extension), the value of an XML file's `android:name` attribute when the resource is a simple value (such as a string), or the resource name when the resource is defined according to the `@+id/resource name` syntax.

Given this information, you can access a resource from your code by typically adhering to the following syntax:

R.resource type.resource name

R identifies the class described by `R.java`, and *resource type* and *resource name* provide the resource's type and name. A period character separates each component. For example, `R.string.cancel` refers to the `cancel` resource name member of the `string` resource type in class `R`.

Various Android API methods require a resource ID argument. For example, the `android.content.res.Resources` class (whose instance is returned by invoking the `Context` class's `Resources` `getResources()` method) provides methods for returning an app's resources. These methods require specific resource IDs as arguments, as demonstrated below:

```
Resources res = getContext();
Drawable flag = res.getDrawable(R.drawable.canada);
String country = res.getString(R.string.canada);
```

The `Drawable` `getDrawable(int id)` method returns an `android.graphics.drawable.Drawable` object for the drawable resource identified by `R.drawable.canada` (probably a bitmap file stored in the `res/drawable` directory). Method `String` `getString(int id)` returns the string resource identified by `R.string.canada`, which is usually an entry in a `strings.xml` file located in the `res/values` directory.

■ **Note** Images stored in the `res\drawable` directory may be scaled to match the screen size and density. Images stored in `res\drawable-hdpi`, `res\drawable-ldpi`, `res\drawable-mdpi`, and `res\drawable-xhdpi` are never scaled. Before scaling an image in `res\drawable`, Android looks in one of these other directories (the chosen directory corresponds to the current screen size/density) for the image. If the image is found, Android uses that image without scaling.

Suppose you've created English and French `strings.xml` files with `canada` entries in `res/values` and `res/values-fr`. When the device's language is set to English, Android obtains `R.string.canada`'s value from `res/values/strings.xml`. When the language is set to French, Android obtains the value from `res/values-fr/strings.xml`. If Android can't find `canada` in `res/values-fr/strings.xml`, it defaults to `res/values/strings.xml`.

■ **Note** You can access a raw resource by invoking one of the `Resources` class's `openRawResource()` methods with a resource ID specified as `R.raw.filename`, where *filename* corresponds to the original filename. Each of these methods returns a `java.io.InputStream` object from which you can read the resource. For any resource files saved in the `assets` directory, you need to use `android.content.res.AssetManager` to access them. Files in `assets` are not given resource IDs.

XML-Based Access

You can refer to existing resources from various XML element attributes. For example, you will often refer to string and image (that is, drawable) resources to supply the text and images for various widgets that you specify in your layout files. When referring to another resource from an XML context, you typically adhere to the following syntax:

`@resource type/resource name`

@ signifies a reference to an existing resource. The forward slash-separated *resource type* and *resource name* have the same meaning as previously specified. For example, `@string/cancel` refers to the `cancel` resource name member of the `string` resource type, which is often located in a `strings.xml` file stored in the `res/values` directory.

You might want to use the same resource for multiple device configurations and you don't want to provide that resource as a default resource. Instead of storing the resource in multiple alternative resource directories, you can (in certain cases) create an alternative resource as an *alias* for the resource saved in your default resource directory.

For example, you need a unique version of your app icon (stored in `icon.png`) for different locales, but the English-Canadian and French-Canadian locales need to use the same version. Instead of copying the same image file into `res/drawable-en-rCA` and `res/drawable-fr-rCA` directories, you could do the following:

1. Store the image used for both locales in `icon_ca.png` (don't use `icon.png`) and place this file in the `res/drawable` directory.
2. Create an `icon.xml` file whose `<bitmap>` element refers to `icon_ca.png` (such as `<bitmap xmlns:android="http://schemas.android.com/apk/res/android" android:src="@drawable/icon_ca" />`) and store it in `res/drawable-en-rCA` and in `res/drawable-fr-rCA`. When `icon.xml` is saved in an alternative resource directory such as `res/drawable-en-rCA`, Android compiles it into a resource that can be referenced from code via `R.drawable.icon` and from XML via `@drawable/icon`. However, it's actually an alias for `R.drawable.icon_ca` (saved in `res/drawable`) or `@drawable/icon_ca`.

Platform-Based Access

A platform resource is accessed in code via its fully qualified package name, as in `android.R.layout.simple_list_item_1` (a layout resource for items presented via an `android.widget.ListView` instance's list). It's accessed in XML via package name `android`, as in `@android:color/white` (the XML equivalent of `android.R.color.white`).

Resources and the User Interface

You previously learned how to create an activity's user interface by instantiating widgets. However, it's often better to create the user interface by declaring it in one or more XML files, to simplify maintenance and to more easily adapt the user interface to multiple devices and locales.

The following `onCreate(Bundle)` method uses the resource approach to create a user interface involving `edittext` and `button` widgets:

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

After calling its superclass counterpart, `onCreate(Bundle)` executes `setContentView(R.layout.main)`, passing resource ID `R.layout.main` to Activity's void `setContentView(int layoutResID)` method.

`setContentView(int)` *inflates* (converts from XML to a view hierarchy) the layout resource identified by `R.layout.main` into a hierarchy of view objects that describe the activity's user interface. This resource is stored in a file named `main.xml` that's located in the `res/layout` directory for portrait orientation or the `res/layout-land` directory for landscape orientation.

`main.xml` declaratively describes the `edittext` and `button` widgets, as well as their linear layout container. The following code fragment reveals the contents of this file (without the `<?xml version="1.0" encoding="utf-8"?>` XML prolog):

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <EditText android:id="@+id/et"
        android:ems="10"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button android:id="@+id/btnOK"
        android:onClick="doClickOk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/ok"/>
</LinearLayout>
```

The `<LinearLayout>` element sandwiches `<EditText>` and `<Button>` elements, which are child elements of `<LinearLayout>`. They will be inflated along with `<LinearLayout>`.

Each of `<LinearLayout>`'s `android:layout_width` and `android:layout_height` attributes are assigned `fill_parent` so that this container will occupy the activity's entire screen.

■ **Note** The `fill_parent` attribute value means that the view wants to be as big as its parent (minus padding). Essentially, the view expands to take up as much space as is available within the container where the view has been placed. Starting, with API Level 8, `fill_parent` has been deprecated in favor of `match_parent`, which means the same thing.

The `<EditText>` element provides an `android:ems` element that corresponds to the `setEms(int)` method described earlier in the appendix. This element is assigned 10 `ems`. `<EditText>` also provides `android:layout_width` and `android:layout_height` attributes that are assigned `wrap_content` to ensure that this widget is shown at its preferred size.

■ **Note** The `wrap_content` attribute means that the view expands only as far as necessary to contain its content. This is analogous to stating that the view wants to be displayed at its *preferred* (natural) size. The preferred size is just large enough to display the view according to its preferences (such as 10 `ems` for the `edittext` widget).

The `<Button>` element offers similar `android:layout_width` and `android:layout_height` attributes that ensure this widget appears at its preferred size. Its `android:text` attribute refers to a string resource that supplies the button's label text (`<string name="ok">OK</string>`). This resource is most likely declared in a `strings.xml` file.

■ **Tip** Avoid hard-coding literal strings in your code and layout resources, and store them instead as separate resource entries in `strings.xml`. Doing so makes it easier to localize the app.

`<Button>` also provides an `onClick` attribute that identifies `doClickOk`, a `void` method with a `View` parameter. This method, which must be `public`, is invoked when the button is clicked. (You don't have to instantiate a listener class and register the instance with the button.) The following code fragment presents `void doClickOk(View view)`:

```
public void doClickOk(View view)
{
    EditText et = (EditText) findViewById(R.id.et);
    Toast.makeText(Test.this, et.getText(),
        Toast.LENGTH_SHORT).show();
}
```

`doClickOk(View)` executes `findViewById(R.id.et)`, passing `edittext` resource ID `R.id.et` to `Activity's View findViewById(int id)` method. `findViewById(int)` inflates the specified resource to an `EditText` object, which is assigned to variable `et`. (The `(EditText)` cast is required.)

■ **Note** `findViewById(int)` returns null when it cannot find the resource.

Lastly, `doClickOk(View)` displays the edittext content via a toast.

■ **Caution** The `setContentView(int)` method must be called at some point before `findViewById(int)`. If not, Android presents a message that the app has stopped. The reason for this message is that no layout resource has been installed, and therefore Android has no way to locate the XML-encoded edittext and button widgets.

Learning the Manifest

Android learns about an app's various components (and more) by examining the app's XML-structured manifest file, `AndroidManifest.xml`. For example, Listing C-7 shows how this file might declare an activity component.

Listing C-7. A Manifest File Declaring an Activity

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.project" android:versionCode="1"
    android:versionName="1.0">
    <application android:label="@string/app_name" android:icon="@drawable/icon">
        <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>
    </application>
</manifest>
```

Listing C-7 begins with the necessary `<?xml version="1.0" encoding="utf-8"?>` prolog, which identifies this file as an XML version 1.0 file, whose content is encoded according to the UTF-8 encoding standard.

Listing C-7 next presents a `<manifest>` tag, which is this XML document's root element: `android` identifies the Android namespace, `package` identifies the app's Java package, and `versionCode/versionName` identifies the version information.

Nested within `<manifest>` is `<application>`, which is the parent of app component tags. Its `android:icon` and `android:label` attributes refer to icon and label resources that Android devices display to represent the app.

■ **Note** The `android:label` attribute specifies the label shown in the list of apps when you select Manage apps from the app launcher screen's options menu (use the MENU control in the phone controls to access this menu). This attribute also provides the default label for an `<activity>` element that doesn't provide an `android:label` attribute.

Nested within `<application>` is `<activity>`, which describes an activity component. This tag's name attribute identifies a class (`MyActivity`) that implements the activity. This name begins with a period character to imply that it's relative to `com.example.project`.

■ **Note** The period isn't present when `AndroidManifest.xml` is created at the command line. However, this character is present when this file is created from within Eclipse. Regardless, `MyActivity` is relative to `<manifest>`'s package value (`com.example.project`).

Nested within `<activity>` is `<intent-filter>`. This tag declares the capabilities of the component described by the enclosing tag. For example, it declares the capabilities of the activity component via its nested `<action>` and `<category>` tags:

- `<action>` identifies the action to perform via the string assigned to its `android:name` attribute. The `"android.intent.action.MAIN"` value signifies that the activity is to be started as the initial activity with no data input to the activity and no output returned from the activity. To launch an app, Android looks for an `<activity>` element with an `<intent-filter>` element whose `<action>` element's `android:name` attribute is set to `"android.intent.action.MAIN"`.
- `<category>` provides additional information about the kind of component that should handle the intent via the string assigned to its `android:name` attribute. The `"android.intent.category.LAUNCHER"` value signifies that the activity can serve as the app's initial activity and that it will appear on the app launcher screen in sorted order by its label.

Other components are similarly declared: services via `<service>` tags, broadcast receivers via `<receiver>` tags, and content providers via `<provider>` tags. Android doesn't create components not declared in the manifest.

■ **Note** You don't need to declare in the manifest broadcast receivers that are created at runtime.

The manifest may also contain `<uses-permission>` tags to identify permissions that the app needs. For example, an app that needs to use the camera would specify the following tag: `<uses-permission android:name="android.permission.CAMERA" />`.

■ **Note** `<uses-permission>` tags are nested within `<manifest>` tags—they appear at the same level as the `<application>` tag.

At app install time, permissions requested by the app (via `<uses-permission>`) are granted to it by Android's package installer, based on checks against the digital signatures of the apps declaring those permissions and/or interaction with the user.

