Chapter 4

Creating an Android App

In This Chapter

- Creating an elementary Android app
- Troubleshooting troublesome apps
- Testing an app on an emulator or a mobile device
- Dissecting an app

Chapter 3 describes the writing and running of a dirt-simple Java program. Like many Java programs, the one in Chapter 3 runs on a plain-old desktop or laptop computer. Behind the scenes, the code in Chapter 3 uses the powerful features of standard Oracle Java. But the two kinds of Java (standard Oracle Java for desktops and laptops, and Android's Java for mobile devices) are slightly different animals, for these reasons:

- **Standard Java uses the power and speed of desktop and laptop computers.**

  Android Java is streamlined to run on smaller devices with less memory.

- **Standard Java uses some features that aren't available in Android Java.**

  For example, the `javax.swing.JOptionPane.showMessageDialog` call in the program in Chapter 3 isn't available in Android Java.

- **Android Java uses some features that aren't available in standard Java.**

  For example, the `Activity` class in this chapter's program isn't available in standard Java.

- **Creating a basic Android app requires more steps than creating a basic standard Java app.**

This chapter covers the steps that are required in order to create a basic Android app, though the app doesn't do much. (In fact, you might argue that it does nothing.) But the example shows you how to create and run a new Android project.

Creating Your First Android App

A gadget typically comes supplied with a manual. The manual's first sentence is "Read all 37 safety warnings before attempting to install this product." Don't you love it? You can't get to the pertinent material without wading through the preliminaries.

Well, nothing in this chapter can set your house on fire or even break your electronic device. But before you follow this chapter's instructions, you need a bunch of software on your development computer. To make sure you have this software, and that it's properly configured, see Chapter 2. (Do not pass Go; do not collect $200.)

When at last you have all the software you need, you're ready to start Eclipse and create a real, live Android app.

Creating an Android project

To create your first Android application, follow these steps:

1. **Launch Eclipse.**

   For details on launching Eclipse, see Chapter 2.

2. **From the main menu in Eclipse, choose File⇨New⇨Android Application Project.**

   As a result, Eclipse fires up its New Android Application dialog box, as shown in Figure 4-1.

3. **In the Application Name field, type a name for the app.**

   In Figure 4-1, I type the boring words `My First Android App`. Ordinary folks such as Joe and Jane User, however, will see this name under the app's icon on the Android launcher screen. If you're planning to market your app, make the name short, sweet, and descriptive. You can even include blank spaces in the name.

   The next several steps involve lots of clicking, but you primarily accept the default settings.

4. **(Optional) In the Project Name and Package Name fields, change the name of the project and the name of the Java package containing the project.**

   Eclipse automatically fills in the Project Name and Package Name fields (guided by whatever text you type in the Application Name field). In

Figure 4-1, Eclipse creates the project name `MyFirstAndroidApp` and the package name `com.example.myfirstandroidapp`. Eclipse uses the project name to label this app's branch in the Package Explorer tree.

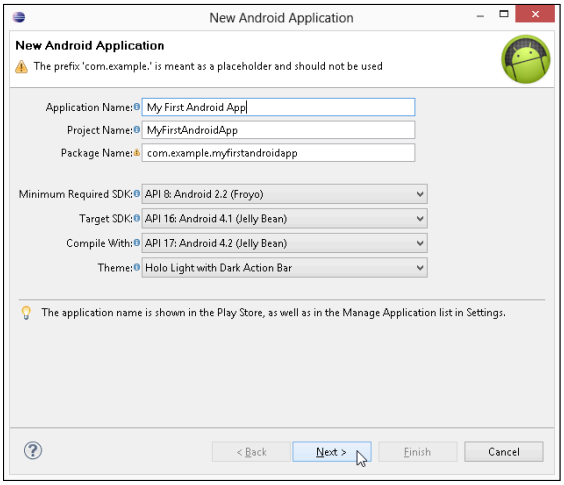

Figure 4-1: The first New Android Application dialog box.

For practice apps, you can cheat by using the package name that Eclipse creates. But if you plan to publish an app, give the app its own package name, using the rules described in Chapter 3.

In Android, a package name belongs to only one app. You can put the first app in the package `org.allyourcode.firstapp` and put the second app in the package `org.allyourcode.secondapp`. But you can't put more than one app in an `org.allyourcode.mystuff` package.

For the lowdown on Java packages and package names, see Chapter 5.

5. **(Optional) Choose values from the drop-down boxes in the dialog box.**

   To find out what you're promising when you select Minimum Required SDK API 8 and Target SDK API 16, see the nearby sidebar, "Using Android versions."

   In Figure 4-1, I accept the defaults offered to me — API 8, API 16, and API 17. You can select any values from the drop-down boxes as long as you've created an Android Virtual Device (AVD) that can run the target's projects. (For example, an Android 2.3.3 AVD can run projects targeted to earlier versions of Android, such as Android 2.3.1, Android 2.2, and Android 1.6. The project target doesn't have to be an exact match with an existing AVD.)

