Use this definitive Android reference
to take your apps further

Pro Android 5

Dave MacLean | Satya Komatineni | Grant Allen

Foreword by Grant Allen, Google

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Welcome to the wonderful world of Android. A world where, with a bit of knowledge and effort, you too can write Android applications. To write good applications, however, you will need to dig deeper, to understand the fundamentals of the Android architecture, to understand how applications can work together, to understand how mobile applications are different from all previous forms of programming. The online documentation on Android is fair, but it does not go far enough. You can read the source code, but that's not at all easy.

This book is the culmination of seven years of researching, developing, testing, refining, and writing about Android. We've read all the online documentation, scoured through source code, explored the far reaches of the Internet, and have compiled this book. We've filled in the gaps, anticipated the questions you have, and provided answers. Along the way we've seen APIs come and go and be revised. We've seen major changes in how applications are constructed. At first we all used Activities, but when tablets came along we started using Fragments. We've taken everything we've learned and filled this book with practical guidance to using the latest Android APIs to write interesting applications.

You will still find coverage of the beginning topics, to help the new learner get started developing for Android. You will also find coverage of the more advanced topics, such as Google Maps Android API v2, which is very different from v1. We've updated this edition with the latest information on the available APIs. You will find in-depth coverage of intents, services, broadcast receivers, communication, fragments, widgets, sensors, animation, security, loaders, persistence, Google Cloud Messaging, audio and video, and more. And for every topic there are sample programs that illustrate each API in meaningful ways. All source code is downloadable, so you can copy and paste it into your applications to get a great head start.

Chapter 1

Hello Android

Welcome to the book, and welcome to the world of Android development. In a little under ten years, Android has helped change the face of modern mobile computing and telephony and launched a revolution in how applications are developed, and by whom. With this book in your hands, you are now part of the great Android explosion! We're going to assume that you want to get straight at working with Android, so we're not going to bore you with a fireside chat about Android's history, major characters, plaudits, or any other prose. We're going to get straight to it!

In this chapter, you'll start by seeing what you need to begin building applications with the Android software development kit (SDK) and set up your choice of development environment. Next, you step through a "Hello World!" application. Then the chapter explains the Android application life cycle and ends with a discussion about running your applications with Android Virtual Devices (AVDs) and on real devices. So let's get started.

Prerequisites for Android Development

To build applications for Android, you need the Java SE Development Kit (JDK), the Android SDK, and a development environment. Strictly speaking, you can develop your applications using nothing more than a primitive text editor and a handful of command-line tools like Ant. For the purposes of this book, we'll use the commonly available Eclipse IDE, though you are free to adopt Android Studio and its IntelliJ underpinnings—we'll even walk through Android Studio for those who have not seen it. With the exception of a few add-on tools, the examples we share in the book will work equally well between these two IDEs.

The Android SDK requires JDK 6 or 7 (the full JDK, not just the Java Runtime Environment [JRE]) and optionally a supported IDE. Currently, Google directly supports two alternative IDEs, providing some choice. Historically, Eclipse was the first IDE supported by Google for Android development, and developing for Android 4.4 KitKat or 5.0 Lollipop requires Eclipse 3.6.2 or higher (this book uses Eclipse 4.2 or 4.4, also known as Juno and Luna, respectively, and other versions). The alternative environment released and supported by Google for Android is now known as Android Studio. This is a packaged version of IDEA IntelliJ with built-in Android SDK and developer tools.

> **Note** At the time of this writing, Java 8 was available but not yet supported by the Android SDK. In previous versions of the Android SDK, Java 5 was also supported, but this is no longer the case. The latest version of Eclipse (4.4, a.k.a. Juno) was also available, but Android has historically not been reliable on the latest Eclipse right away. Check the system requirements here to find the latest: http://developer.android.com/sdk/index.html.

The Android SDK is compatible with Windows (Windows XP, Windows Vista, and Windows 7), Mac OS X (Intel only), and Linux (Intel only). In terms of hardware, you need an Intel machine, the more powerful the better.

To make your life easier, if you choose Eclipse as your IDE, you will want to use Android development tools (ADT). ADT is an Eclipse plug-in that supports building Android applications with the Eclipse IDE.

The Android SDK is made up of two main parts: the tools and the packages. When you first install the SDK, all you get are the base tools. These are executables and supporting files to help you develop applications. The packages are the files specific to a particular version of Android (called a *platform*) or a particular add-on to a platform. The platforms include Android 1.5 through 4.4.2. The add-ons include the Google Maps API, the Market License Validator, and even vendor-supplied ones such as Samsung's Galaxy Tab add-on. After you install the SDK, you then use one of the tools to download and set up the platforms and add-ons.

Remember, you only need to set up and configure one of Eclipse or Android Studio. You can use both if you are so inclined, but it's certainly not required. Let's get started!

Setting Up Your Eclipse Environment

In this section, you walk through downloading JDK 6, the Eclipse IDE, the Android SDK (tools and packages), and ADT. You also configure Eclipse to build Android applications. Google provides a page to describe the installation process (http://developer.android.com/sdk/installing.html) but leaves out some crucial steps, as you will see.

Downloading JDK

The first thing you need is the JDK. The Android SDK requires JDK 6 or higher; we've developed our examples using JDK 6 and 7, depending on the version of Eclipse or Android Studio in use. For Windows and Mac OS X, download JDK 7 from the Oracle web site (www.oracle.com/technetwork/java/javase/ downloads/index.html) and install it. You only need the JDK, not the bundles. To install the JDK for Linux, open a Terminal window and instruct your package manager to install it. For example, in Debian or Ubuntu try the following:

```
sudo apt-get install sun-java7-jdk
```

This should install the JDK plus any dependencies such as the JRE. If it doesn't, it probably means you need to add a new software source and then try that command again. The web page https://help.ubuntu.com/community/Repositories/Ubuntu explains software sources and how to add the connection to third-party software. The process is different depending on which version of Linux you have. After you've done that, retry the command.

With the introduction of Ubuntu 10.04 (Lucid Lynx), Ubuntu recommends using OpenJDK instead of the Oracle/Sun JDK. To install OpenJDK, try the following:

```
sudo apt-get install openjdk-7-jdk
```

If this is not found, set up the third-party software as outlined previously and run the command again. All packages on which the JDK depends are automatically added for you. It is possible to have both OpenJDK and the Oracle/Sun JDK installed at the same time. To switch active Java between the installed versions of Java on Ubuntu, run this command at a shell prompt

```
sudo update-alternatives --config java
```

and then choose which Java you want as the default.

Now that you have a Java JDK installed, it's time to set the `JAVA_HOME` environment variable to point to the JDK install folder. To do this on a Windows XP machine, choose Start ➤ My Computer, right-click, select Properties, choose the Advanced tab, and click Environment Variables. Click New to add the variable or Edit to modify it if it already exists. The value of `JAVA_HOME` is something like `C:\Program Files\Java\jdk1.7.0_79`.

For Windows Vista and Windows 7, the steps to get to the Environment Variables screen are a little different. Choose Start ➤ Computer, right-click, choose Properties, click the link for Advanced System Settings, and click Environment Variables. After that, follow the same instructions as for Windows XP to change the `JAVA_HOME` environment variable.

For Mac OS X, you set `JAVA_HOME` in the `.bashrc` file in your home directory. Edit or create the `.bashrc` file, and add a line that looks like this

```
export JAVA_HOME=path_to_JDK_directory
```

where `path_to_JDK_directory` is probably `/Library/Java/Home`. For Linux, edit your `.bashrc` file and add a line like the one for Mac OS X, except that your path to Java is probably something like `/usr/lib/jvm/java-6-sun` or `/usr/lib/jvm/java-6-openjdk`.

Downloading Eclipse

After the JDK is installed, you can download the Eclipse IDE for Java Developers. (You don't need the edition for Java EE; it works, but it's much larger and includes things you don't need for this book.) The examples in this book use Eclipse 4.2 or 4.4 (on both Linux and Windows environments). You can download all versions of Eclipse from www.eclipse.org/downloads/.