No checks with the user are done while an app is running. It was granted a specific permission when installed and can use that feature as desired, or the permission wasn't granted and any attempt to use the feature will fail without prompting the user.

■ **Note** `AndroidManifest.xml` provides additional information, such as naming any libraries that the app needs to be linked against (besides the default Android library), and identifying all app-enforced permissions (via `<permission>` tags) to other apps, such as controlling who can start the app's activities.

Additional Manifest Examples

Listing C-8 presents an `AndroidManifest.xml` file that identifies Listing C-1's `SimpleActivity` class and the subsequently mentioned `SimpleActivity2` class as the `SimpleActivity` app's two components—the ellipsis refers to content not relevant to this discussion.

Listing C-8. SimpleActivity's Manifest File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.project" ...>
    <application ...>
        <activity android:name=".SimpleActivity" ...>
            <intent-filter ...>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".SimpleActivity2" ...>
            <intent-filter ...>
                <action android:name="android.intent.action.VIEW" />
                <data android:mimeType="image/jpeg" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

```

        </intent-filter>
    </activity>
    ...
</application>
</manifest>

```

Listing C–8 reveals that each of `SimpleActivity` and `SimpleActivity2` is associated with an intent filter via an `<intent-filter>` tag that's nested within `<activity>`. `SimpleActivity2`'s `<intent-filter>` tag helps Android determine that this activity is to be launched when the `Intent` object's values match the following tag values:

- `<action>`'s `android:name` attribute is assigned `"android.intent.action.VIEW"`
- `<data>`'s `android:mimeType` attribute is assigned the `"image/jpeg"` MIME type—additional attributes (such as `android:path`) would typically be present to locate the data to be viewed.
- `<category>`'s `android:name` attribute is assigned `"android.intent.category.DEFAULT"` to allow the activity to be launched without explicitly specifying its component.

Given this information, the following example shows you how to start `SimpleActivity2` implicitly:

```

Intent intent = new Intent();
intent.setAction("android.intent.action.VIEW");
intent.setType("image/jpeg");
intent.addCategory("android.intent.category.DEFAULT");
SimpleActivity.this.startActivity(intent);

```

The first four lines create an `Intent` object describing an implicit intent. Values passed to `Intent`'s `Intent setAction(String action)`, `Intent setType(String type)`, and `Intent addCategory(String category)` methods specify the intent's action, MIME type, and category. They help Android identify `SimpleActivity2` as the activity to be launched.

Listing C–2 presented a `SimpleService` class. You would expand Listing C–8 with the following entry so that you could access this class from your app:

```

<service android:name=".SimpleService">
</service>

```

Listing C–5 presented a `SimpleBroadcastReceiver` class. You would expand Listing C–8 with the following entry unless you'll create the broadcast receiver at runtime:

```

<receiver android:name=".SimpleBroadcastReceiver">
</receiver>

```

Finally, Listing C–6 presented a `SimpleContentProvider` class. You would expand Listing C–8 with the following entry so that you could access this class from your app:

```

<provider android:name=".SimpleContentProvider">

```

```
</provider>
```

Learning the App Package

Android apps are written in Java. The compiled Java code for an app's components is further transformed into Dalvik's DEX format. The resulting code files along with any other required data and resources are subsequently bundled into an *App Package (APK)*, a ZIP file identified by the `.apk` suffix.

An APK isn't an app but is used to distribute the app and install it on a mobile device. It's not an app because its components may reuse another APK's components, and (in this situation) not all of the app would reside in a single APK. However, it's common to refer to an APK as representing a single app.

An APK must be signed with a certificate (which identifies the app's author) whose private key is held by its developer. The certificate doesn't need to be signed by a certificate authority. Instead, Android allows APKs to be signed with self-signed certificates, which is typical. I'll have more to say about signing an APK later in this appendix.

APK FILES, USER IDS, AND SECURITY
--

Each APK installed on an Android device is given its own unique Linux user ID, and this user ID remains unchanged for as long as the APK resides on that device. Because security enforcement occurs at the process level, the code contained in any two APKs cannot normally run in the same process, because each APK's code needs to run as a different Linux user. However, you can have the code in both APKs run in the same process by assigning the same name of a user ID to the `<manifest>` tag's `sharedUserId` attribute in each APK's `AndroidManifest.xml` file. When you make these assignments, you tell Android that the two packages are to be treated as being the same app, with the same user ID and file permissions. To retain security, only two APKs signed with the same signature (and requesting the same `sharedUserId` value in their manifests) will be given the same user ID.

Developing Android Apps

Now that you've created a development environment and understand app architecture, you're ready to develop some apps. In this section I present three apps that demonstrate various Android APIs and features: HelloWorld, Circle, and Planets. I also present additional information that's helpful when developing apps.

HelloWorld

It's traditional to begin exploring a new computer language by printing out the “hello, world” message. My first app salutes this tradition by displaying this message via a toast. It also demonstrates various logging capabilities. Check out Listing C–9 for the source code.

Listing C–9. Android Sends You Greetings

```
package ca.tutortutor.helloworld;

import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.Toast;

import java.util.logging.Level;
import java.util.logging.Logger;

public class HelloWorld extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        Toast.makeText(HelloWorld.this, "hello, world",
            Toast.LENGTH_SHORT).show();
        Log.v("onCreate() called", "logged via Log.v");
        System.out.println("onCreate() called: logged via System.out.println()");
        System.err.println("onCreate() called: logged via System.err.println()");
        Logger logger = Logger.getLogger("HelloWorld");
        logger.log(Level.INFO, "onCreate() called: logged via logger.log");
        logger.log(Level.FINE, "onCreate() called: logged via logger.log");
    }
}
```

Listing C–9 presents the source code to HelloWorld, a subclass of Activity. HelloWorld is stored in the ca.tutortutor.helloworld package—an Android app must be stored in its own package.

The onCreate() method is overridden to establish the activity’s layout/view hierarchy, display hello, world via a toast, and perform some logging tasks. It shows you four ways to log messages:

- **Log:** The `android.util.Log` class is the preferred way to log messages. This class provides static methods that log messages according to log levels such as debug, error, info, verbose, and warn. For example, `int v(String tag, String msg)` logs a message at the verbose level. The `tag` parameter is used to identify the source of a logged message. It usually identifies the class or activity where the log call occurs. The `msg` parameter is the message that you would like to have logged.
- **System.out:** You can call the various methods of the `System.out` object to log messages. These method calls are translated into `Log.i()` method calls. For example, `System.out.println("test");` is equivalent to `Log.i("System.out", "test");`.
- **System.err:** You can call the various methods of the `System.err` object to log messages. These method calls are translated into `Log.w()` method calls. For example, `System.out.println("test");` is equivalent to `Log.w("System.out", "test");`.
- **Logger:** You can also use the `java.util.logging.Logger` class and related types to perform logging.

There are two limitations when using `System.out` and `System.err` method calls to log messages:

- You cannot log at the verbose, error, or debug levels.
- You cannot define your own tag for identifying the source of the log message. Instead, `System.out` or `System.err` is logged as the source.

■ **Note** It's been reported that Android's default logging handler ignores any log messages with levels finer than `Level.INFO` when using the Java Logging API. For more information, check out the [stackoverflow.com](http://stackoverflow.com/questions/4561345/how-to-configure-java-util-logging-on-android) "How to configure `java.util.logging` on Android?" topic (<http://stackoverflow.com/questions/4561345/how-to-configure-java-util-logging-on-android>).

Building, Installing, and Running HelloWorld from the Command Line

The first step in creating HelloWorld from the command line is to use the `android` tool to create a project. When used in this way, `android` requires you to adhere to the following syntax, which is spread across multiple lines for readability:

```
android create project --target target_ID
```



```
--name your_project_name
--path /path/to/your/project/project_name
--activity your_main_activity_name
--package your_package_namespace
```

Except for `--name` (or `-n`), which specifies the project's name (if provided, this name will be used for the resulting `.apk` filename when you build your app), all of the following options are required:

- The `--target` (or `-t`) option specifies the app's build target. The *target_ID* value is an integer value that identifies the Android platform. You can obtain this value by invoking `android list targets`. If you've installed only the Android 4.4.2 platform, this command should output a single Android 4.4.2 platform target identified as integer ID 1.
- The `--path` (or `-p`) option specifies the project directory's location. The directory is created when it doesn't exist.
- The `--activity` (or `-a`) option specifies the name for the main (default) activity class. The resulting classfile is created inside */path/to/your/project/project_name/src/your_package_namespace/* and is used as the `.apk` filename when `--name` (or `-n`) isn't specified.
- The `--package` (or `-k`) option specifies the project's package namespace, which must follow the rules for packages that are specified in the Java language.

Assuming a Windows 7 platform, and assuming a `C:\prj\android` hierarchy where the HelloWorld project is to be stored in `C:\prj\android\HelloWorld`, the following command creates HelloWorld:

```
android create project -t 1 -p C:\prj\android\HelloWorld -a HelloWorld -k ca.tutortutor.helloworld
```

This command outputs messages, creates various directories, and adds files to some of these directories. It creates the following file and directory structure in `C:\prj\android\HelloWorld`:

- `AndroidManifest.xml` is the manifest file for the app being built. This file is synchronized to the Activity subclass previously specified via the `--activity` or `-a` option.
- `ant.properties` is a customizable properties file for the Apache Ant build system. You can edit this file to override Ant's default build settings, and you can provide a pointer to your keystore (discussed later) and key alias so that the build tools can sign your app when it's built in release mode (discussed later).
- `bin` is the output directory for the Apache Ant build script.
- `build.xml` is the Apache Ant build script for this project.

- `libs` contains private libraries (when required).
- `local.properties` is a generated file that contains the Android SDK home directory location.
- `proguard-project.txt` contains information on enabling *ProGuard*, an SDK tool that lets developers obfuscate their code (making it very difficult to reverse engineer the code) as an integrated part of a release build.
- `project.properties` is a generated file that identifies the project's target Android platform.
- `res` contains project resources.
- `src` contains the project's source code.

For this app, no changes need to be made to `AndroidManifest.xml`. Also, you don't have to be aware of the default resources that are organized under the `res` directory (although I'll shortly reveal the default layout file). Instead, you'll need to update the project's source code.

The `C:\prj\android\HelloWorld\src\ca\tutortutor\helloworld` directory contains a generated `HelloWorld.java` source file with the contents shown in Listing C-10.

Listing C-10. A Generated Activity

```
package ca.tutortutor.helloworld;

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

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

Listing C-10 describes an activity that doesn't do anything more than display the content of the layout file identified by resource ID `R.layout.main`. This resource ID refers to the `main.xml` file that's generated during project creation, and which is located in the `res\layout` directory. Listing C-11 describes this resource file.

Listing C–11. The Generated Layout Resource

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

Listing C–11’s XML describes a linear layout container that organizes its contained widgets in a vertical column and occupies the entire screen (less room for the status bar). The solitary text view widget displays `Hello World, HelloWorld` on the screen.

Replace the default skeletal `HelloWorld.java` content with Listing C–9’s content.

Assuming that `C:\prj\android\HelloWorld` is current, build this app with the help of Apache’s `ant` tool, which defaults to processing this directory’s `build.xml` file. At the command line, specify `ant` followed by `debug` or `release` to indicate the build mode:

- *Debug mode:* Build the app for testing and debugging. The resulting APK is signed with a debug key and optimized with the `zipalign` tool. Specify `ant debug`.
- *Release mode:* Build the app for release. You must sign the resulting APK with your private key, and then optimize the APK with `zipalign`. Specify `ant release`.

Build `HelloWorld` in debug mode. The `ant debug` command creates a `gen` directory containing the ant-generated `R.java` file (in a `ca\tutortutor\helloworld` directory hierarchy), and it stores the created `HelloWorld-debug.apk` file in the `bin` directory.

Assuming that the emulated device identified by `MyAVD` is running, execute the following command to install `HelloWorld-debug.apk` on this device:

```
adb install C:\prj\android\HelloWorld\bin\HelloWorld-debug.apk
```

If `C:\prj\android\HelloWorld\bin` is the current directory, you can specify the following shorter command.