Using Android versions

Android has a few different uses for version numbers. For example, in Figure 4-1, the minimum required SDK is API 8 and the target SDK is API 16. What's the difference?

You design an Android app to run on a range of API versions. You can think informally of the minimum SDK version as the lowest version in the range, and the target version as the highest. So if you select API 8 as the minimum SDK and select API 16 as the target, you design an app to run on API levels 8 through 16.

But the lowest-to-highest-version idea needs refining. The official Android documentation reports that " . . . new versions of the platform are fully backward-compatible." So an app that runs correctly on API 8 should run correctly on all versions higher than API 8. (I write "*should* run correctly" because, in practice, full backward compatibility is difficult to achieve. Anyway, if the Android team is willing to promise full backward compatibility, I'm willing to take my chances.)

The *target version* (it's API 16 in Figure 4-1) is the version for which you test the app. When you run this chapter's example, Eclipse opens an emulator with API 16 or higher installed. (For example, if you've created an AVD whose API is level 17 but you have no AVD whose API is level 16, Eclipse opens the emulator with API 17.) To the extent that your app passes your testing, the app runs correctly on devices that run API 16 (also known as Android 4.1). What about devices that run other versions of Android? This list provides an explanation:

- *The app's target version is API 16, but the app uses only features that are available in API 8 and earlier:* In that case, you can safely enter the number 8 in the Minimum Required SDK field in Eclipse.
- *The app uses some features available only in API 16 and later, but the app contains workarounds for devices that run API 8:* (The app's code can detect a device's Android version and contains alternative code for different versions.) In that case, you can safely put the number 8 in the Minimum Required SDK field.
- *The app's target version is API 16:* In 2019, someone installs your app on a device running API 99 (code-named Zucchini Bread). Because of backward compatibility, the app runs awkwardly but correctly on the API 99 device. Then the app's target version (API 16) isn't truly the upper limit.

When you select a target version and a minimum SDK version, Android stores these numbers in the project's `AndroidManifest.xml` file. You can see the `AndroidManifest.xml` file in the project's tree in the Package Explorer in Eclipse.

If you mistakenly select a target for which you have no AVD, Eclipse hollers at you when you try to run the project. (Though Eclipse hollers, it also offers to help you create the necessary AVD, so everything turns out just fine.)

For help with creating an AVD, see Chapter 2.

6. **Click Next.**

   As a result, the New Android Application dialog box reappears. (See Figure 4-2 — okay, originality in naming dialog boxes may not be Eclipse's strong suit.)

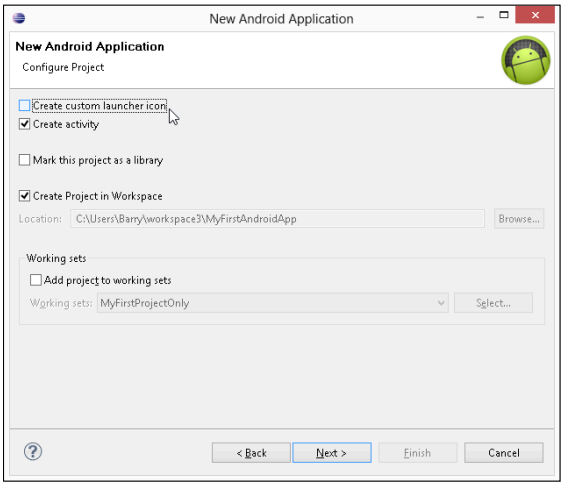

Figure 4-2: The second New Android Application dialog box.

7. **(Optional) Tweak the settings in the latest incarnation of the New Android Application dialog box.**

   For a practice app, I recommend deselecting the Create Custom Launcher Icon check box and leaving untouched the other settings in this New Android Application dialog box. In particular, keep the Create Activity option selected.

8. **Click Next.**

   As a result, the Create Activity dialog box appears, as shown in Figure 4-3.

   For the truth about activities in Android, see Chapter 5.

9. **Click Next again. (In other words, accept the defaults in the Create Activity dialog box.)**

   The next box in the sequence is the New Blank Activity dialog box, as shown in Figure 4-4.

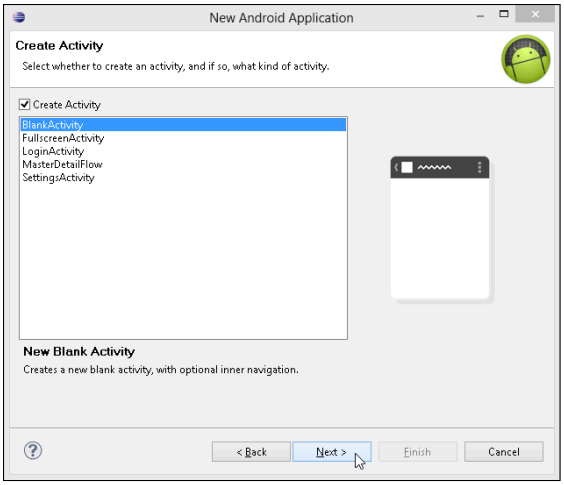

Figure 4-3: Creating a new activity.