> **Note** As an alternative to the individual steps presented here, you can also download the ADT Bundle from the Android developer site. This includes Eclipse with built-in developer tools and the Android SDK in one package. It's a great way to get started quickly, but if you have an existing environment, or just want to know how all the components are stitched together, then following the step-by-step instructions is the way to go.

The Eclipse distribution is a `.zip` file that can be extracted just about anywhere. The simplest place to extract to on Windows is `C:\`, which results in a `C:\eclipse` folder where you find `eclipse.exe`. Depending on your security configuration, Windows may insist on enforcing UAC when running from C:\. For Mac OS X, you can extract to `Applications`. For Linux, you can extract to your home directory or have your administrator put Eclipse into a common place where you can get to it. The Eclipse executable is in the `eclipse` folder for all platforms. You may also find and install Eclipse using Linux's Software Center for adding new applications, although this may not provide you with the latest version.

When you first start up Eclipse, it asks you for a location for the workspace. To make things easy, you can choose a simple location such as `C:\android` or a directory under your home directory. If you share the computer with others, you should put your workspace folder somewhere underneath your home directory.

Downloading the Android SDK

To build applications for Android, you need the Android SDK. As stated before, the SDK comes with the base tools; then you download the package parts that you need and/or want to use. The tools part of the SDK includes an emulator so you don't need a mobile device with the Android OS to develop Android applications. It also has a setup utility to allow you to install the packages that you want to download.

You can download the Android SDK from `http://developer.android.com/sdk`. It ships as a `.zip` file, similar to the way Eclipse is distributed, so you need to unzip it to an appropriate location. For Windows, unzip the file to a convenient location (we used the `C:` drive), after which you should have a folder called something like `C:\android-sdk-windows` that contains the files as shown in Figure 1-1. For Mac OS X and Linux, you can unzip the file to your home directory. Notice that Mac OS X and Linux do not have an SDK Manager executable; the equivalent of the SDK Manager in Mac OS X and Linux is to run the `tools/android` program.

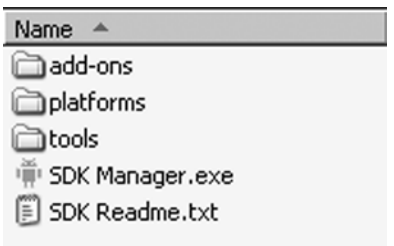

Figure 1-1. *Base contents of the Android SDK*

An alternative approach (for Windows only) is to download an installer EXE instead of the zip file and then run the installer executable. This executable checks for the Java JDK, unpacks the embedded files for you, and runs the SDK Manager program to help you set up the rest of the downloads.

Whether through using the Windows installer or by executing the SDK Manager, you should install some packages next. When you first install the Android SDK, it does not come with any platform versions (that is, versions of Android). Installing platforms is pretty easy. After you've launched the SDK Manager, you see what is installed and what's available to install, as shown in Figure 1-2. You must add Android SDK tools and platform-tools in order for your environment to work. Because you use it shortly, add at least the Android 1.6 SDK platform, as well as the latest platform shown in your installer.

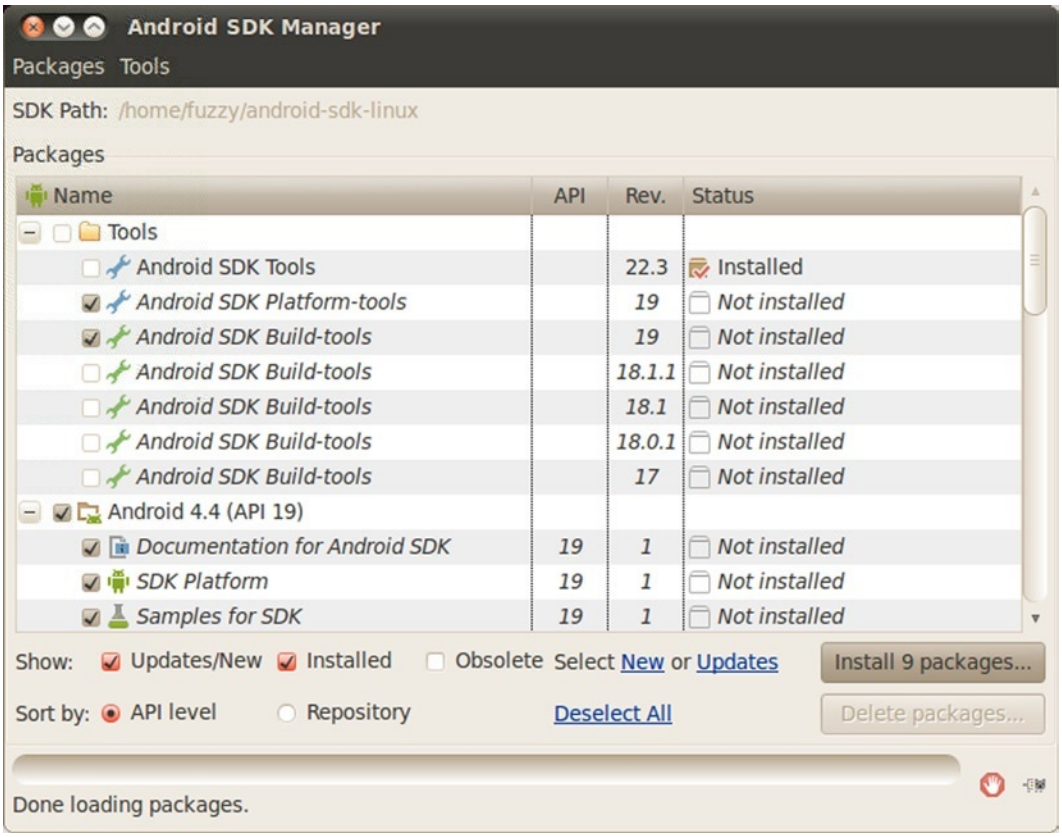

***Figure 1-2.** Adding packages to the Android SDK*

Click the Install button. You need to click Accept for each item you're installing (or Accept All) and then click Install. Android then downloads your packages and platforms to make them available to you. The Google APIs are add-ons for developing applications using Google Maps. You can always come back to add more packages later.

Updating Your PATH Environment Variable

The Android SDK comes with a `tools` directory that you want to have in your PATH. You also need in your PATH the `platform-tools` directory you just installed. Let's add them now or, if you're upgrading, make sure they're correct. While you're there, you can also add a JDK bin directory, which will make life easier later.

For Windows, get back to the Environment Variables window. Edit the PATH variable and add a semicolon (;) on the end, followed by the path to the Android SDK tools folder, followed by another semicolon, followed by the path to the Android SDK platform-tools folder, followed by another semicolon, and then `%JAVA_HOME%\bin`. Click OK when you're done. For Mac OS X and Linux, edit your `.bashrc` file and add the Android SDK tools directory path to your PATH variable, as well as the Android SDK `platform-tools` directory and the `$JAVA_HOME/bin` directory. Something like the following works for Linux:

```
export PATH=$PATH:$HOME/android-sdk-linux_x86/tools:$HOME/android-sdk-linux_x86/platform-
tools:$JAVA_HOME/bin
```

Just make sure that the PATH component that's pointing to the Android SDK tools directories is correct for your particular setup.

The Tools Window

Later in this book, there are times when you need to execute a command-line utility program. These programs are part of the JDK or part of the Android SDK. By having these directories in your PATH, you don't need to specify the full pathnames in order to execute them, but you need to start up a *tools window* in order to run them (later chapters refer to this tools window). The easiest way to create a tools window in Windows is to choose Start ➤ Run, type in `cmd`, and click OK. For Mac OS X, choose Terminal from your `Applications` folder in Finder or from the Dock if it's there. For Linux, run your favorite terminal.

You may need to know the IP address of your workstation later. To find this in Windows, launch a tools window and enter the command `ipconfig`. The results contain an entry for IPv4 (or something like that) with your IP address listed next to it. An IP address looks something like this: 192.168.1.25. For Mac OS X and Linux, launch a tools window and use the command `ifconfig`. You find your IP address next to the label `inet addr`.