```
adb install HelloWorld-debug.apk
```

After a few moments, you should observe messages similar to the following:

```
* daemon not running. Starting it now on port 5037 *
* daemon started successfully *
74 KB/s (38009 bytes in 0.497s)
pkg: /data/local/tmp/HelloWorld-debug.apk
```

Success

If you observe a failure message instead, the probable cause is that the app is already installed.

The first two messages signify that the ADB daemon isn't running and that it has been started. These messages don't reappear after the ADB daemon is running when you reinstall the app.

From the home screen, click the app launcher icon (the rectangular grid icon centered at the bottom of the home screen). Figure C-16 shows you the HelloWorld app entry.



Figure C-16. The HelloWorld app entry presents a robot icon, which is the default icon

Click the HelloWorld icon and you should see the screen shown in Figure C-17.

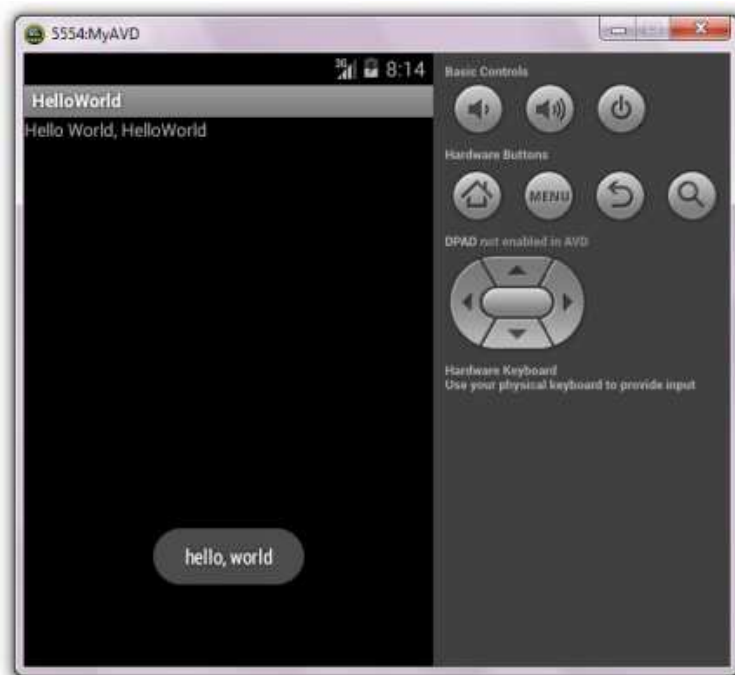
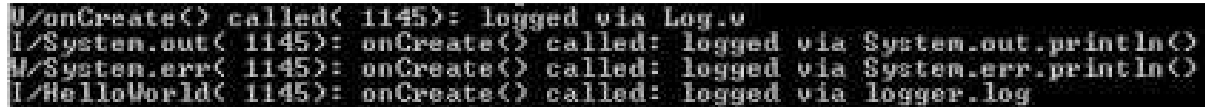


Figure C-17. The toast appears near the bottom center of the app screen

Press the Escape key to return to the home screen that displays the grid of apps. At the command line, execute the following command to present Android's message log:

```
adb logcat
```

Click the HelloWorld icon a second time. You should now observe the messages that are sent to the log—see Figure C-18.



```
V/onCreate() called( 1145): logged via Log.v
I/System.out( 1145): onCreate() called: logged via System.out.println()
W/System.err( 1145): onCreate() called: logged via System.err.println()
I/HelloWorld( 1145): onCreate() called: logged via logger.log
```

Figure C-18. These messages were sent to the log from HelloWorld's solitary activity in response to `onCreate()`

■ **Tip** You'll find it helpful to view the message log when debugging a failing app. For example, if you notice a dialog box displaying "Unfortunately, *app* has stopped" after attempting to launch an *app*, the cause is a thrown exception. You can identify this exception from the message log.

You can uninstall HelloWorld by executing the following command:

```
adb uninstall ca.tutortutor.helloworld
```

Building, Installing, and Running HelloWorld from Eclipse

I previously showed how to build, install, and run HelloWorld in a command-line context. For more significant apps, you'll probably use the Eclipse IDE. Assuming that you've previously configured Eclipse to support Android via the ADT Plugin, complete the follows steps to build this app:

1. Start Eclipse if not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.
3. On the resulting New Project dialog box, expand the Android node in the wizard tree (if necessary), select the Android Application Project branch below this node (if necessary), and click the Next button.
4. On the resulting New Android Application pane, enter **HelloWorld** into the Application Name text field (this entered name also appears in the Project Name text field, and it identifies the directory in which the HelloWorld project is stored), and enter **ca.tutortutor.helloworld** into the Package Name text field. Click the Next button.
5. On the resulting pane, uncheck the "Create custom launcher icon" check box, ensure that the "Create Activity" check box is checked, and click the Next button.

6. On the resulting Create Activity pane, click the Next button.
7. On the resulting Blank Activity pane, enter **HelloWorld** into the Activity Name text field, and enter **main** into the Layout Name text field. Click the Finish button.

Eclipse responds by creating a HelloWorld directory with the following subdirectories and files in your Eclipse workspace directory:

- **.settings**: This directory contains an `org.eclipse.jdt.core.prefs` file that records project-specific settings.
- **assets**: This directory is used to store an unstructured hierarchy of files. Anything stored in this directory can be subsequently retrieved by an app via a raw byte stream.
- **bin**: Your APK file is stored here.
- **gen**: The generated `R.java` file is stored in a subdirectory structure that reflects the package hierarchy (such as `ca\tutortutor\helloworld`).
- **libs**: This directory is created to store the `android-support-v4.jar` file that was installed earlier when I installed the Android 4.4.2 platform. This directory wouldn't have been created if I had unchecked this item in the Extras section of the SDK Manager's main window.
- **res**: App resources are stored in various subdirectories.
- **src**: App source code is stored according to a package hierarchy.
- **.classpath**: This file stores the project's classpath information so that any external libraries on which the project depends can be located.
- **.project**: This file contains important project information, such as the name of the project and the build specification.
- **AndroidManifest.xml**: This file contains HelloWorld's manifest.
- **proguard-project.txt**: This file contains information on enabling ProGuard.
- **project.properties**: This file identifies the project's target Android platform.

Eclipse presents the user interface that's shown in Figure C–19.

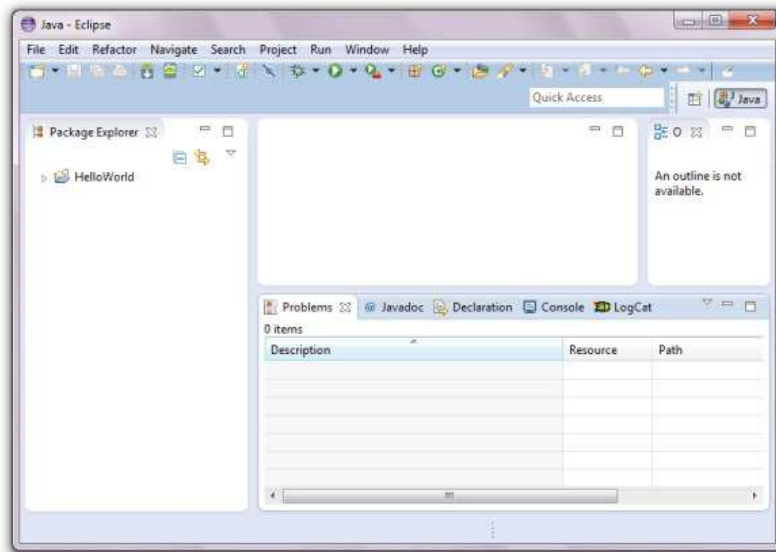


Figure C–19. Eclipse adds a HelloWorld node to its Package Explorer pane

To learn how Eclipse organizes the HelloWorld project, click the triangle icon to the left of the HelloWorld node. Figure C–20 reveals an expanded project hierarchy.

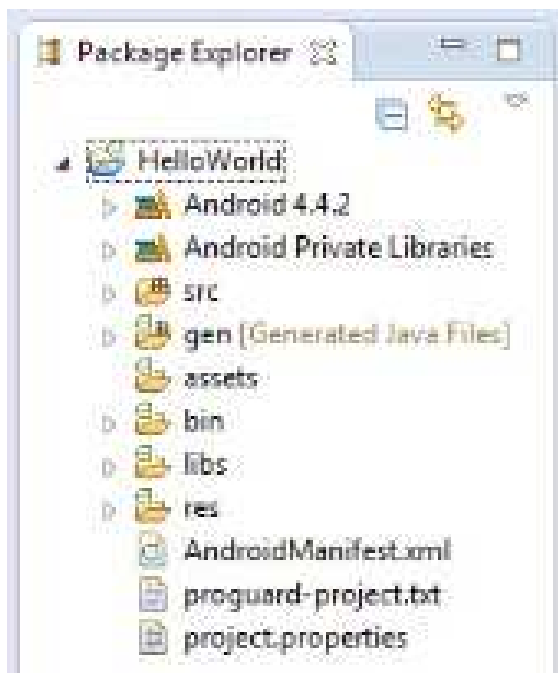


Figure C–20. The hierarchy reveals the important src and res directories along with AndroidManifest.xml

Expand the `src` node and you should observe a `ca.tutortutor.helloworld` node. Expand this node and you should observe a `HelloWorld.java` node. This node references a `HelloWorld.java` source file with skeletal contents. You'll need to replace these contents with Listing C-9. Accomplish this task by completing the following steps:

1. Copy the contents of Listing C-9 to the clipboard.
2. Right-click the `HelloWorld.java` node and select Paste from the pop-up menu.

Now that the file structure has been specified, select Run from the menu bar, and select Android Application from the resulting Run As dialog box. Click the OK button. After a few moments, the emulated MyAVD device should start running and launch HelloWorld—you might have to unlock the device before you can observe the app screen. Figure C-21 shows the resulting app screen.

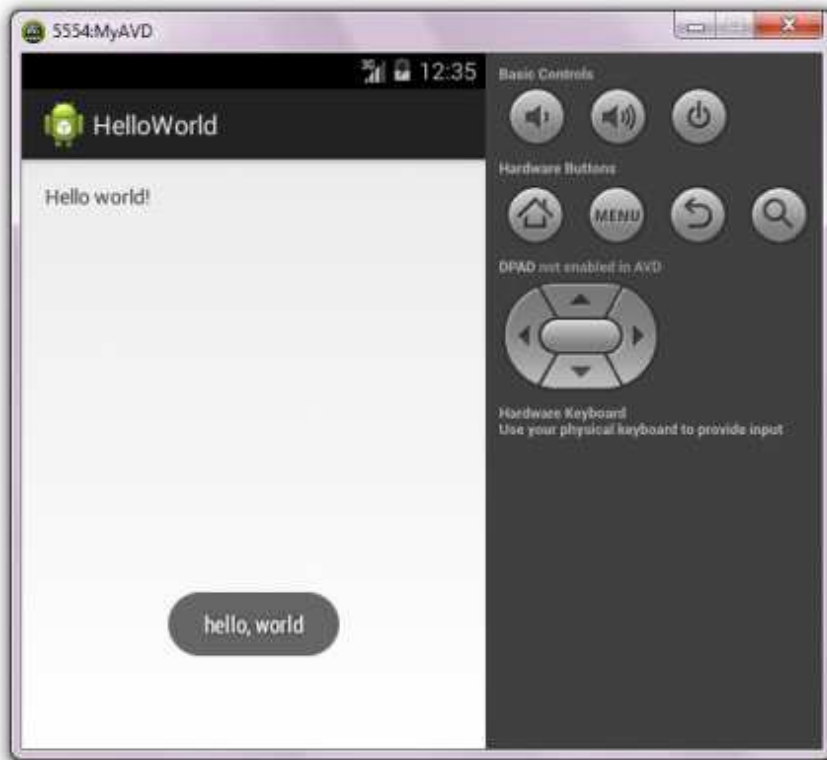


Figure C-21. The app screen differs from that shown in Figure C-17 because Eclipse creates a different style from this project

Additionally, if you select the LogCat tab, you should observe the logged messages shown in Figure C-22.



Figure C-22. Viewing all logged messages except for the message whose log level was set to Level.FINE

Circle

I previously introduced the `EditText` and `TextView` classes while discussing views, view groups, and event listeners. My second app uses these classes along with an event listener to input a circle's radius and dynamically calculate and output its area and circumference. Check out Listing C-12.

Listing C-12. *Playing with Circles*