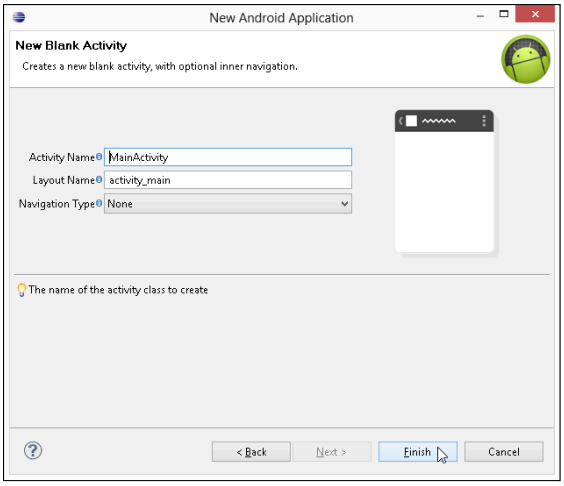

Figure 4-4: Creating a blank activity.

10. **Click Finish. (That is, accept the defaults.)**

    As a result, the New Blank Activity dialog box closes, and the Eclipse workbench moves to the foreground. The Package Explorer tree in Eclipse has a new branch. The branch's label is the name of the new project, as shown in Figure 4-5.

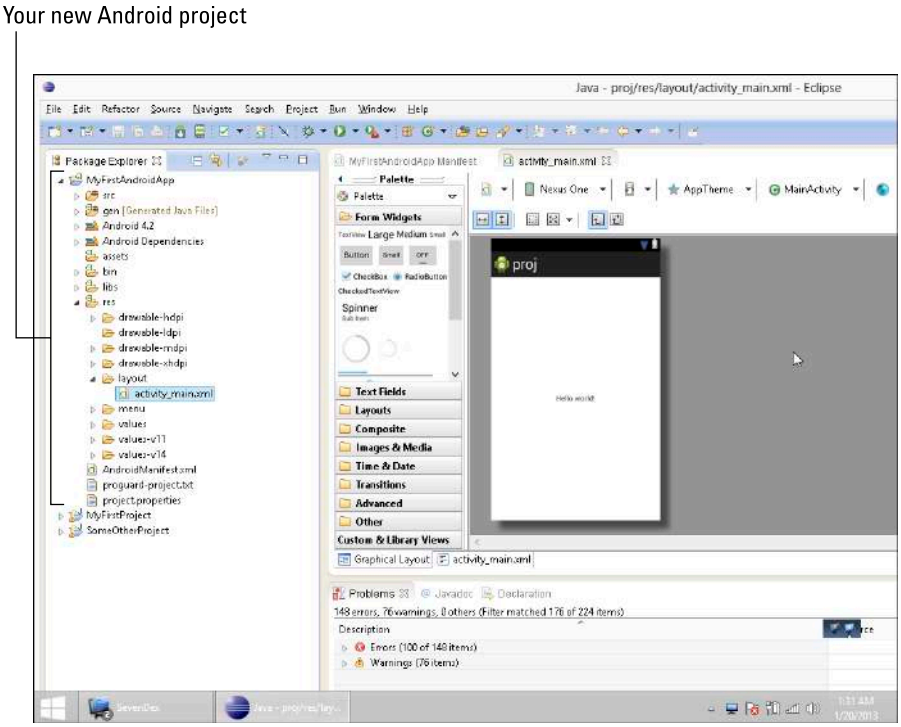

Figure 4-5: A new `MyFirst Android App` branch.

Congratulations — you've created an Android application.

Running your project

To kick your new app's tires and take your app around the block, do the following:

1. **Select the app's branch in the Package Explorer in Eclipse.**

   (Refer to Figure 4-5.)

2. **In the main menu, choose Run➪Run As➪Android Application.**

   As a result, the Console view displays several lines of text. Among them, you might find the phrases `Launching a new emulator`, `Waiting for HOME`, and (as shown in Figure 4-6) my personal favorite, `Success!`

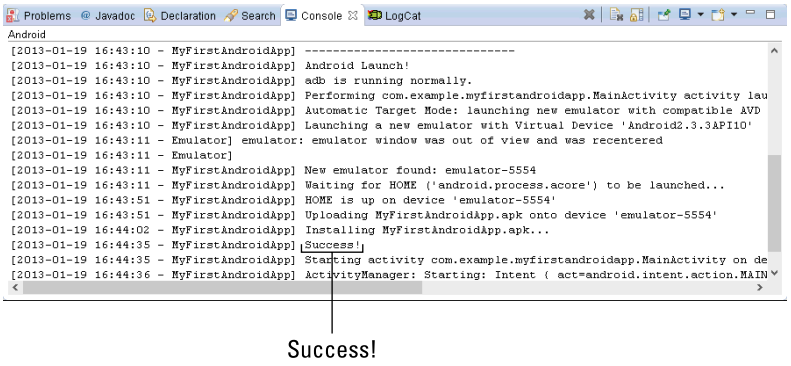

Figure 4-6: The Console view during the successful launch of an app.