You may see a network connection called localhost or lo; the IP address for this network connection is 127.0.0.1. This is a special network connection used by the operating system and is not the same as your workstation's IP address. Look for a different number for your workstation's IP address.

Installing ADT

Now you need to install ADT (very recently renamed to GDT, the Google Developer Tools), an Eclipse plug-in that helps you build Android applications. Specifically, ADT integrates with Eclipse to provide facilities for you to create, test, and debug Android applications. You need to use the Install New Software facility in Eclipse to perform the installation. (The instructions for upgrading ADT appear later in this section.) To get started, launch the Eclipse IDE and follow these steps:

1. Select Help ➤ Install New Software.
2. Select the Work With field, type in

   `https://dl-ssl.google.com/android/eclipse/`,

   and press Enter. Eclipse contacts the site and populates the list as shown in Figure 1-3.

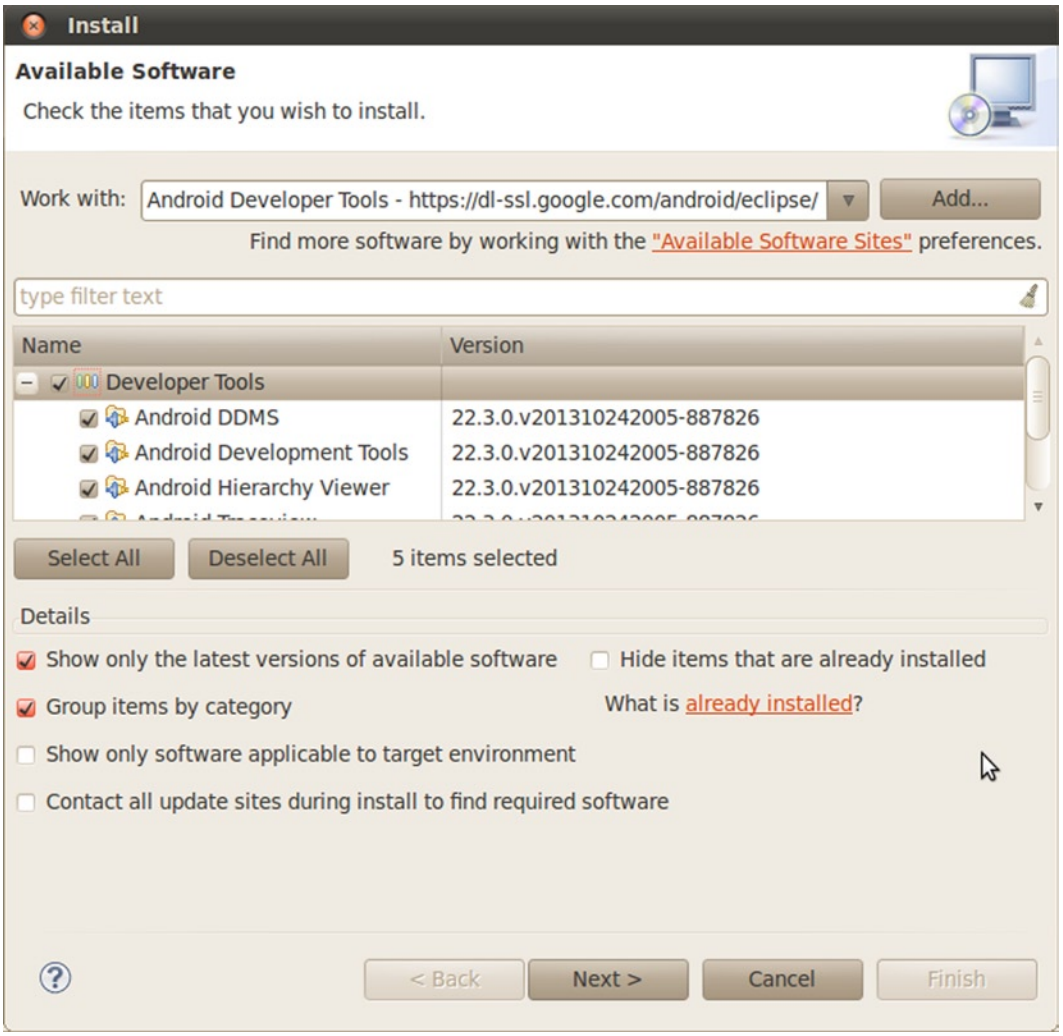

Figure 1-3. Installing ADT using the Install New Software feature in Eclipse

3. You should see an entry named Developer Tools with four child nodes: Android DDMS, Android Development Tools, Android Hierarchy Viewer, and Android Traceview. Just before publishing this book, Google updated the ADT to be part of the more generic Google Developer Tools plugin for Eclipse, or GDT. Look for the same options in the GDT. Select the parent node Developer Tools, make sure the child nodes are also selected, and click the Next button. The versions you see may be newer than these, and that's okay. You may also see additional tools. These tools are explained further in Chapter 11.
4. Eclipse asks you to verify the tools to install. Click Next.
5. You're asked to review the licenses for ADT as well as for the tools required to install ADT. Review the licenses, click "I accept," and then click the Finish button.

Eclipse downloads the developer tools and installs them. You need to restart Eclipse for the new plug-in to show up in the IDE.

If you already have an older version of ADT in Eclipse, go to the Eclipse Help menu and choose Check for Updates. You should see the new version of ADT and be able to follow the installation instructions, picking up at step 3.

> **Note** If you're doing an upgrade of ADT, you may not see some of these tools in the list of tools to be upgraded. If you don't see them, then after you've upgraded the rest of the ADT, go to Install New Software and select `https://dl-ssl.google.com/android/eclipse/` from the Works With menu. The middle window should show you other tools that are available to be installed.

The final step to make ADT functional in Eclipse is to point it to the Android SDK. In Eclipse, select Window ➤ Preferences. (On Mac OS X, Preferences is under the Eclipse menu.) In the Preferences dialog box, select the Android node and set the SDK Location field to the path of the Android SDK (see Figure 1-4) and then click the Apply button. Note that you may see a dialog box asking if you want to send usage statistics to Google concerning the Android SDK; that decision is up to you.