```
package ca.tutortutor.circle;

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

import android.text.Editable;
import android.text.TextWatcher;

import android.widget.EditText;
import android.widget.TextView;

public class Circle extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        final EditText etRadius = (EditText) findViewById(R.id.radius);
        final TextView tvArea = (TextView) findViewById(R.id.area);
        final TextView tvCircumference =
            (TextView) findViewById(R.id.circumference);
        final String area = getString(R.string.area);
        final String area_undefined = getString(R.string.area_undefined);
```

```

final String circumference = getString(R.string.circumference);
final String circumference_undefined =
    getString(R.string.circumference_undefined);
TextWatcher tw;
tw = new TextWatcher()
{
    @Override
    public void afterTextChanged(Editable s)
    {
    }

    @Override
    public void beforeTextChanged(CharSequence s, int start,
                                   int count, int after)
    {
    }

    @Override
    public void onTextChanged(CharSequence s, int start,
                               int before, int count)
    {
        String text = etRadius.getText().toString();
        if (text.equals(""))
        {
            tvArea.setText(area_undefined);
            tvCircumference.setText(circumference_undefined);
            return;
        }
        try
        {
            double radius = Double.parseDouble(text);
            tvArea.setText(area + " " + Math.PI * radius * radius);
            tvCircumference.setText(circumference + " " + Math.PI *
                                    radius * 2);
        }
        catch (NumberFormatException nfe)
        {
            System.err.println("bad input detected: " +
                                nfe.getMessage());
        }
    }
};
etRadius.addTextChangedListener(tw);
}

```

Listing C-12 presents the source code to `Circle`, a subclass of `Activity`. `Circle` is stored in the `ca.tutortutor.circle` package.

The `onCreate()` method creates the user interface. After obtaining and installing the layout (stored in `main.xml`), this method uses `findViewById()` to inflate the `R.id.radius` `EditText` view, and the `R.id.area` and `R.id.circumference` `TextView` views that are described by `main.xml`.

`onCreate()` continues by invoking `getString()` to return the strings associated with the `R.string.area`, `R.string.area_undefined`, `R.string.circumference`, and `R.string.circumference_undefined` resource IDs. These resources are stored in a `strings.xml` file.

■ **Tip** It's a good idea to avoid hard-coding strings in your source code. That way, you can more easily localize your app to support other locales.

`onCreate()`'s final tasks are to instantiate `android.text.TextWatcher`, which provides methods that are called when a `TextView` or `EditText` view's content changes, and to register this text watcher with the `EditText` view by invoking `EditText`'s inherited (from its `TextView` superclass) `void addTextChangedListener(TextWatcher watcher)` method.

The only `TextWatcher` method of interest is `void onTextChanged(CharSequence s, int start, int before, int count)`. This method is invoked when the text in the associated `TextView` or `EditText` view has been changed. The parameters are of no interest in this app. Instead, the method call alone is sufficient for updating the user interface.

`onTextChanged()` first retrieves the text from the radius view. It accomplishes this task by invoking `EditText`'s inherited (from its `TextView` superclass) `CharSequence getText()` method. Because this method returns a `CharSequence` and not a `String`, the `toString()` method is invoked on the returned object.

Next, the text is compared with `""`. If nothing has been entered, `setText()` is invoked on each of the area and circumference `TextView` views to display undefined messages. Otherwise, the text is parsed into a double precision floating-point value, and the area and circumference text views are updated with the results of calculations that determine the area and circumference.

Although unlikely, it's possible for `java.lang.NumberFormatException` to be thrown—I've done so by changing the screen orientation and pressing another key combination—and for the app to crash. To avoid this possibility, I've wrapped the `Double.parseDouble()` call in a try statement and log an error message from the associated catch block when this exception is thrown.

Listing C-13 presents the contents of `main.xml`.

Listing C-13. Laying Out Circle's User Interface

```
<?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:gravity="center_vertical"
    android:background="#ffffff"
    android:padding="5dip">
```

```
<LinearLayout android:layout_width="fill_parent"
    android:layout_height="wrap_content">
    <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginRight="10dip"
        android:text="@string/radius"
        android:textColor="#000000"
        android:textSize="15sp"
        android:textStyle="bold"/>
    <EditText android:id="@+id/radius"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:hint="@string/type_a_number"
        android:inputType="numberDecimal|numberSigned"
        android:maxLines="1"/>
</LinearLayout>

<TextView android:layout_height="wrap_content"
    android:layout_width="wrap_content"/>

<TextView android:id="@+id/area"
    android:layout_gravity="center"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/area_undefined"
    android:textColor="#000000"
    android:textSize="15sp"
    android:textStyle="bold"/>

<TextView android:layout_height="wrap_content"
    android:layout_width="wrap_content"/>

<TextView android:id="@+id/circumference"
    android:layout_gravity="center"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/circumference_undefined"
    android:textColor="#000000"
    android:textSize="15sp"
    android:textStyle="bold"/>
</LinearLayout>
```

Listing C–13 describes a layout that consists of a linear layout that occupies the screen and lays out its contained views in a vertical direction. Furthermore, it vertically centers these views (via `android:gravity="center_vertical"`), sets the background color to white (via `android:background="#ffffff"`), and places 5 device-independent pixels of padding around the vertically-centered views (via `android:padding="5dip"`).

The first contained view is a linear layout that lays out text and edit text views in a horizontal direction. The height of the linear layout is set to `wrap_content` so that this view doesn't fill the whole screen—there are two more views to layout vertically.

The contained text and edit text views offer some interesting attributes:

- `android:layout_marginRight="10dip"`: This attribute leaves a 10-dip gap between the text view on the left and the edit text view on the right. This gap makes the screen look nicer.
- `android:text="@string/radius"`: This attribute refers to a string resource named `radius`. The string identified by `radius` will be extracted from `strings.xml` and assigned to the text attribute, and will ultimately be displayed on the screen. As with source code, you should avoid hard-coding text in a resource file and simply reference that text from a `strings.xml` file (discussed shortly).
- `android:textColor="#000000"`: This attribute assigns color black to the `textColor` attribute. Because the background is white, I want to ensure maximum contrast—the default text color might not be black under every possible theme. (I could store this attribute value in an external resource file and reference it in a manner that's similar to referencing a string. However, I chose to not do so in this case for convenience.)
- `android:textSize="15sp"`: This attribute assigns 15 scale-independent pixels to the `textSize` attribute. This is the size at which the text is to be displayed.
- `android:textStyle="bold"`: This attribute assigns the bold style to the `textStyle` attribute.
- `android:hint="@string/type_a_number"`: This attribute assigns a hint message to be displayed in the edit text view when nothing has been entered. It gives the user some feedback on what to do next.
- `android:inputType="numberDecimal|numberSigned"`: This attribute identifies the kind of value that can be entered into an edit text view. Here, the value must be a signed decimal number.
- `android:maxLines="1"`: This attribute specifies that exactly one line of text can be entered into an edit text view—an edit text view can display multiple lines.

You'll notice a pair of text views that define only `layout_width` and `layout_height` attributes. These text views exist as spacers to provide blank vertical space between vertically adjacent views. The resulting layout looks nicer.

One final item of interest is `android:layout_gravity="center"`. This assignment horizontally centers the text views that display the current area and circumference.

Listing C–14 presents the contents of `strings.xml`.

Listing C–14. Defining the Circle App's Text

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Circle</string>
  <string name="area">Area:</string>
  <string name="area_undefined">Area: undefined</string>
  <string name="circumference">Circumference:</string>
  <string name="circumference_undefined">Circumference: undefined</string>
  <string name="radius">Radius</string>
  <string name="type_a_number">type a number</string>
</resources>
```

Listing C–14 describes an XML file that defines all of the text strings used by `Circle.java` and the layout resource. This file stores individual string resources as `<string>` elements between a `<resources>` open tag and its associated `</resources>` close tag.

Each `<string>` tag defines a name attribute that names the resource. This name is referenced from source code (such as `R.string.area`) or a resource file (such as `@string/circumference_undefined`). The text between `<string>` and `</string>` is extracted by Android, typically via `getString()`.

Building, Installing, and Running Circle from the Command Line

The first step in creating Circle from the command line is to use the `android` tool to create a project. Specify the following command to accomplish this task:

```
android create project -t 1 -p C:\prj\android\Circle -a Circle -k ca.tutortutor.circle
```

This command outputs messages, creates various directories, and adds files to some of these directories.

Perform the following tasks to create the file structure for this app:

- Copy Listing C–12 to a `Circle.java` file and store this file in the `Circle\src\ca\tutortutor\circle` directory.
- Copy Listing C–13 to a `main.xml` file and store this file in the `Circle\res\layout` directory.
- Copy Listing C–14 to a `strings.xml` file and store this file in the `Circle\res\values` directory.

With `Circle` as the current directory, execute the following command to build this app:

```
ant debug
```

Assuming a successful compilation, and assuming that the MyAVD device is running, switch to the bin directory and execute the following command to install Circle-debug.apk on the device:

```
adb install Circle-debug.apk
```

Assuming a successful install, you should observe a Circle entry with a robot icon on the app screen. Click this icon and you should observe a screen similar to that shown in Figure C-23.