If you don't see the Console view, you have to coax it out of hiding. For details, see Chapter 3.

In the lingo of general app development, a *console* is a text-only window that displays the output of a running program. A console might also accept commands from the user (in this case, the app developer). A single Android run might create several consoles at a time, so the Console view in Eclipse can display several consoles at a time. If the material you see in the Console view in Eclipse is nothing like the text shown in Figure 4-6, the Console view may be displaying the wrong console. To fix this problem, look for a button showing a picture of a computer terminal in the upper-right corner of the Console view, as shown in Figure 4-7. Click the arrow to the right of the button. In the resulting drop-down list, choose Android.

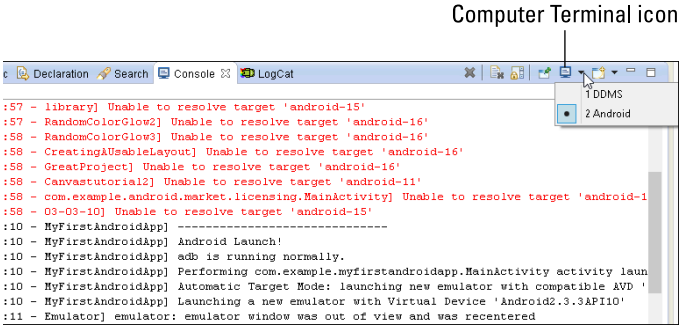

Figure 4-7: Choosing a console.

3. **Wait for the Android emulator to display the Device Locked screen, a Home screen, or an app's screen.**

   First you see the word ANDROID as though it's part of a scene from *The Matrix,* as shown in Figure 4-8. Then you see the word ANDROID in

shimmering, silvery letters, as shown in Figure 4-9. Finally, you see the Device Locked screen, a Home screen, or an app's screen, as shown in Figure 4-10.

Figure 4-8: The emulator starts running.

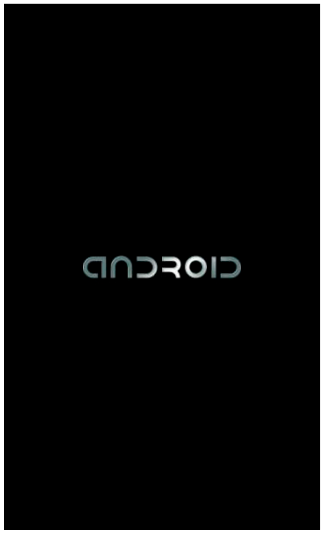

Figure 4-9: Android starts running on the emulator.

4. **I can't overemphasize this point: Wait for the Android emulator to display the Device Locked screen, a Home screen, or an app's screen.**

   The Android emulator takes a long time to start. For example, on my 2 GHz processor with 4GB of RAM, the emulator takes a few minutes to mimic a fully booted Android device. You need lots of patience when you deal with the emulator.

5. **Keep waiting.**

   While you're waiting, you can search the web for the phrase *Android emulator speed up*. Lots of people have posted advice, workarounds, and other hints.

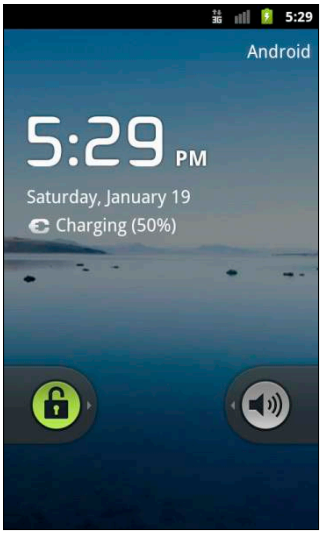

Figure 4-10: The Device Locked screen in Android 2.3.3 appears.

Oh! I see that the emulator is finally displaying the Device Locked screen. It's time to proceed. . . .

6. **If the emulator displays the Device Locked screen, do whatever you normally do to unlock an Android device.**

   Usually, you unlock the device by sliding something from one part of the screen to another.

7. **See the app on the emulator's screen.**

   Figure 4-11 shows the running of the Hello World app in Android. (The screen even displays `Hello World!`) Eclipse creates this tiny app when you create a new Android project.

   The Hello World app in Android has no widgets for the user to push, and the app doesn't do anything interesting. But the appearance of an app on the Android screen is a good start. Following the steps in this chapter, you can start creating many exciting apps.