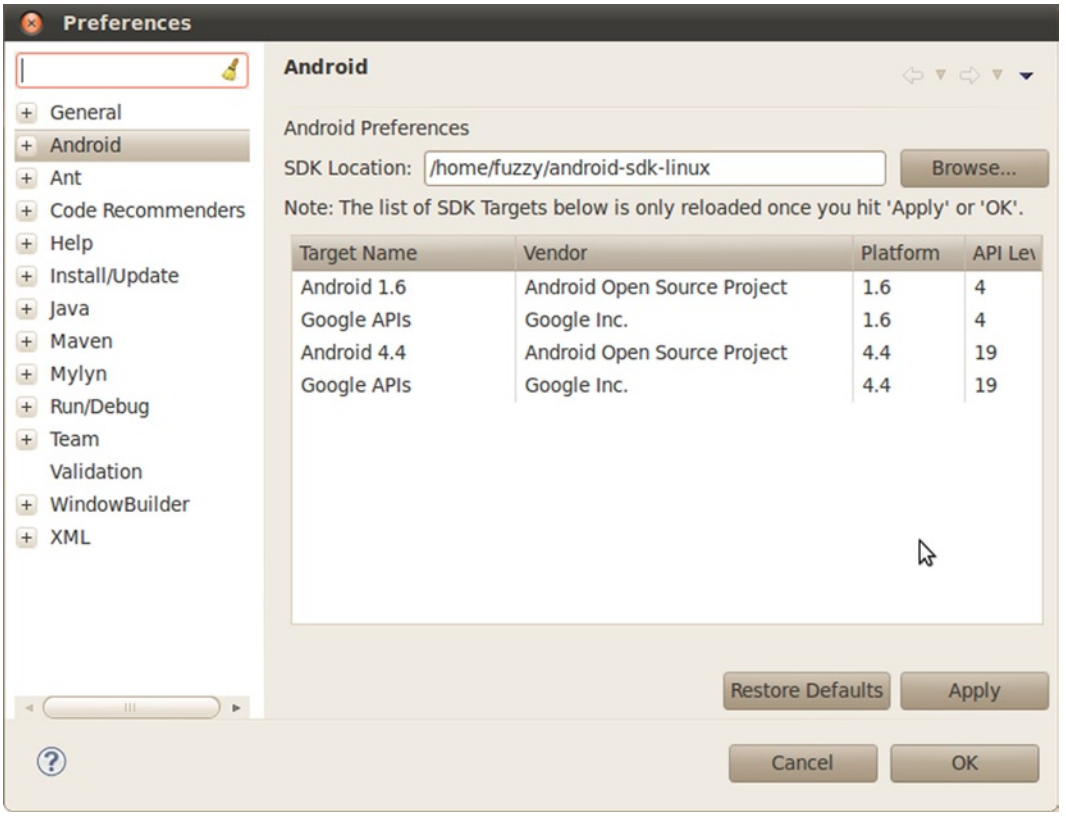

Figure 1-4. *Pointing ADT to the Android SDK*

You may want to make one more Preferences change on the Android ➤ Build page. The Skip Packaging option should be checked if you'd like to make your file saves faster. By default, the ADT readies your application for launch every time it builds it. By checking this option, packaging and indexing occur only when truly needed.

From Eclipse, you can launch the SDK Manager. To do so, choose Window ➤ Android SDK Manager. You should see the same window as in Figure 1-2.

If you've chosen Eclipse as your IDE, you are almost ready for your first Android application—you can skip the following section on Android Studio and head straight to the "Learning Android's Fundamental Components" section.

Setting Up Your Android Studio Environment

In 2013, Google introduced a second supported development environment, known as Android Studio (or Android Developer Studio at the time of launch). This is based around a popular Java IDE: IDEA IntelliJ. The most important thing to know about Android Studio is that it is still a work in progress. As of this book's writing, the latest version is 1.2. Anyone familiar with the vagaries of version numbers knows that starting with a low number usually means "beware!"

The second most important thing to remember is that Android Studio currently assumes a 64-bit development environment. That means dependencies like Java also need to be 64-bit.

The next sections *briefly* cover the setup of Android Studio for those interested or gung-ho enough to use it. Be mindful that the rest of the book predominantly shows examples and options using Eclipse.

Java requirements for Android Studio

Like Eclipse, Android Studio relies on a working Java installation. Android Studio will attempt to automatically discover your Java environment during installation, so it pays to have Java installed and configured.

For Java installation, remember that Android Studio is 64-bit. In all other respects, you can follow the preceding section titled "Downloading JDK"—we won't repeat that word-for-word here to save some trees. Ensure you follow all the instructions there, including setting the `JAVA_HOME` environment variable, as this is the main indicator used by the Android Studio installer to find your Java installation.

Downloading and Installing Android Studio

Google makes Android Studio available from the main Android development site, currently at the URL `http://developer.android.com/sdk/installing/studio.html`. That may change at any time, but a quick search on the developer.android.com site should find it. Android Studio is packaged as a monolithic bundle, with nearly all the components you need. The Java SDK is the exception—we'll cover that shortly. The package downloaded from the preceding URL will be named something like android-studio-bundle-132.893413-windows.exe for windows, or a similar name with a different extension for OS X and Linux, and includes the following:

- Current latest build of the Android Studio bundle of IntelliJ IDEA
- Built-in Android SDK
- All related Android build tools
- Android Virtual Device images

We'll talk more about these components in later chapters. For a Windows installation run the executable and follow the prompts to choose an installation path, and decide whether Android Studio is made available to all users on the Windows machine, or just the current user. For OS X, open the `.dmg` file and copy the Android Studio entry to your Applications folder. Under Linux, extract the contents of the `.tgz` file to your desired location.

Once installed, you can start Android Studio under Windows from the start menu folder you chose when prompted; under OS X from the Applications folder; and under Linux by running the `./android-studio/bin/studio.sh` file under your installation directory. Whatever the operating system, you should see the Android Studio home screen as depicted in Figure 1-5.

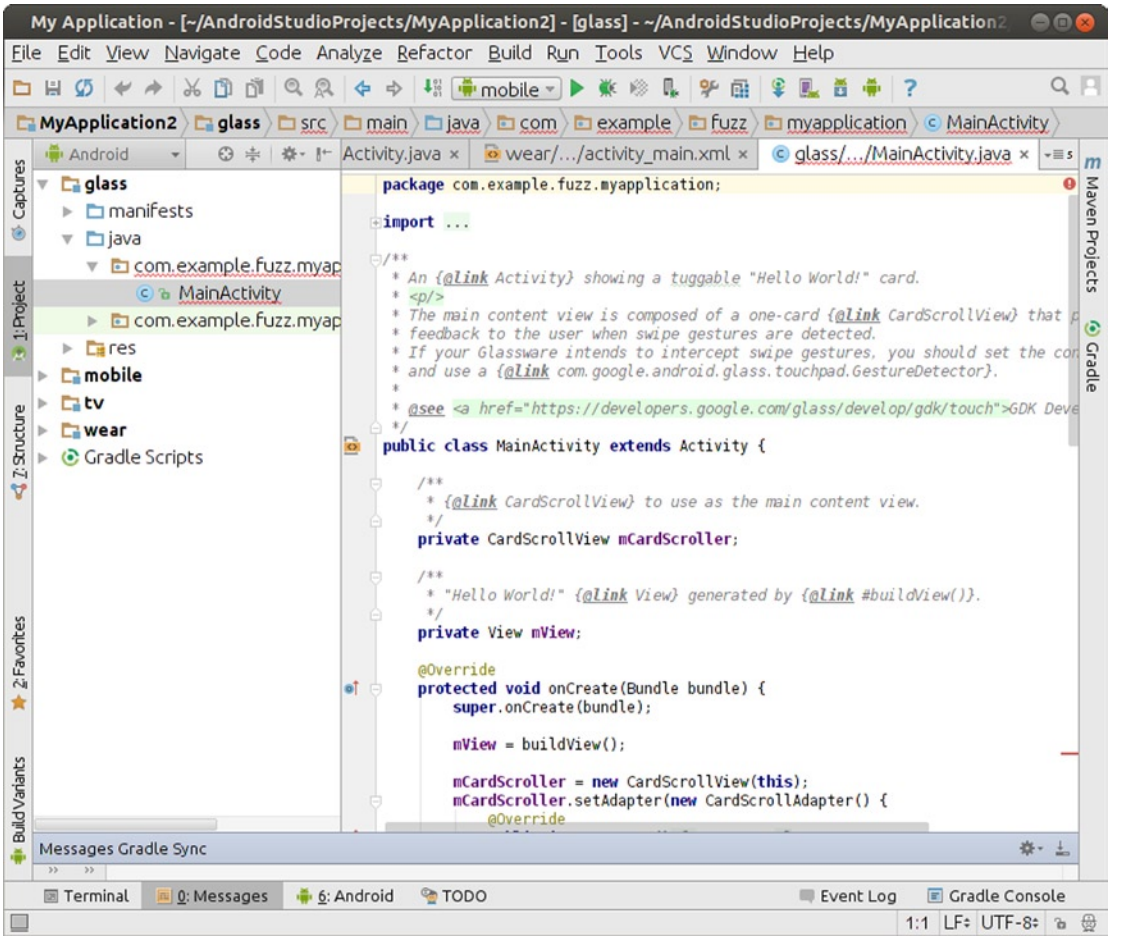

Figure 1-5. Android Studio when first launched

Learning Android's Fundamental Components

Every application framework has some key components that developers need to understand before they can begin to write applications based on the framework. For example, you need to understand JavaServer Pages (JSP) and servlets in order to write Java 2 Platform, Enterprise Edition (J2EE) applications. Similarly, you need to understand views, activities, fragments, intents, content providers, services, and the `AndroidManifest.xml` file when you build applications for Android. You briefly cover these fundamental concepts here and explore them in more detail throughout the book.