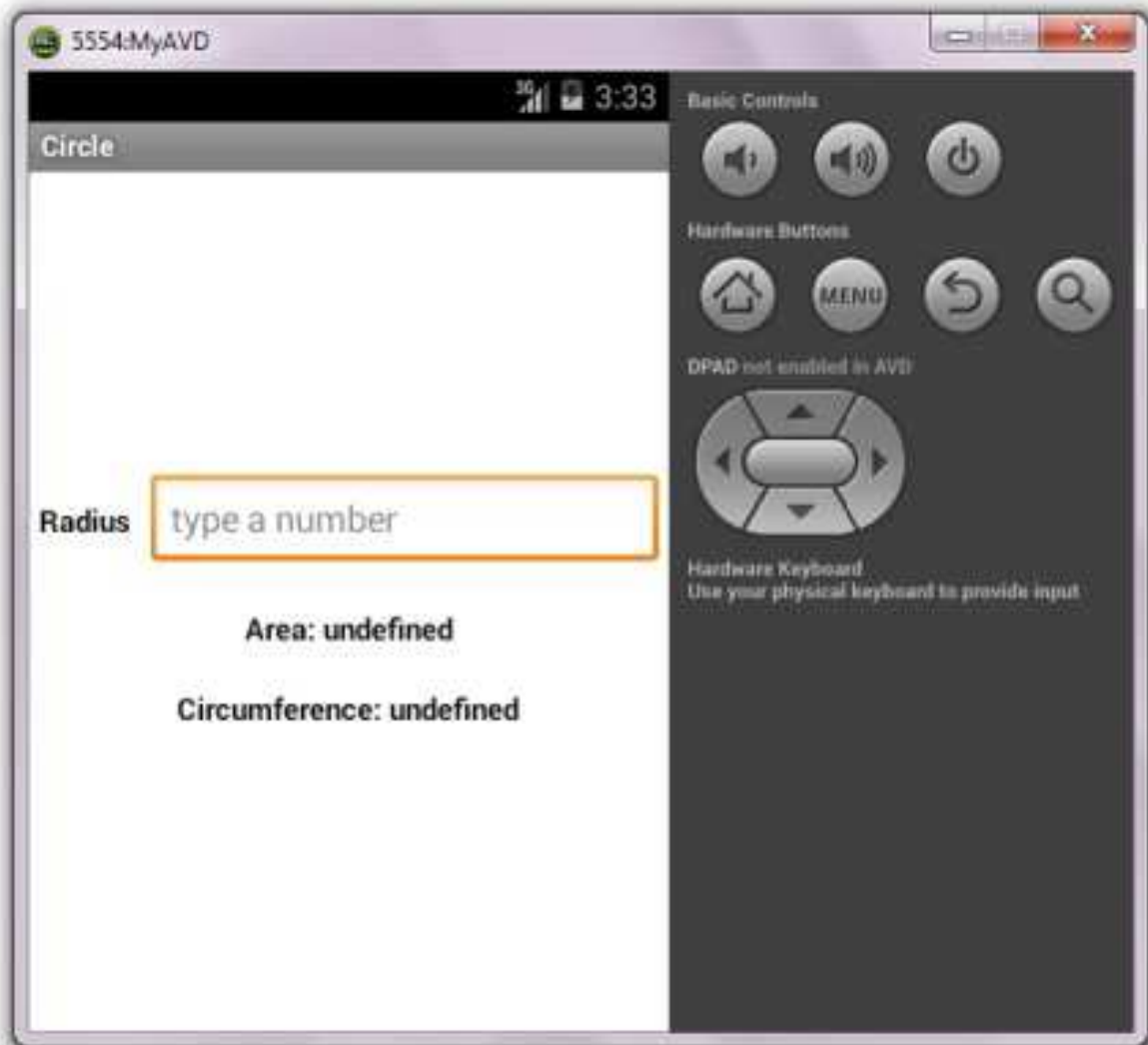


Figure C-23. You're requested to enter a number

As you enter a number, the value associated with Area and the value associated with Circumference will dynamically change, as shown in Figure C–24.



Figure C–24. The area and circumference values dynamically change as you enter a number

Building, Installing, and Running Circle from Eclipse

Now that you’ve built, installed, and run Circle in a command-line context, let’s develop and run this app in an Eclipse context. Complete the following steps to build Circle:

1. Start Eclipse if not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.
3. On the resulting New Project dialog box, expand the Android node in the wizard tree (if necessary), select the Android Application Project branch below this node (if necessary), and click the Next button.
4. On the resulting New Android Application pane, enter **Circle** into the Application Name text field (this entered name also appears in the Project Name text field, and it identifies the directory in which the Circle project is stored), and enter **ca.tutortutor.circle** into the Package Name text field. Click the Next button.
5. On the resulting pane, uncheck the “Create custom launcher icon” and “Create activity” check boxes, and click the Finish button.

Eclipse responds by creating a Circle directory with the same 12 subdirectories and files that I previously discussed in the context of the HelloWorld project.

This time, the src node will be empty. Right-click this node and select New followed by Package from the resulting pop-up menus. On the resulting New Java Package dialog box, enter **ca.tutortutor.circle** into the Name text field, and click Finish. A ca.tutortutor.circle node now appears under src.

You will need to add a `Circle.java` node to `ca.tutortutor.circle`. You can accomplish this task by completing the following steps:

1. Copy the contents of Listing C–12 to the clipboard.
2. Right-click the `ca.tutortutor.circle` node and select Paste from the pop-up menu.

You next need to add a `main.xml` node to `res/layout`. Accomplish this task by completing these steps:

1. Copy the contents of Listing C–13 to the clipboard.
2. Right-click the `res/layout` node and select Paste from the pop-up menu.

You now need to replace the default `strings.xml` file with Listing C–14. Accomplish this task by completing the following two steps:

1. Copy the contents of Listing C–14 to the clipboard.
2. Right-click the `res/values/strings.xml` node and select Paste from the pop-up menu.

Finally, you need to replace the default `AndroidManifest.xml` node content with the `AndroidManifest.xml` file content that was generated when you built this app at the command line:

1. Copy the previous `AndroidManifest.xml` file contents to the clipboard.
2. Right-click the `AndroidManifest.xml` node and select Paste from the pop-up menu.

Now that the file structure has been specified, select Run from the menu bar, and, if displayed, select Android Application from the resulting Run As dialog box and click the OK button. After a few moments, the emulated MyAVD device should start running and launch Circle. You should see the same screen that appears in Figure C–23.

Planets

The previous apps were pretty basic. In contrast, my final Planets app is more advanced. This app lets you view an image of and some statistics for each of the 8 planets in the solar system. It demonstrates the following additional features:

- launching an activity from another activity
- passing data to an activity via an intent
- using a table layout

- localizing an app's string resources
- creating a custom icon

Listing C–15 presents the source code to the app's main Planets activity.

Listing C–15. Implementing the Planets Activity

```
package ca.tutortutor.planets;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

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

    public void doBtnClicked(View view)
    {
        Intent intent = new Intent(Planets.this, PlanetInfo.class);
        String tag = (String) view.getTag();
        if (tag.equals("mercury"))
        {
            intent.putExtra("layout", R.layout.mercury);
            intent.putExtra("name", R.string.mercury);
        }
        else
            if (tag.equals("venus"))
            {
                intent.putExtra("layout", R.layout.venus);
                intent.putExtra("name", R.string.venus);
            }
            else
                if (tag.equals("earth"))
                {
                    intent.putExtra("layout", R.layout.earth);
                    intent.putExtra("name", R.string.earth);
                }
                else
                    if (tag.equals("mars"))
                    {
                        intent.putExtra("layout", R.layout.mars);
                        intent.putExtra("name", R.string.mars);
                    }
            }
    }
}
```

```

    }
    else
    if (tag.equals("jupiter"))
    {
        intent.putExtra("layout", R.layout.jupiter);
        intent.putExtra("name", R.string.jupiter);
    }
    else
    if (tag.equals("saturn"))
    {
        intent.putExtra("layout", R.layout.saturn);
        intent.putExtra("name", R.string.saturn);
    }
    else
    if (tag.equals("uranus"))
    {
        intent.putExtra("layout", R.layout.uranus);
        intent.putExtra("name", R.string.uranus);
    }
    else
    {
        intent.putExtra("layout", R.layout.neptune);
        intent.putExtra("name", R.string.neptune);
    }
    startActivity(intent);
}
}

```

Listing C–15 presents a `Planets` class that extends `Activity` and overrides `onCreate()`. This method doesn't accomplish much beyond executing `setContentView(R.layout.main)`; to inflate the layout into a user interface screen that's presented to the user.

`Planets` also declares a `void doBtnClicked(View view)` method that's called in response to one of this activity's 8 buttons being clicked—these buttons are described in the `main.xml` layout file that I'll present later.

In response to a button being clicked, `doBtnClicked()` launches an activity with planet-specific information. Before doing so, it creates an `Intent` object that it will pass to the activity. This activity is described by the `Intent()` constructor's second argument, which is a `java.lang.Class` object specified as `PlanetInfo.class`—the `PlanetInfo` activity is to be launched.

`doBtnClicked()` next extracts the tag associated with the passed `view` argument, which identifies the view that was clicked (a button, in this example). The tag can be an arbitrary object but is specified as a string in `main.xml`. `doBtnClicked()` compares this tag to each of the planet names and, when there's a match, invokes the appropriate pair of `putExtra()` methods to configure the intent with planet-specific resource information. The configuration consists of the layout resource ID for the layout file associated with the planet along with the string resource ID for the planet's name.

`doBtnClick()`'s final task is to start the `PlanetInfo` activity, passing the `Intent` object to this activity.

Listing C–16 presents the source code to the PlanetInfo activity.

Listing C–16. Implementing the PlanetInfo Activity

```
package ca.tutortutor.planets;

import android.app.Activity;

import android.content.res.Resources;

import android.os.Bundle;

public class PlanetInfo extends Activity
{
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        Bundle extras = getIntent().getExtras();
        int layout = extras.getInt("layout");
        setContentView(layout);
        Resources res = getResources();
        setTitle(res.getString(R.string.app_name) + ": " +
            res.getString(extras.getInt("name")));
    }
}
```

Listing C–16 presents a PlanetInfo class that extends Activity and overrides onCreate(), which is called when this activity is created.

Expression Bundle extras = getIntent().getExtras() first calls Activity's Intent getIntent() method to return the Intent object that was used to start the activity, and then calls Intent's Bundle getExtras() method to return a Bundle instance with the layout- and name-associated resource IDs.

The layout ID is obtained from the intent by calling extras.getInt("layout"). The resulting ID is passed to setContentView() to install the activity's user interface. Then, Activity's inherited Resources getResources() method is called to access the app's string (and other) resources.

The Resources class's String getString(int id) method returns the value of the string resource whose identifier is passed to id. This method is called twice to return the app's name (Planets) and (with the help of the Bundle class's int getInt(String key) method) the planet name. These values are combined into a string that's passed to Activity's void setTitle(CharSequence title) method, to set the activity's title.

Listing C–17 presents the source code to the main.xml layout file.

Listing C–17. Laying Out the Planets Activity

```
<?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">
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="mercury"
        android:text="@string/mercury"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="venus"
        android:text="@string/venus"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="earth"
        android:text="@string/earth"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="mars"
        android:text="@string/mars"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="jupiter"
        android:text="@string/jupiter"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="saturn"
        android:text="@string/saturn"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="uranus"
        android:text="@string/uranus"/>
<Button android:layout_height="wrap_content"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:onClick="doBtnClicked"
        android:tag="neptune"
        android:text="@string/neptune"/>
</LinearLayout>

```

Listing C–17 describes a linear layout that lays out its 8 nested button views vertically on the screen. Each button view is described by a <Button> element that presents the following attributes:

- `android:layout_height`: This attribute ensures that a single button doesn't occupy the entire screen.
- `android:layout_weight`: This attribute ensures that all buttons have the same height so that the bottom button isn't partly cut off on some screen sizes.
- `android:layout_width`: This attribute ensures that the button is as wide as the screen.
- `android:onClick`: This attribute identifies the method to invoke when the button is clicked. The method must be declared to be `public`.
- `android:tag`: This attribute provides the locale-independent name of the planet to facilitate the identification of the clicked button in the `Planets` activity.
- `android:text`: This attribute identifies the string resource that provides the button's text.