Don't close an Android emulator unless you know that you won't be using it for a while. The emulator is fairly reliable after it gets going. (It's sluggish, but reliable.) While the emulator runs, you can modify the Android code and choose Run➪Run As➪Android Application again. When you do, Android reinstalls the app on the running emulator. The process isn't speedy, but you don't have to wait for the emulator to start. (Actually, if you run a different app — an app whose minimum required SDK is higher than the running emulator can handle — Android fires up a second emulator. But in many developer scenarios, jumping between emulators is the exception rather than the rule.)

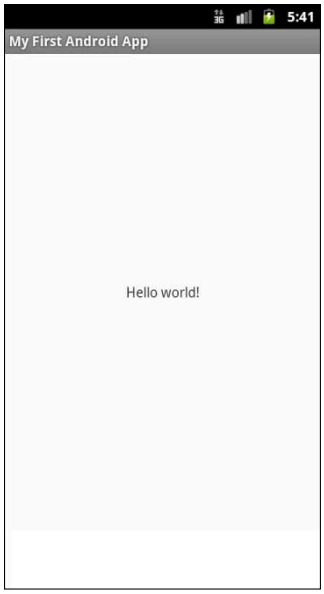

Figure 4-11: The Hello World app in action.

What if . . .

You try to run your first Android app. If your effort stalls, don't despair. This section has some troubleshooting tips.

Error message: R cannot be resolved

Every Android app has an `R.java` file. The Android development tools generate this file automatically, so normally you don't have to worry about `R.java`. Occasionally, the file takes longer than average to be generated. In this case, Eclipse finds references to the `R` class in the rest of the project's code and complains that the project has no `R` class. My advice is to wait.

If one minute of waiting doesn't bring good results, follow these steps to double-check the project settings:

1. **Highlight the project in the Package Explorer in Eclipse.**
2. **From the main menu, choose Project.**

   A list of submenu items appears.
3. **Look for a check mark next to the Build Automatically menu subitem.**

4. **If you don't see a check mark, select the Build Automatically subitem to add one.**

   With any luck, the `R.java` file appears almost immediately.

If the project is set to Build Automatically and you still don't have an `R.java` file, try these steps:

1. **Highlight the project in the Package Explorer.**
2. **From the main menu, choose Project.**

   A list of submenu items appears.
3. **In the Clean dialog box in Eclipse, select the project that's giving you trouble along with the Clean Projects Selected Below radio button.**
4. **Click OK.**

Cleaning the project should fix the problem. But if the problem persists, close Eclipse and then restart it. (Eclipse occasionally becomes "confused" and has to be restarted.)

After copying Java code from one Android project to another, you might see the annoying message *Import cannot be resolved* near the top of the program. If so, you might have inadvertently told one project to fetch material from another project's `R.java` file. If the offending line of code is `import somethingOrOther.R`, try deleting that line of code. Who knows? Your deletion might just fix the problem.

Error message: No compatible targets were found

When you see this message, it probably means that you haven't created an Android Virtual Device (AVD) capable of running your project. If Eclipse offers to help you create a new AVD, accept it. Otherwise, choose Window➪Android Virtual Device Manager to create a new AVD.

For information about Android Virtual Devices, see Chapter 2.

The emulator stalls during start-up

After five minutes or so, you don't see the Device Locked screen or the Android Home screen. Try these solutions:

- **Close the emulator and launch the application again. (Or lather, rinse, repeat.)**

  Sometimes, the second or third time's a charm. On rare occasions, my first three attempts fail but my fourth attempt succeeds.

- **Start the emulator independently.**

  That is, start the emulator without trying to run an Android project. Follow these four steps:

  a. ***From the Eclipse main menu, choose Window➪Android Virtual Device Manager.***

     The Android Virtual Device Manager window opens. It contains a list of AVDs that you've already created.

     For help creating an AVD, see Chapter 2.

  b. ***In the Android Virtual Device Manager, select the AVD that you want to start.***

  c. ***On the right side of the Android Virtual Device Manager, click Start.***

     As a result, Eclipse displays the Launch Options dialog box.

  d. ***In the Launch Options dialog box, click Launch.***

     In other words, accept the default options and fire up the emulator.

  When, at last, you see the new emulator's Device Locked screen or Home screen, follow Steps 1, 2, 6, and 7 in the earlier section "Running your project."

  If you try the tricks in this section but the stubborn Android emulator still doesn't start, visit this book's website (`http://allmycode.com/Java4Android`) for more strategies to try.

- **Run the app on a phone, a tablet, or another real Android device.**

  Testing a brand-new app on a real device makes me queasy. But the Android sandbox is fairly safe for apps to play in. Besides, apps load quickly and easily on phones and tablets.

  For instructions on installing apps to Android devices, see the section "Testing Apps on a Real Device," later in this chapter.

Error message: The user data image is used by another emulator

If you see this message, a tangle involving the emulator prevents Android from doing its job. First try closing and restarting the emulator.