View

Views are user interface (UI) elements that form the basic building blocks of a user interface. A view can be a button, a label, a text field, or many other UI elements. If you're familiar with views in J2EE and Swing, then you understand views in Android. Views are also used as containers for views, which means there's usually a hierarchy of views in the UI. In the end, everything you see is a view.

Activity

An *activity* is a UI concept that usually represents a single screen in your application. It generally contains one or more views, but it doesn't have to. An activity is pretty much like it sounds—something that helps the user do one thing, which could be viewing data, creating data, or editing data. Most Android applications have several activities within them.

Fragment

When a screen is large, it becomes difficult to manage all of its functionality in a single activity. *Fragments* are like sub-activities, and an activity can display one or more fragments on the screen at the same time. When a screen is small, an activity is more likely to contain just one fragment, and that fragment can be the same one used within larger screens.

Intent

An *intent* generically defines an "intention" to do some work. Intents encapsulate several concepts, so the best approach to understanding them is to see examples of their use. You can use intents to perform the following tasks:

- Broadcast a message
- Start a service
- Launch an activity
- Display a web page or a list of contacts
- Dial a phone number or answer a phone call

Intents are not always initiated by your application—they're also used by the system to notify your application of specific events (such as the arrival of a text message).

Intents can be explicit or implicit. If you simply say that you want to display a URL, the system decides what component will fulfill the intention. You can also provide specific information about what should handle the intention. Intents loosely couple the action and action handler.

Content Provider

Data sharing among mobile applications on a device is common. Therefore, Android defines a standard mechanism for applications to share data (such as a list of contacts) without exposing the underlying storage, structure, and implementation. Through content providers, you can expose your data and have your applications use data from other applications.

Service

Services in Android resemble services you see in Windows or other platforms—they're background processes that can potentially run for a long time. Android defines two types of services: local services and remote services. Local services are components that are only accessible by the application that is hosting the service. Conversely, remote services are services that are meant to be accessed remotely by other applications running on the device.

An example of a service is a component that is used by an e-mail application to poll for new messages. This kind of service may be a local service if the service is not used by other applications running on the device. If several applications use the service, then it's implemented as a remote service.

AndroidManifest.xml

`AndroidManifest.xml`, which is similar to the `web.xml` file in the J2EE world, defines the contents and behavior of your application. For example, it lists your application's activities and services, along with the permissions and features the application needs to run.

AVDs

An AVD allows developers to test their applications without hooking up an actual Android device (typically a phone or a tablet). AVDs can be created in various configurations to emulate different types of real devices.

Hello World!

Now you're ready to build your first Android application. You start by building a simple "Hello World!" program. Create the skeleton of the application by following these steps:

1. Launch Eclipse, and select File ➤ New ➤ Project. In the New Project dialog box, select Android Application Project and then click Next. You see the New Android Project dialog box, as shown in Figure 1-6. (Eclipse may have added Android Project to the New menu, so you can use it if it's there.) There's also a New Android Project button on the toolbar.

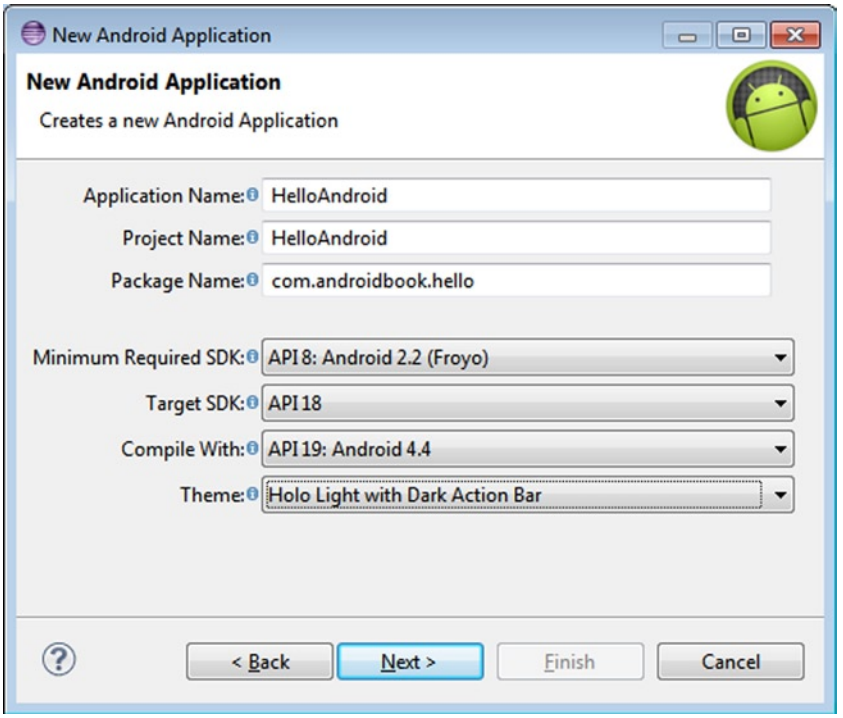

Figure 1-6. Using the New Project Wizard to create an Android application

2. As shown in Figure 1-6, enter **HelloAndroid** as the project name. You need to distinguish this project from other projects you create in Eclipse, so choose a name that will make sense to you when you are looking at all the projects in your Eclipse environment. You will also see the available Build Targets. Select Android 2.2. This is the version of Android you use as your base for the application. You can run your application on later versions of Android, such as 4.3 and 4.4; but Android 2.2 has all the functionality you need for this example, so choose it as your target. In general, it's best to choose the lowest version number you can, because that maximizes the number of devices that can run your application.

3. Leave the Project Name to auto-complete itself based on your Application Name.

4. Use **com.androidbook.hello** as the package name. Like all Java applications, your application must have a base package name, and this is it. This package name will be used as an identifier for your application and must be unique across all applications. For this reason, it's best to start the package name with a domain name that you own. If you don't own one, be creative to ensure that your package name won't likely be used by anyone else. Click Next.

5. The next window provides options for customer launcher icons, the actual directory for the workspace in which you source code and other files are stored, and several other options. Leave all of these at the default, and click Next.

6. The next window shows you the Configure Launcher Icon options and settings, as shown in Figure 1-7. Feel free to play with the options here, though any changes you make are cosmetic and affect the look of the launcher icon when your application is deployed, and not its actual logic. Click Next when ready.

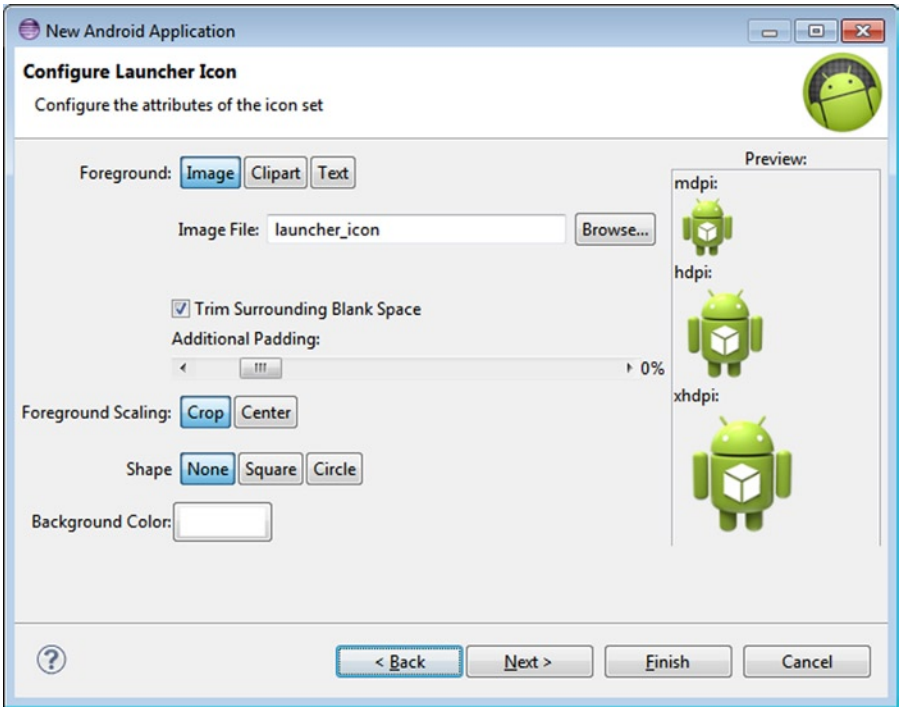

Figure 1-7. The Android launcher configuration options for a new Android project

7. You'll next see the Create Activity screen. Choose Blank Activity as the activity type, and click Next to move to the last screen of the wizard.