Each of the planet-specific layout files is named for the planet whose screen it lays out; for example, `mars.xml`. Listing C–18 presents this layout file's content.

Listing C–18. A Martian Layout File

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:gravity="center_horizontal|center_vertical"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:orientation="vertical"
    android:padding="5dip">
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:text="@string/mars"
        android:textColor="#ffffff"
        android:textSize="25sp"
        android:textStyle="bold"/>
    <ImageView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:src="@drawable/mars"/>
    <TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:gravity="center_horizontal"
        android:layout_height="wrap_content"
        android:layout_width="wrap_content">
        <TableRow>
            <TextView android:layout_height="wrap_content"
                android:layout_width="wrap_content"
                android:padding="3dip"
                android:text="@string/diameter"
                android:textColor="#ffffff">
```

```

        android:textStyle="bold"/>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/mars_diameter"
        android:textColor="#ffffff"/>
</TableRow>
<TableRow>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/mass"
        android:textColor="#ffffff"
        android:textStyle="bold"/>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/mars_mass"
        android:textColor="#ffffff"/>
</TableRow>
<TableRow>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/orbit"
        android:textColor="#ffffff"
        android:textStyle="bold"/>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/mars_orbit"
        android:textColor="#ffffff"/>
</TableRow>
<TableRow>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/trivia"
        android:textColor="#ffffff"
        android:textStyle="bold"/>
    <TextView android:layout_height="wrap_content"
        android:layout_width="wrap_content"
        android:padding="3dip"
        android:text="@string/mars_trivia"
        android:textColor="#ffffff"/>
</TableRow>
</TableLayout>
</LinearLayout>

```

Listing C–18 describes a linear layout that vertically lays out a text view followed by an image view, which is followed by a table layout. The `android:gravity="center_horizontal|center_vertical"` attribute ensures that these views (as a unit) are centered horizontally and vertically on the screen.

The `<ImageView>` element describes a view for presenting an image of the planet. Its `layout_height` and `layout_width` attributes are each assigned `wrap_content` so that the image doesn't occupy the entire screen. Its `src` attribute is set to `drawable/mars`, which identifies the `mars.jpg` image file stored in the `res/drawable` directory.

The `<TableLayout>` element lays out its content in a row/column grid. Each row is described by a nested `<TableRow>` element—there are 4 rows of 2 columns and each entry is a text view that presents some textual information about the planet.

I designed `Planets` so that its text could be easily localized. The default English text is stored in the `res/values/strings.xml` file whose contents appear in Listing C–19.

Listing C–19. Default English Strings

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Planets</string>
    <string name="diameter">Diameter</string>
    <string name="earth">Earth</string>
    <string name="earth_diameter">12,756.3 km</string>
    <string name="earth_mass">5.972e24 kg</string>
    <string name="earth_orbit">149,600,000 km (1.00 AU)</string>
    <string name="earth_trivia">supports life</string>
    <string name="jupiter">Jupiter</string>
    <string name="jupiter_diameter">142,984 km</string>
    <string name="jupiter_mass">1.900e27 kg</string>
    <string name="jupiter_orbit">778,330,000 km (5.20 AU)</string>
    <string name="jupiter_trivia">largest planet</string>
    <string name="mars">Mars</string>
    <string name="mars_diameter">6,794 km</string>
    <string name="mars_mass">6.4219e23 kg</string>
    <string name="mars_orbit">227,940,000 km (1.52 AU)</string>
    <string name="mars_trivia">the red planet, tallest volcanoes</string>
    <string name="mass">Mass</string>
    <string name="mercury">Mercury</string>
    <string name="mercury_diameter">4,880 km</string>
    <string name="mercury_mass">3.30e23 kg</string>
    <string name="mercury_orbit">57,910,000 km (0.38 AU)</string>
    <string name="mercury_trivia">closest to Sun</string>
    <string name="neptune">Neptune</string>
    <string name="neptune_diameter">49,532 km</string>
    <string name="neptune_mass">1.0247e26 kg</string>
    <string name="neptune_orbit">4,504,000,000 km (30.06 AU)</string>
    <string name="neptune_trivia">farthest from Sun</string>
    <string name="orbit">Orbit</string>
    <string name="saturn">Saturn</string>
    <string name="saturn_diameter">120,536 km</string>
    <string name="saturn_mass">5.68e26 kg</string>
    <string name="saturn_orbit">1,429,400,000 km (9.54 AU)</string>
    <string name="saturn_trivia">most visible rings</string>
    <string name="trivia">Trivia</string>
    <string name="uranus">Uranus</string>
```



```

<string name="uranus_diameter">51,118 km</string>
<string name="uranus_mass">8.683e25 kg</string>
<string name="uranus_orbit">2,870,990,000 km (19.218 AU)</string>
<string name="uranus_trivia">seventh from Sun</string>
<string name="venus">Venus</string>
<string name="venus_diameter">12,103.6 km</string>
<string name="venus_mass">4.869e24 kg</string>
<string name="venus_orbit">108,200,000 km (0.72 AU)</string>
<string name="venus_trivia">most circular orbit</string>
</resources>

```

Although I could localize this app by using resource bundles (see Chapter 16), this isn't necessary because Android simplifies this task by letting you create an alternative `res\values` directory with a locale qualifier. For example, I created a `res\values-fr` directory whose `strings.xml` file contains the equivalent French text. Check out Listing C–20.

Listing C–20. The French Alternative Includes Character Entity References (see Chapter 15) for Accented Characters

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="app_name">Plan&#232;tes</string>
  <string name="diameter">Diam&#232;tre</string>
  <string name="earth">Terre</string>
  <string name="earth_diameter">12 756,3 km</string>
  <string name="earth_mass">5,972e24 kg</string>
  <string name="earth_orbit">149 600 000 km (1,00 UA)</string>
  <string name="earth_trivia">prend en charge de la vie</string>
  <string name="jupiter">Jupiter</string>
  <string name="jupiter_diameter">142 984 km</string>
  <string name="jupiter_mass">1,900e27 kg</string>
  <string name="jupiter_orbit">778 330 000 km (5,20 AU)</string>
  <string name="jupiter_trivia">plus grosse plan&#232;te</string>
  <string name="mars">Mars</string>
  <string name="mars_diameter">6 794 km</string>
  <string name="mars_mass">6,4219e23 kg</string>
  <string name="mars_orbit">227 940 000 km (1,52 UA)</string>
  <string name="mars_trivia">la plan&#232;te rouge, les plus hauts volcans</string>
  <string name="mass">Masse</string>
  <string name="mercury">Mercure</string>
  <string name="mercury_diameter">4 880 km</string>
  <string name="mercury_mass">3,30e23 kg</string>
  <string name="mercury_orbit">57 910 000 km (0,38 UA)</string>
  <string name="mercury_trivia">plus proche au soleil</string>
  <string name="neptune">Neptune</string>
  <string name="neptune_diameter">49 532 km</string>
  <string name="neptune_mass">1,0247e26 kg</string>
  <string name="neptune_orbit">4 504 000 000 km (30,06 AU)</string>
  <string name="neptune_trivia">plus &#233;loign&#233;e du soleil</string>
  <string name="orbit">Orbite</string>
  <string name="saturn">Saturne</string>
  <string name="saturn_diameter">120 536 km</string>
  <string name="saturn_mass">5,68e26 kg</string>
  <string name="saturn_orbit">1 429 400 000 km (9,54 AU)</string>

```

```

<string name="saturn_trivia">anneaux plus visible</string>
<string name="trivia">Jeu-questionnaire</string>
<string name="uranus">Uranus</string>
<string name="uranus_diameter">51 118 km</string>
<string name="uranus_mass">8,683e25 kg</string>
<string name="uranus_orbit">2 870 990 000 km (19,218 AU)</string>
<string name="uranus_trivia">septième du soleil</string>
<string name="venus">Vénus</string>
<string name="venus_diameter">12 103,6 km</string>
<string name="venus_mass">4,869e24 kg</string>
<string name="venus_orbit">108 200 000 km (0,72 UA)</string>
<string name="venus_trivia">orbite plus circulaire</string>
</resources>

```

There's one more resource to consider. To make the app look professional, I've created a set of 4 icons for display on the app launcher screen—only one of these icons is displayed. Each icon corresponds to a specific screen size and is named `ic_launcher.png`. These files are stored in `res\drawable-hdpi`, `res\drawable-ldpi`, `res\drawable-mdpi`, and `res\drawable-xhdpi` directories.

Building, Installing, and Running Planets from the Command Line

The first step in creating Planets from the command line is to use the `android` tool to create a project. Specify the following command to accomplish this task:

```
android create project -t 1 -p C:\prj\android\Planets -a Planets -k ca.tutortutor.planets
```

This command outputs messages, creates various directories, and adds files to some of these directories.

Perform the following tasks to create the file structure for this app:

- Copy Listing C-15 to a `Planets.java` file and store this file in the `Planets\src\ca\tutortutor\planets` directory.
- Copy Listing C-16 to a `PlanetInfo.java` file and store this file in the `Planets\src\ca\tutortutor\planets` directory.
- Copy Listing C-17 to a `main.xml` file and store this file in the `Planets\res\layout` directory.
- Copy Listing C-18 to a `mars.xml` file and store this file in the `Planets\res\layout` directory. Repeat this process for `mercury.xml`, `venus.xml`, `earth.xml`, `jupiter.xml`, `saturn.xml`, `uranus.xml`, and `neptune.xml`, which are stored elsewhere in the code archive.

- Copy Listing C-19 to a `strings.xml` file and store this file in the `Planets\res\values` directory.
- Create a `Planets\res\values-fr` directory and copy Listing C-20 to a `strings.xml` file in this directory.
- Create a `Planets\res\drawable` directory and copy the `earth.jpg`, `jupiter.jpg`, `mars.jpg`, `mercury.jpg`, `neptune.jpg`, `saturn.jpg`, `uranus.jpg`, and `venus.jpg` image files, which are stored elsewhere in the code archive, to this directory.
- Add the following line after the closing `</activity>` tag in `AndroidManifest.xml`:
`<activity android:name="PlanetInfo" />`. Android must be aware of this second activity or an exception will be thrown when the app attempts to launch the `PlanetInfo` activity.

With `Planets` as the current directory, execute the following command to build this app:

```
ant debug
```

Assuming a successful compilation, and assuming that the MyAVD device is running, switch to the `bin` directory and execute the following command to install `Planets-debug.apk` on the device:

```
adb install Planets-debug.apk
```

Switch to app launcher screen. You'll probably need to swipe the screen to the left to view the `Planets` app entry. Figure C-25 shows you what you should see.



Figure C-25. The Planets app is highlighted and ready to be launched

Click on this icon and you should observe a screen similar to that shown in Figure C-26—I assume that the default locale isn't French.

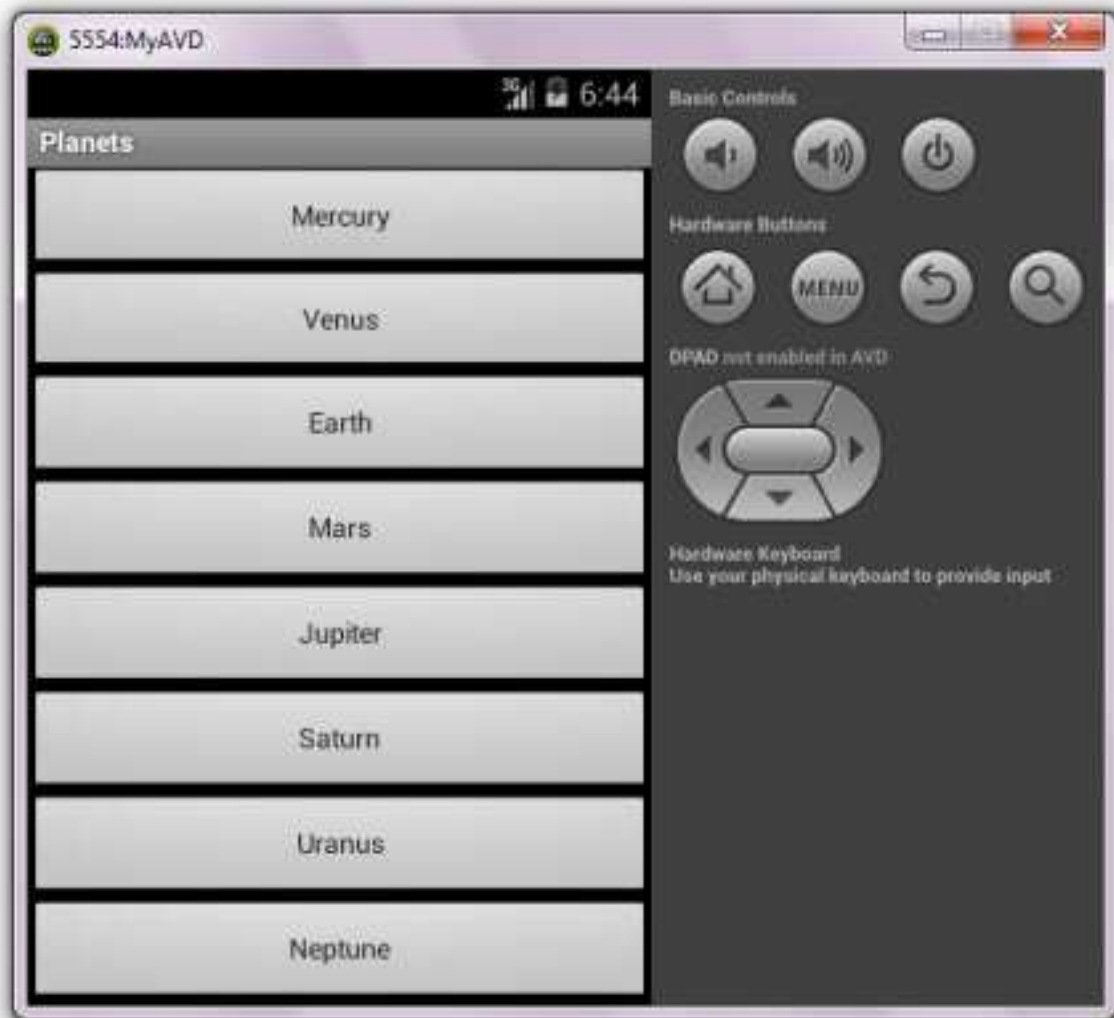


Figure C–26. Click a button to view a planet-specific screen

Suppose you click the Mars button. Figure C–27 reveals the PlanetInfo activity displaying an image of Mars and information about this planet.

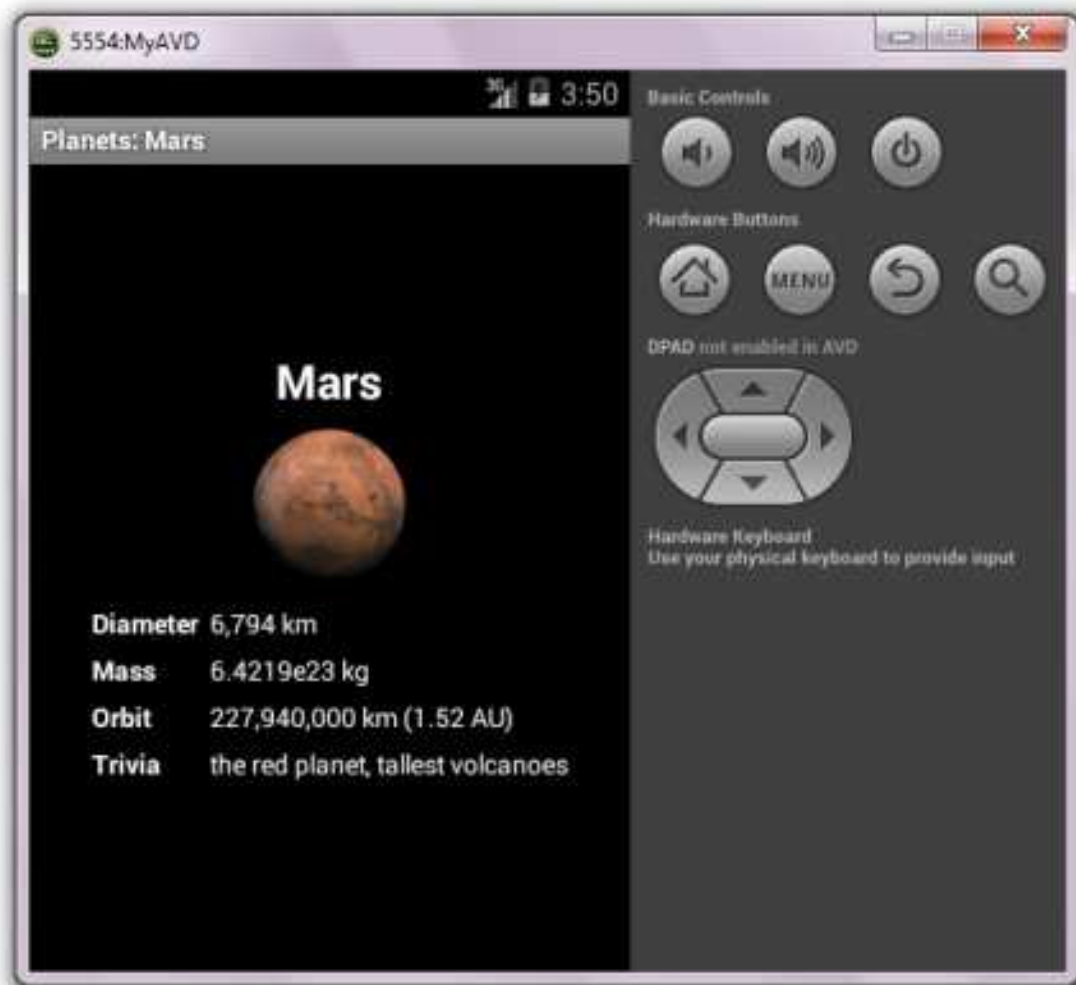


Figure C-27. Press the Escape key to return to the main activity

Return to the app launcher screen and click the Custom Locale app. Scroll down the list and select fr-French. Exit this app and you should observe Figure C-28's icon entry for Planets.



Figure C-28. The Planets app icon text has been localized

Launch this app and you should observe a few textual differences on the buttons—see Figure C–29.

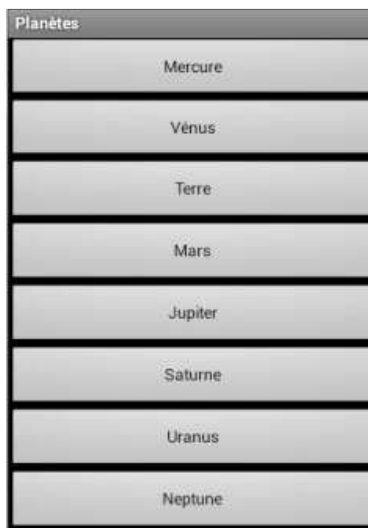


Figure C–29. Button text has been localized

Finally, click the Terre (Earth) button and you should observe Figure C–30.



Figure C–30. The Terre (Earth) screen presents its information according to the French locale

Building, Installing, and Running Planets from Eclipse

Now that you've built, installed, and run Planets in a command-line context, let's develop and run this app in an Eclipse context. Complete the following steps to build Planets:

1. Start Eclipse if not running.
2. Select New from the File menu, and select Project from the resulting pop-up menu.
3. On the resulting New Project dialog box, expand the Android node in the wizard tree (if necessary), select the Android Application Project branch below this node (if necessary), and click the Next button.
4. On the resulting New Android Application pane, enter **Planets** into the Application Name text field (this entered name also appears in the Project Name text field, and it identifies the directory in which the Planets project is stored), and enter **ca.tutortutor.planets** into the Package Name text field. Click the Next button.
5. On the resulting pane, uncheck the "Create custom launcher icon" and "Create activity" check boxes, and click the Finish button.

Eclipse responds by creating a Planets directory with the same 12 subdirectories and files that I previously discussed in the context of the HelloWorld project.

This time, the src node will be empty. Right-click this node and select New followed by Package from the resulting pop-up menus. On the resulting New Java Package dialog box, enter **ca.tutortutor.planets** into the Name text field, and click Finish. A ca.tutortutor.planets node now appears under src.

You will need to add Planets.java and PlanetInfo.java nodes to ca.tutortutor.planets. You can accomplish this task by completing the following steps:

1. Copy the contents of Listing C-15 to the clipboard.
2. Right-click the ca.tutortutor.planets node and select Paste from the pop-up menu.
3. Copy the contents of Listing C-16 to the clipboard.
4. Right-click the ca.tutortutor.planets node and select Paste from the pop-up menu.

You next need to add main.xml, mars.xml, and the other planet layout nodes to res/layout. Accomplish this task by completing these steps:

1. Copy the contents of the XML files to the clipboard. In Windows 7, select all of the files and copy the selection to the clipboard.

2. Right-click the `res/layout` node and select Paste from the pop-up menu.

You now need to replace the default `strings.xml` file with Listing C-19. Accomplish this task by completing the following two steps:

1. Copy the contents of Listing C-19 to the clipboard.
2. Right-click the `res/values/strings.xml` node and select Paste from the pop-up menu.

As an exercise, create a `res/values-fr/strings.xml` node and paste Listing C-20 to this node.

Create a drawable node under `res` by right-clicking `res` and selecting New followed by Folder from the pop-up menus. Then select all 8 planet JPEG files, copy the selection to the clipboard, and paste them to drawable.

Your next step is to replace the `ic_launcher.png` files in the `drawable-hdpi`, `drawable-mdpi`, and `drawable-xhdpi` nodes—Eclipse scales the image for the `drawable-ldpi` node so you don't need to paste an image to this node. Accomplish this task by following steps similar to previous steps.

Finally, you need to replace the default `AndroidManifest.xml` node content with the `AndroidManifest.xml` file content that was generated when you built this app at the command line:

1. Copy the previous `AndroidManifest.xml` file contents to the clipboard.
2. Right-click the `AndroidManifest.xml` node and select Paste from the pop-up menu.

Now that the file structure has been specified, select Run from the menu bar, and, if displayed, select Android Application from the resulting Run As dialog box and click the OK button. After a few moments, the emulated MyAVD device should start running and launch Planets. You should see the same screen that appears in Figure C-26.

Preparing an App for Publication on Google Play

At some point, you'll want to publish your apps on Google Play (<https://play.google.com/store>). Before publishing an app, you need to follow 6 preparation steps:

1. Test the app thoroughly.
2. Version the app in the manifest.
3. Request all necessary permissions in the manifest.
4. Build the app in release mode.
5. Sign the app package.

6. Align the app package.

After completing these steps, register to upload apps on Google Play (if you haven't done so already) and then upload the app's APK file.

Test the App Thoroughly

Android supports various versions, device categories (handsets and tablets), and device characteristics (such as screen densities and the presence or absence of a camera), which collectively offer a challenging environment for developing apps. It's important to test the app thoroughly for all desired version/category/characteristic combinations.

Android provides tools and resources to help you with this testing. For example, Android includes JUnit-based unit testing via the packages `junit.framework` and `junit.runner`. Check out the "Testing" (<http://developer.android.com/tools/testing/index.html>) section in Google's Android documentation for more information.

Version the App in the Manifest

Android lets you add version information to an app by specifying this information in `AndroidManifest.xml`'s `<manifest>` tag via its `versionCode` and `versionName` attributes:

- `versionCode` is assigned an integer value that represents the version of the app's code. The value is an integer so that other apps can programmatically evaluate it to check an upgrade or downgrade relationship, for example. Although you can set the value to any desired integer, you should ensure that each successive release of the app uses a greater value. Android doesn't enforce this behavior but increasing the value in successive releases is normative.
- `versionName` is assigned a string value that represents the release version of the app's code and it should be shown to users (by the app). This value is a string so that you can describe the app version as a `<major>.<minor>.<point>` string, or as any other type of absolute or relative version identifier. As with `android:versionCode`, Android doesn't use this value for any internal purpose. Publishing services may extract the `versionName` value for display to users.

The `<manifest>` tag in `Planets's AndroidManifest.xml` file includes a `versionCode` attribute initialized to "1" and a `versionName` attribute initialized to "1.0".

While on the subject of versioning, you should also specify the minimum SDK version that the app supports. You can accomplish this task by introducing, into `AndroidManifest.xml`, a `<uses-sdk>` element whose `minSdkVersion` attribute is set to the desired minimum API level. For example, the following `<uses-sdk>` element sets this level to 10 (Gingerbread/2.3.3):