If a simple restart doesn't work, try these steps:

1. **Close the emulator.**
2. **From the main menu in Eclipse, choose Window➪Android Virtual Device Manager.**

   To read about the Android Virtual Device Manager, see Chapter 2.

3. **In the list of virtual devices, select an AVD that's appropriate to the project and click Start.**
4. **In the resulting Launch Options dialog box, select the Wipe User Data check box and click Launch.**

   As a result, Eclipse launches a new copy of the emulator — this time, with a clean slate.

If you follow the steps in this section but you still see the message `User data image is used by another emulator`, visit this book's website (`http://allmycode.com/Java4Android`) for more help with this problem.

Error message: Unknown virtual device name

Android looks for AVDs in the home directory's `.android/avd` subdirectory, and occasionally the search goes awry. For example, one of my Windows computers lists my home directory on an `i` drive. My AVDs are in `i:\Users\barry\.android\avd`. But Android ignores the computer's home directory advice and instead looks in `c:\Users\Barry`. When Android doesn't find any AVDs, it complains.

You can devise fancy solutions to this problem by using either junctions or symbolic links. But solutions of this kind require special handling of their own. To keep it simple, I copy the contents of my `i:\Users\barry\.android` directory to `c:\Users\barry\.android` to fix the problem.

Error message: INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATE

This error message indicates that an app you previously installed conflicts with the app you're trying to install. So, on the emulator screen, navigate to the list of installed applications (which is usually an option on the Settings screen). In the list of applications, delete any apps that you installed previously.

Occasionally, you might have trouble finding previously installed apps from the Settings➪Applications menus in the emulator. If you do, visit this book's website (`http://allmycode.com/Java4Android`) for a geeky workaround solution.

The app starts, but the emulator displays the Force Close or Wait dialog box

The formal name of the Force Close or Wait dialog box is Application Not Responding (ANR). Android displays the ANR dialog box whenever an app takes too long to do whatever it's supposed to do. When the app runs on a real device (a phone or a tablet), the app shouldn't make Android display the ANR dialog box.

But on a slow emulator, seeing a few Force Close or Wait messages is par for the course. When I see the ANR dialog box in an emulator, I usually select Wait. Within about ten seconds, the dialog box disappears and the app continues to run.

Changes to your app don't appear in the emulator

Your app runs and you want to make a few improvements. So, with the emulator still running, you modify the app's code. But after choosing Run➪Run As➪Android Application, the app's behavior in the emulator remains unchanged.

When this happens, something is clogged up. Close and restart the emulator. If necessary, use the Wipe User Data trick that I describe in the earlier section "Error message: The user data image is used by another emulator."

The emulator's screen is too big

Sometimes, the development computer's screen resolution isn't high enough. (Maybe your eyesight isn't what it used to be.) This symptom isn't a deal breaker, but if you can't see the emulator's lower buttons, you can't easily test the app. You can change the development computer's screen resolution, though adjusting the emulator window is less invasive.

To change the emulator window size, follow these steps:

1. **Close the emulator.**
2. **From the Eclipse main menu, choose Window➪Android Virtual Device Manager.**
3. **In the list of virtual devices, select an AVD that's appropriate to the project and click Start.**
4. **In the resulting Launch Options dialog box, select the Scale Display to Real Size check box.**
5. **Lower the value in the Screen Size field.**

   As you change the Screen Size value, the value in the Scale field changes automatically. The smaller the Scale value, the smaller the emulator appears on the development computer's screen.
6. **Click Launch.**

   As a result, Eclipse launches a new copy of the emulator — this time, with a smaller emulator window.

Testing Apps on a Real Device

You can bypass emulators and test apps on a phone, a tablet, or maybe an Android-enabled trash compactor. To do so, you have to prepare the device, prepare the development computer, and then hook the two together. This section describes the process.

To test an app on a real Android device, follow these steps:

1. **On the Android device, turn on USB debugging.**

   Various Android versions have their own ways of enabling (or disabling) USB debugging. You can poke around for the debugging option on your own device or visit this site for the procedures on some representative Android versions:

   ```
   www.teamandroid.com/2012/06/25/how-to-enable-usb-
           debugging-in-android-phones
   ```

   On my device, I keep USB debugging on all the time. But if you're nervous about security, turn off USB debugging when you aren't using the device to develop apps.

2. **In your project's branch of the Package Explorer, double-click the** `AndroidManifest.xml` **file.**

   Eclipse offers several ways to examine and edit this file.

3. **At the bottom of the Eclipse editor, click the Application tab.**

   Eclipse displays a form like the one shown in Figure 4-12.

4. **In the Debuggable drop-down list, choose True. (Refer to Figure 4-12.)**

   When Debuggable is set to True, Android tools can monitor the run of the app.

   The ability to debug is the ability to hack. Debugging also slows down an app. Never distribute an app to the public with Debuggable set to True.