8. The final screen of the New Android Application wizard will be the Blank Activity details page. Type **HelloActivity** as the Activity Name. You're telling Android that this activity is the one to launch when your application starts up. You may have other activities in your application, but this is the first one the user sees. Allow the Layout Name to auto-populate with the value `activity_hello`.

9. Click the Finish button, which tells ADT to generate the project skeleton for you. For now, open the `HelloActivity.java` file under the `src` folder and modify the `onCreate()` method as follows:

```
/** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        /** create a TextView and write Hello World! */
        TextView tv = new TextView(this);
        tv.setText("Hello World!");
        /** set the content view to the TextView */
        setContentView(tv);
    }
```

You will need to add an `import android.widget.TextView;` statement at the top of the file with the other imports to get rid of the error reported by Eclipse. Save the `HelloActivity.java` file.

To run the application, you need to create an Eclipse launch configuration, and you need a virtual device on which to run it. We'll run quickly through these steps and come back later to more details about AVDs. Create the Eclipse launch configuration by following these steps:

1. Select Run ➤ Run Configurations.
2. In the Run Configurations dialog box, double-click Android Application in the left pane. The wizard inserts a new configuration named New Configuration.
3. Rename the configuration **RunHelloWorld**.
4. Click the Browse button, and select the HelloAndroid project.
5. Leave Launch Action set to Launch Default Activity. The dialog should appear as shown in Figure 1-8.

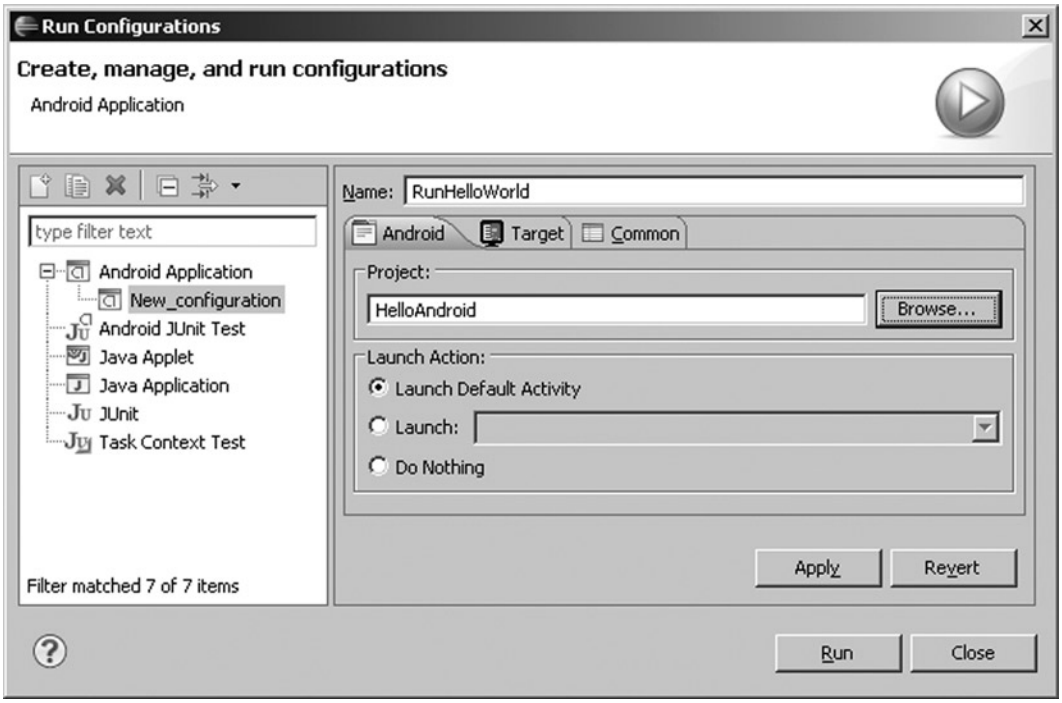

Figure 1-8. Configuring an Eclipse run configuration to run the "Hello World!" application

6. Click Apply and then Run. You're almost there! Eclipse is ready to run your application, but it needs a device on which to run it. As shown in Figure 1-9, you're warned that no compatible targets were found and asked if you'd like to create one. Click Yes.

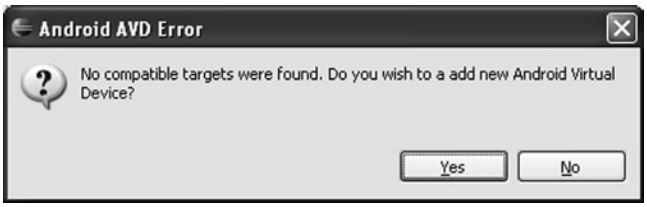

Figure 1-9. Error message warning about targets and asking for a new AVD

7. You're presented with a window that shows the existing AVDs (see Figure 1-10). You need to add an AVD suitable for your new application. Click the New button.

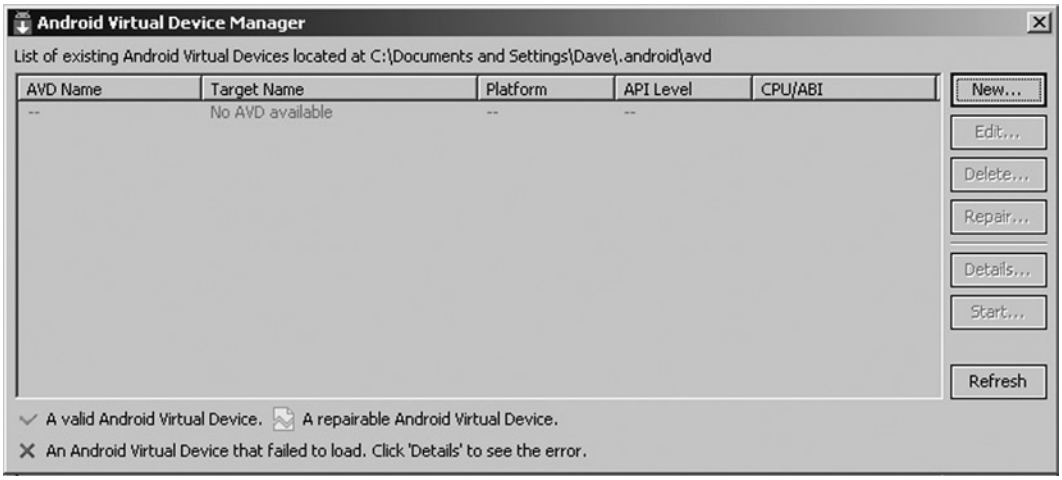

Figure 1-10. *The existing AVDs*

8. Fill in the Create AVD form as shown in Figure 1-11. Set Name to KitKat, choose Android 4.4 - API Level 19 (or some other version) for the Target, set SD Card Size to 64 (for 64MB), and choose other values as shown. Click Create AVD. The Manager may confirm the successful creation of your AVD. Close the AVD Manager window by clicking X in the upper-right corner.