```
<uses-sdk android:minSdkVersion="10"/>
```

Request All Necessary Permissions in the Manifest

An app may need to obtain permission before performing some task. For example, if the app uses the `android.webkit.WebView` class to view web pages over the Internet, you must add the following `<uses-permission>` element to `AndroidManifest.xml`:

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

Build the App in Release Mode

You cannot publish an app built in debug mode; you must rebuild the app in release mode. Accomplish this task by executing the following command:

```
ant release
```

Assuming that `Planets` is being built in release mode, the `bin` directory should contain a `Planets-release-unsigned.apk` file.

Sign the App Package

Android requires that all installed apps be digitally signed with a certificate whose private key is held by the app's developer. It uses the certificate as a means of identifying the app's author and establishing trust relationships between apps; it doesn't use the certificate to control which apps can be installed by the user.

■ **Note** Certificates don't need to be signed by certificate authorities: it's perfectly allowable, and typical, for Android apps to use self-signed certificates.

Android tests a signer certificate's expiration date only at install time. If an app's signer certificate expires after the app is installed, the app will continue to function normally.

Before you can sign an app package, you must obtain a suitable private key. A private key is suitable when it meets the following criteria:

- The key represents the personal, corporate, or organizational entity to be identified with the app.
- The key has a validity period that exceeds the expected lifespan of the app. Google recommends a validity period of more than 25 years. If you plan to publish the app on Google Play, keep in mind that a validity period ending after October 22, 2033 is a requirement. You cannot upload an app when it's signed with a key whose validity expires before (and possibly even on) that date.
- The key isn't the debug key generated by the Android SDK tools.

The JDK's `keytool` tool is used to create a suitable private key. The following command line (split over two lines for readability), which assumes that `C:\prj\android\Planets` is the current directory, uses `keytool` to generate this key:

```
keytool -genkey -v -keystore planets-release-key.keystore -alias planets_key  
-keyalg RSA -keysize 2048 -validity 10000
```

The following command-line arguments are specified:

- `-genkey` causes `keytool` to generate a public key and a private key (a key pair).
- `-v` enables verbose output.
- `-keystore` identifies the *keystore* (a database of private keys and their associated X.509 certificate chains authenticating the corresponding public keys) that stores the private key; the keystore is named `planets-release-key.keystore` on the command line.
- `-alias` identifies an alias for the keystore entry (only the first 8 characters are used when the alias is specified during the actual signing operation); the alias is named `planets_key` on the command line.
- `-keyalg` specifies the encryption algorithm to use when generating the key; DSA and RSA are supported, and RSA is specified on the command line.
- `-keysize` specifies the size of each generated key (in bits); 2048 is specified on the command line because Google recommends using a key size of 2048 bits or higher (the default size is 1024 bits).
- `-validity` specifies the period (in days) in which the key remains valid (Google recommends a value of 10000 or greater); 10000 is specified on the command line.

keytool prompts you for a password (to protect access to the keystore) and will then prompt you to reenter the same password. It then prompts for your first and last name, your organizational unit name, the name of your organization, the name of your city or locality, the name of your state or province, and a two-letter country code for your organizational unit.

keytool subsequently prompts you to indicate whether or not this information is correct (by typing yes and pressing Enter, or by pressing Enter for no). Assuming you entered yes, keytool lets you choose a different password for the key, or you can use the same password as that of the keystore.

■ **Caution** Keep your private key secure. Fail to do so and your app authoring identity and user trust could be compromised. Here are some tips for keeping your private key secure:

- * Select strong passwords for the keystore and key.
- * When you generate your key with keytool, don't supply the `-storepass` and `-keypass` options on the command line. If you do so, your passwords will be available in your shell history, which any user on your computer can access.
- * When signing your apps with jarsigner, don't supply the `-storepass` and `-keypass` options on the command line (for the same reason as mentioned in the previous tip).
- * Don't give or lend anyone your private key, and don't let unauthorized persons know your keystore and key passwords.

keytool creates `planets-release-key.keystore` in the current directory. You can view this keystore's information by executing the following command line:

```
keytool -list -v -keystore planets-release-key.keystore
```

After requesting the keystore password, keytool outputs the number of entries in the keystore (which should be one) and certificate information.

The JDK's jarsigner tool is used to sign `Planets-release-unsigned.apk`. Assuming that `C:\prj\android\Planets` is the current directory, this directory contains the keytool-created `planets-release-key.keystore` file, and this directory contains a `bin` subdirectory that contains `Planets-release-unsigned.apk` execute the following command line (split over two lines for readability) to sign this file:

```
jarsigner -verbose -keystore planets-release-key.keystore  
bin/Planets-release-unsigned.apk planets_key
```

The following command-line arguments are specified:

- `-verbose` enables verbose output.

- `-keystore` identifies the keystore that stores the private key; `planets-release-key.keystore` is specified in the command line.
- `bin/Planets-release-unsigned.apk` identifies the location and name of the APK being signed.
- `planets-key` identifies the previously created alias for the private key.

`jarsigner` prompts you to enter the keystore password that you previously specified via `keytool`. This tool then outputs various messages that identify the files added to the JAR file and the files that have been signed.

■ **Note** In the past, the previous `jarsigner` command was problematic with JDK 7. After signing the release version of the APK file (and aligning the APK, which is discussed shortly), the APK cannot be installed on the device. This problem and its solution, which consists of adding `-digestalg SHA1 -sigalg MD5withRSA` to the command line is documented at <http://code.google.com/p/android/issues/detail?id=19567>. For JDK 7 users, the following command line (split over three lines for readability) should be used instead:

```
jarsigner -verbose -keystore planets-release-key.keystore
         bin/Planets-release-unsigned.apk -digestalg SHA1
         -sigalg MD5withRSA planets_key
```

This problem appears to have been corrected for Android 4.4.2.

Execute the following command line (split over two lines for readability) to verify that `Planets-release-unsigned.apk` has been signed:

```
jarsigner -verify -keystore planets-release-key.keystore
         bin/Planets-release-unsigned.apk
```

Assuming success, you should notice a single “`jar verified.`” message.

Align the App Package

As a performance optimization, Android requires that a signed APK’s uncompressed content be aligned relative to the start of the file; it supplies the `zipalign` SDK tool for this task. According to Google’s documentation, all uncompressed data within the APK, such as images or raw files, are aligned on 4-byte boundaries.

`zipalign` requires the following syntax to align an input APK to an output APK:

```
zipalign [-f] [-v] alignment infile.apk outfile.apk
```

The following command-line arguments are specified:

- `-f` forces *outfile.apk* to be overwritten if it exists.
- `-v` enables verbose output.
- *alignment* specifies that the APK content is to be aligned on this number of bytes boundary; it appears that `zipalign` ignores any value other than 4.
- *infile.apk* identifies the signed APK file to be aligned.
- *outfile.apk* identifies the resulting signed and aligned APK file.

Assuming that `C:\prj\android\Planets\bin` is the current directory, execute the following command line to align `Planets-release-unsigned.apk` to `Planets.apk`:

```
zipalign -f -v 4 Planets-release-unsigned.apk Planets.apk
```

`zipalign` requires the following syntax to verify that an existing APK is aligned:

```
zipalign -c -v alignment existing.apk
```

The following command-line arguments are specified:

- `-c` confirms the alignment of *existing.apk*.
- `-v` enables verbose output.
- *alignment* specifies that the APK content is aligned on this number of bytes boundary; it appears that `zipalign` ignores any value other than 4.
- *existing.apk* identifies the signed APK file to be aligned.

Execute the following command line to verify that `Planets.apk` is aligned:

```
zipalign -c -v 4 Planets.apk
```

`zipalign` presents a list of APK entries, indicating which are compressed and which are not, followed by a verification successful or a verification failed message.

`Planets.apk` is now ready for publication.

Guidelines for Successful Apps

Google has created a series of articles that can help you create exceptional apps that stand out from the crowd. The following list presents a representative sample of these articles:

- Keeping Your App Responsive (<http://developer.android.com/training/articles/perf-anr.html>)
- Managing Your App's Memory (<http://developer.android.com/training/articles/memory.html>)
- Optimizing Battery Life (<http://developer.android.com/training/monitoring-device-state/index.html>)
- Performance Tips (<http://developer.android.com/training/articles/perf-tips.html>)
- Security Tips (<http://developer.android.com/training/articles/security-tips.html>)
- Testing Your Android Activity (<http://developer.android.com/training/activity-testing/index.html>)

I recommend that you read these and similar articles to learn more about developing exceptional apps. For example, among other advices, the “Performance Tips” article recommends that you prefer library code over rolling your own. As an example, it mentions that “the `System.arraycopy()` method is about 9x faster than a hand-coded loop on a Nexus One with the JIT.”

Additionally, there are some quality guidelines that you should consider before (and even after) publishing your apps to Google Play:

- Core App Quality Guidelines (<http://developer.android.com/distribute/googleplay/quality/core.html>)
- Tablet App Quality Checklist (<http://developer.android.com/distribute/googleplay/quality/tablet.html>)
- Improving App Quality After Launch (<http://developer.android.com/distribute/googleplay/strategies/app-quality.html>)

Conclusion

In this appendix I summarized some of what you need to know to develop Android apps. I couldn't dig deeper because I don't want to create a book within a book. Because you'll probably want to learn more about Android app development, I recommend that you browse through Google's Android Developers website (<http://developer.android.com/index.html>). I wish you success.

■ **Note** I'm not making any promises but I might create some additional Java/Android material that can be downloaded from my website (<http://tutortutor.ca/cgi-bin/makepage.cgi?/books/ljfad>). For example, I might create an Appendix D that explores the security APIs offered by Java and Android. I wouldn't include security APIs that Android doesn't support, such as `java.lang.SecurityManager`.