5. **Choose File➪Save to store the new** `AndroidManifest.xml` **file.**

6. **Set up the development computer to communicate with the device.**

   - *On Windows:* Visit `http://developer.android.com/sdk/oem-usb.html` to download the device's Windows USB driver. Install the driver on the development computer.
   - *On a Mac:* `/*` Do nothing. It just works. `*/`

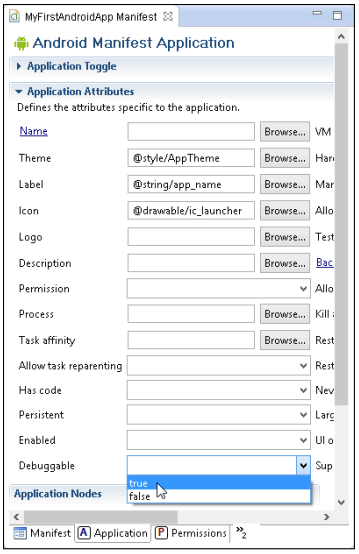

Figure 4-12: The Application tab of a project's `Android Manfest .xml` file.

7. **Using a USB cable, connect the device to the development computer.**

   For ways to verify that the device is connected to the development computer, visit this book's website at `http://allmycode.com/Java4Android`.

8. **In Eclipse, run the project.**

   A connected device trumps a running emulator. So, if the Android version on the device can handle the project's minimum SDK version, choosing Run➪Run As➪Android Application installs the app on the connected device.

Eventually, you'll disconnect the device from the development computer. If you're a Windows user, you may dread reading `Windows can't stop your device because a program is still using it`. To disconnect the device safely, do the following:

1. **Open the Command Prompt window.**

   On Windows 7 or earlier: **Choose Start➪All Programs➪Accessories➪Command Prompt.**

   On Windows 8: **First press Windows+Q. Then type** Command Prompt **and press Enter.**

2. **In the Command Prompt window, navigate to the** `ANDROID_HOME/platform-tools` **directory.**

For example, if the ANDROID_HOME directory is

```
C:\Users\yourName\adt-bundle-windows-x86_64\sdk
```

type this command:

```
cd C:\Users\yourName\adt-bundle-windows-x86_64\sdk\platform-tools
```

3. **In the Command Prompt window, type** adb kill-server **and then press Enter.**

   The `adb kill-server` command stops communication between the development computer and any Android devices, real or virtual. In particular,

   - The development computer no longer talks to the device at the end of the USB cable.
   - The development computer no longer talks to any emulators it's running.

   After issuing the `adb kill-server` command, you see the friendly `Safe to Remove Hardware` message.

4. **Unplug the Android device from the development computer.**

   After unplugging the device, you might want to reestablish communication between the development computer and any emulators you're running. If so, follow Step 5.

5. **In the Command Prompt window, type** adb start-server **and then press Enter.**

Examining an Android App

In Figure 4-13, the Package Explorer in Eclipse shows the structure of a newly created Android project. Each branch of the tree represents a file or a folder, and if you expand all branches of the tree, you see even more files and folders. Why so many files and folders in an Android project? This section provides answers.

The src directory

The `src` directory contains the project's Java source code. Files in this directory have names such as `MainActivity.java`, `MyService.java`, `DatabaseHelper.java`, and `MoreStuff.java`.

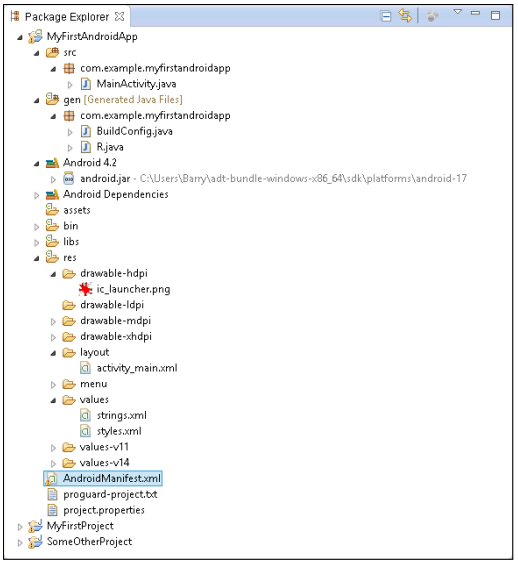

Figure 4-13: The Package Explorer displays an Android app.

You can cram hundreds of Java files into a project's `src` directory. But when you create a new project, Android typically creates only one file for you. Earlier in this chapter, I accepted the default name `MainActivity` so that Android creates a file named `MainActivity.java`. (Refer to Figure 4-4.)

An Android activity is one "screenful" of components. For more information about Android activities, see Chapter 5.

Most of the material in this book is about files in the `src` directory. In this chapter, I focus on the other directories.

The res directory

A project's `res` directory contains resources for use by the Android application. In Figure 4-13, you see that `res` has a bunch of subdirectories: four `drawable` directories, a `layout` directory, a `menu` directory, and three `values` directories.