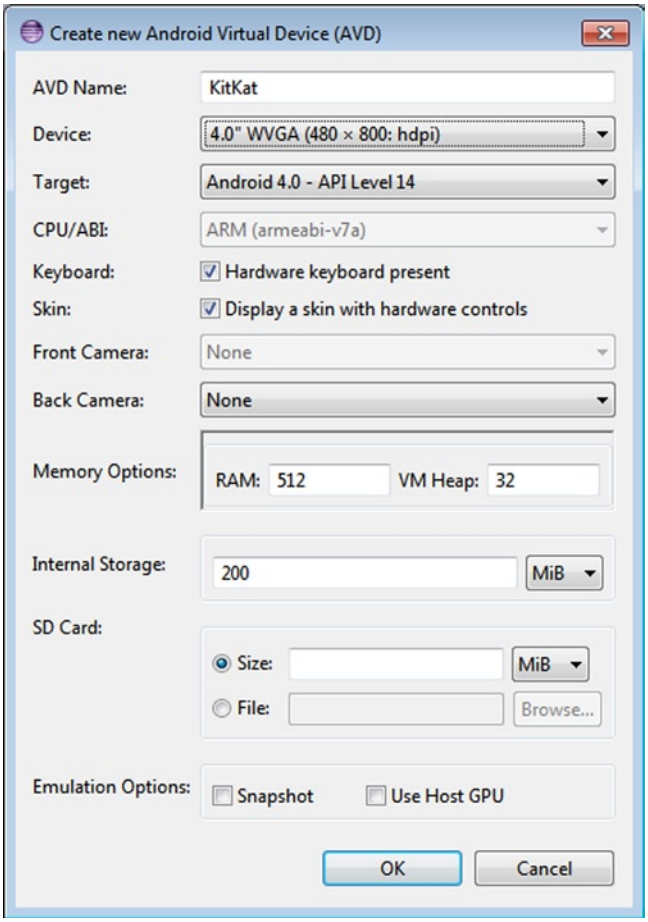

Figure 1-11. *Configuring an AVD*

> **Note** You're choosing a newer version of the SDK for your AVD, but your application can also run on an older one. This is okay because AVDs with newer SDKs can run applications that require older SDKs. The opposite, of course, is not true: an application that requires features of a newer SDK won't run on an AVD with an older SDK.

9. Select your new AVD from the bottom list. Note that you may need to click the Refresh button to make any new AVDs to show up in the list. Click the OK button.

10. Eclipse launches the emulator with your very first Android app (see Figure 1-12)!

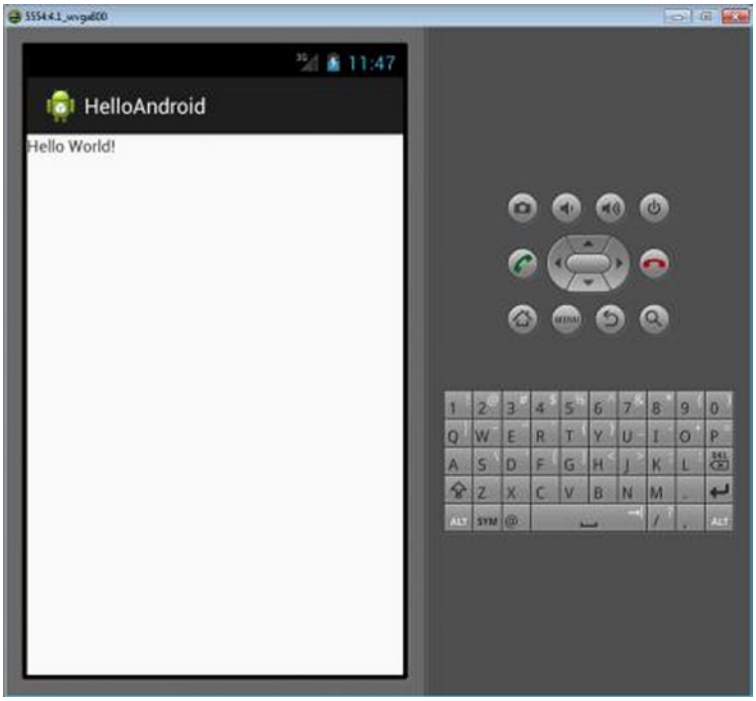

Figure 1-12. HelloAndroidApp running in the emulator

> **Note** It may take the emulator a while to emulate the device bootup process. Once the bootup process has completed, you typically see a locked screen. Click the Menu button or drag the unlock image to unlock the AVD. After unlocking, you should see HelloAndroidApp running in the emulator, as shown in Figure 1-11. Be aware that the emulator starts other applications in the background during the startup process, so you may see a warning or error message from time to time. If you do, you can generally dismiss it to allow the emulator to go to the next step in the startup process. For example, if you run the emulator and see a message like "application abc is not responding," you can either wait for the application to start or simply ask the emulator to forcefully close the application. Generally, you should wait and let the emulator start up cleanly.

Now you know how to create a new Android application and run it in the emulator. Next, we'll look more closely at AVDs, and also how to deploy to a real device.

AVDs

An AVD represents a device and its configuration. For example, you could have an AVD representing a really old Android device running version 1.5 of the SDK with a 32MB SD card. The idea is that you create AVDs you are going to support and then point the emulator to one of those AVDs when developing and testing your application. Specifying (and changing) which AVD to use is very easy and makes testing with various configurations a snap. Earlier, you saw how to create an AVD using Eclipse. You can make more AVDs in Eclipse by choosing Window ➤ Android Virtual Device Manager. You can also create AVDs using the command line with the utility named `android` under the `tools` directory (e.g., `c:\android-sdk-windows\tools\`). `android` allows you to create a new AVD and manage existing AVDs. For example, you can view existing AVDs, move AVDs, and so on by invoking `android` with the "avd" option. You can see the options available for using `android` by running `android -help`. For now, let's just create an AVD.

Running on a Real Device

The best way to test an Android app is to run it on a real device. Any commercial Android device should work when connected to your workstation, but you may need to do a little work to set it up. If you have a Mac, you don't need to do anything except plug it in using the USB cable. Then, on the device itself, choose Settings ➤ Applications ➤ Development (though this may vary by phone and version) and enable USB debugging. On Linux, you probably need to create or modify this file: `/etc/udev/rules.d/51-android.rules`. We put a copy of this file on our web site with the project files; copy it to the proper directory, and modify the username and group values appropriately for your machine. Then, when you plug in an Android device, it will be recognized. Next, enable USB debugging on the device.

For Windows, you have to deal with USB drivers. Google supplies some with the Android packages, which are placed under the `usb_driver` subdirectory of the Android SDK directory. Other device vendors provide drivers for you, so look for them on their web sites. You can also visit the XDA forums, `forum.xda-developers.com`, where advice on sourcing and configuring drivers for a variety of phones and devices is discussed. When you have the drivers set up, enable USB debugging on the device, and you're ready.

Now that your device is connected to your workstation, when you try to launch an app, either it launches directly on the device or (if you have an emulator running or other devices attached) a window opens in which you choose which device or emulator to launch into. If not, try editing your Run Configuration to manually select the target.

Exploring the Structure of an Android Application

Although the size and complexity of Android applications can vary greatly, their structures are similar. Figure 1-13 shows the structure of the "Hello World!" app you just built.

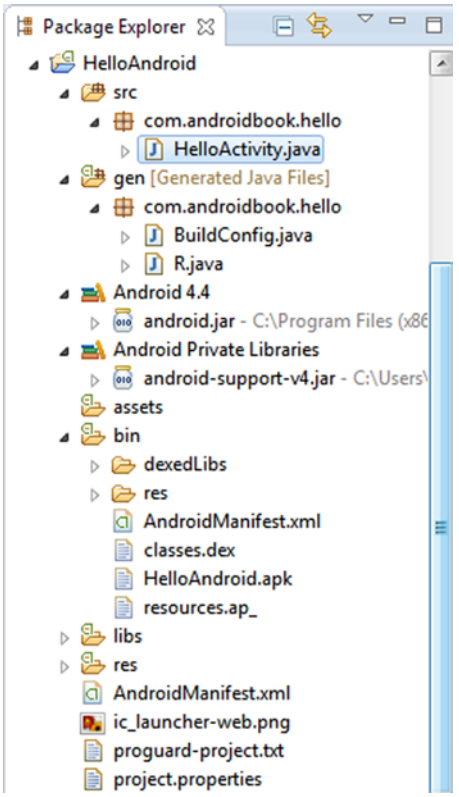

Figure 1-13. *The structure of the "Hello World!" application*

Android applications have some artifacts that are required and some that are optional. Table 1-1 summarizes the elements of an Android application.

Table 1-1. *The Artifacts of an Android Application*

Artifact	Description	Required?
`AndroidManifest.xml`	The Android application descriptor file. This file defines the activities, content providers, services, and intent receivers of the application. You can also use this file to declaratively define permissions required by the application, as well as instrumentation and testing options.	Yes
`src`	A folder containing all of the source code of the application.	Yes
`assets`	An arbitrary collection of folders and files.	No
`res`	A folder containing the resources of the application. This is the parent folder of `drawable`, `animator`, `layout`, `menu`, `values`, `xml`, and `raw`.	Yes

(*continued*)

Table 1-1. (*continued*)

Artifact	Description	Required?
`drawable`	A folder containing the images or image-descriptor files used by the application.	No
`animator`	A folder containing the XML-descriptor files that describe the animations used by the application.	No
`layout`	A folder containing views of the application.	No
`menu`	A folder containing XML-descriptor files for menus in the application.	No
`values`	A folder containing other resources used by the application. Examples of resources found in this folder include strings, arrays, styles, and colors.	No
`xml`	A folder containing additional XML files used by the application.	No
`raw`	A folder containing additional data—possibly non-XML data—that is required by the application.	No

As you can see from Table 1-1, an Android application is primarily made up of three mandatory pieces: the application descriptor, a collection of various resources, and the application's source code. If you put aside the `AndroidManifest.xml` file for a moment, you can view an Android app in this simple way: you have some business logic implemented in code, and everything else is a resource.

Android has also adopted the approach of defining views via markup in XML. You benefit from this approach because you don't have to hard-code your application's views; you can modify the look and feel of the application by editing the markup.

It is also worth noting a few constraints regarding resources. First, Android supports only a single-level list of files within the predefined folders under `res`. For example, there are some similarities between the `assets` folder and the `raw` folder under `res`. Both folders can contain raw files, but the files in `raw` are considered resources, and the files in `assets` are not. So the files in `raw` are localized, accessible through resource IDs, and so on. But the contents of the `assets` folder are considered general-purpose content to be used without resource constraints and support. Note that because the contents of the `assets` folder are not considered resources, you can put an arbitrary hierarchy of folders and files in this folder. (Chapter 3 talks a lot more about resources.)

Note You may have noticed that XML is used quite heavily with Android. You know that XML can be a bloated data format, so does it make sense to rely on XML when you know your target is a device with limited resources? It turns out that the XML you create during development is actually compiled down to binary using the Android Asset Packaging Tool (AAPT). Therefore, when your application is installed on a device, the files on the device are stored as binary. When the file is needed at runtime, the file is read in its binary form and is not transformed back into XML. This gives you the benefits of both worlds—you get to work with XML, and you don't have to worry about taking up valuable resources on the device.

Examining the Application Life Cycle

The life cycle of an Android application is strictly managed by the system, based on the user's needs, available resources, and so on. A user may want to launch a web browser, for example, but the system ultimately decides whether to start the application. Although the system is the ultimate manager, it adheres to some defined and logical guidelines to determine whether an application can be loaded, paused, or stopped. If the user is currently working with an activity, the system gives high priority to that application. Conversely, if an activity is not visible and the system determines that an application must be shut down to free up resources, it shuts down the lower-priority application.

The concept of application life cycle is logical, but a fundamental aspect of Android applications complicates matters. Specifically, the Android application architecture is component- and integration-oriented. This allows a rich user experience, seamless reuse, and easy application integration but creates a complex task for the application life-cycle manager.

Let's consider a typical scenario. A user is talking to someone on the phone and needs to open an e-mail message to answer a question. The user goes to the home screen, opens the mail application, opens the e-mail message, clicks a link in the e-mail, and answers the friend's question by reading a stock quote from a web page. This scenario requires four applications: the home application, a talk application, an e-mail application, and a browser application. As the user navigates from one application to the next, the experience is seamless. In the background, however, the system is saving and restoring application state. For instance, when the user clicks the link in the e-mail message, the system saves metadata on the running e-mail message activity before starting the browser-application activity to launch a URL. In fact, the system saves metadata on any activity before starting another so that it can come back to the activity (when the user backtracks, for example). If memory becomes an issue, the system has to shut down a process running an activity and resume it as necessary.

Android is sensitive to the life cycle of an application and its components. Therefore, you need to understand and handle life-cycle events in order to build a stable application. The processes running your Android application and its components go through various life-cycle events, and Android provides callbacks that you can implement to handle state changes. For starters, you should become familiar with the various life-cycle callbacks for an activity (see Listing 1-1).

Listing 1-1. Life-Cycle Methods of an Activity

```
protected void onCreate(Bundle savedInstanceState);
protected void onStart();
protected void onRestart();
protected void onResume();
protected void onPause();
protected void onStop();
protected void onDestroy();
```

Listing 1-1 shows the list of life-cycle methods that Android calls during the life of an activity. It's important to understand when each of the methods is called by the system in order to ensure that you implement a stable application. Note that you do not need to react to all of these methods. If you do, however, be sure to call the superclass versions as well. Figure 1-14 shows the transitions between states.

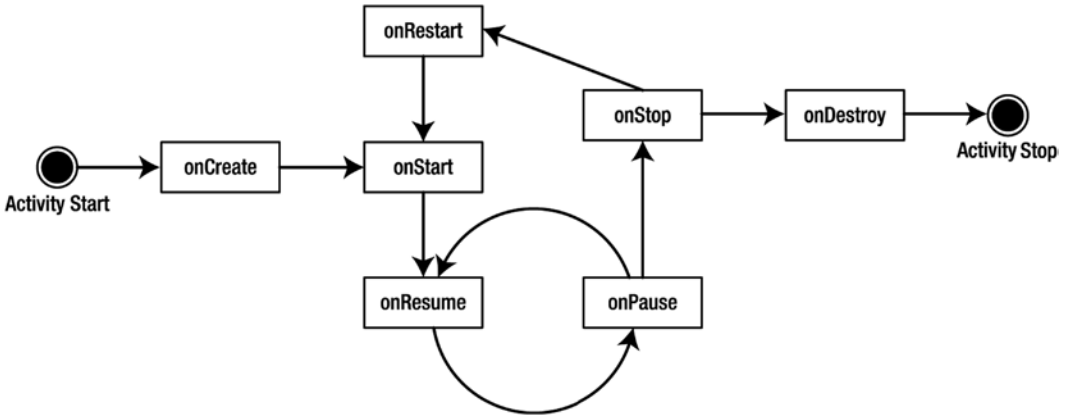

***Figure 1-14.** State transitions of an activity*

The system can start and stop your activities based on what else is happening. Android calls the `onCreate()` method when the activity is freshly created. `onCreate()` is always followed by a call to `onStart()`, but `onStart()` is not always preceded by a call to `onCreate()` because `onStart()` can be called if your application was stopped. When `onStart()` is called, your activity is not visible to the user, but it's about to be. `onResume()` is called after `onStart()`, just when the activity is in the foreground and accessible to the user. At this point, the user can interact with your activity.

When the user decides to move to another activity, the system calls your activity's `onPause()` method. From `onPause()`, you can expect either `onResume()` or `onStop()` to be called. `onResume()` is called, for example, if the user brings your activity back to the foreground. `onStop()` is called if your activity becomes invisible to the user. If your activity is brought back to the foreground after a call to `onStop()`, then `onRestart()` is called. If your activity sits on the activity stack but is not visible to the user, and the system decides to kill your activity, `onDestroy()` is called.