The drawable subdirectories

The `drawable` directories contain images, shapes, and other elements.

Each `drawable` directory applies to certain screen resolutions. For example, in the name `drawable-hdpi`, the letters `hdpi` stand for *h*igh number of *d*ots *p*er *i*nch. Files in the `drawable-hdpi` directory apply to devices whose resolutions are (roughly) between 180 and 280 dots per inch.

For more information about Android screen resolutions, visit `http://developer.android.com/guide/practices/screens_support.html`.

In Figure 4-13, the `drawable-hdpi` directory contains one file named `ic_launcher.png`. This file describes the image that appears on the app's icon on the Android launcher screen.

The values subdirectory

An app's `res/values` directory contains a file named `strings.xml`. (Refer to Figure 4-13.) Listing 4-1 shows the code in a simple `strings.xml` file.

Listing 4-1: A Small strings.xml File

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

    <string name="app_name">My First Android App</string>
    <string name="hello_world">Hello world!</string>
    <string name="menu_settings">Settings</string>

</resources>
```

The code in Listing 4-1 is XML code. For information about XML code, see the "All about XML files" sidebar, later in this chapter.

In the `strings.xml` file, you collect all the words, phrases, and sentences that the app's user might see. You lump together phrases such as *Hello world!* and *My First Android App* so that someone can translate them all into different languages. With all those phrases collected in the `strings.xml` file, a translator doesn't have to poke around to find phrases in the Java code. (Poking around in the code in any real programming language can be dangerous because program code is intricate, and it can be brittle. Believe me: If I were a translator, I'd much rather translate the phrases in a `strings.xml` file.)

Listing 4-1 describes a `"hello_world"` string containing the characters *Hello World!* So in the app's Java code, you refer to the words *Hello world!* by typing `R.string.hello_world`. To refer to the words *Hello world!* in another XML file (such as the one in Listing 4-2), you type `"@string/hello_world"`. Either way, the text `R.string.hello_world` or the text `"@string/hello_world"` stands for the words *Hello world!* in Listing 4-1.

The use of `strings.xml` files helps with *localization,* which, in the tech world, is what you do to adapt an app to a culture's local language and customs. To localize the app for French-speaking users, for example, you create an additional folder named `values-fr`. You add this folder to the tree shown in Figure 4-13. Inside the `values-fr` folder, you create a second `strings.xml` file, and the new `strings.xml` file contains a line such as this one:

```
<string name="hello_world">Bonjour tout le monde!</string>
```

For Romanian, you create a `values-ro` directory, containing a `strings.xml` file with this line:

```
<string name="hello_world">Salut lume!</string>
```

When Android sees either `R.string.hello_world` or `"@string/hello_world"` in the code, Android determines the user's country of origin and automatically displays the correct translation. This localization happens with no further effort on your part.

The layout subdirectory

The `layout` directory contains descriptions of the activities' screens.

A minimal app's `res/layout` directory contains an XML file describing an activity's screen. (Refer to the `activity_main.xml` branch in Figure 4-13.) Listing 4-2 shows the code in the simple `activity_main.xml` file.

Listing 4-2: A Small Layout File

```
<RelativeLayout xmlns:android=
        "http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world" />

</RelativeLayout>
```

The code in Listing 4-2 specifies that the layout of the app's activity is a `RelativeLayout` (whatever that means) and, centered inside the `RelativeLayout`, you have a `TextView`. This `TextView` thingy is a little label containing the words *Hello world!* (Refer to Figure 4-11.)

All about XML files

Every Android app consists of some Java code, some XML documents, and some other information. (The acronym *XML* stands for eXtensible Markup Language.) You might already be familiar with HTML documents — the bread and butter of the World Wide Web.

Listings 4-1 and 4-2 contain XML documents. Like an HTML document, every XML document consists of tags (angle-bracketed descriptions of various pieces of information). But unlike an HTML document, an XML document doesn't necessarily describe a displayable page.

Here are some facts about XML code:

- **A *tag* consists of text surrounded by angle brackets.**

  For example, the code in Listing 4-2 consists of three tags: The first tag is the `<RelativeLayout ... >` tag, the second tag is the `<TextView ... />` tag, and the third tag is the `</RelativeLayout>` tag.

- **An XML document may have three different kinds of tags: start tags, empty element tags, and end tags.**

  A *start tag* begins with an open angle bracket and a name. A start tag's last character is a closing angle bracket.

  The first tag in Listing 4-2 (the `<RelativeLayout ... >` tag on lines 1–6) is a start tag. Its name is `RelativeLayout`.

  An *empty element tag* begins with an open angle bracket followed by a name. An empty element tag's last two characters are a forward slash followed by a closing angle bracket.

  The second tag in Listing 4-2 (the `<TextView ... />` tag on lines 8–13 in the listing) is an empty element tag. Its name is `TextView`.

  An *end tag* begins with an open angle bracket followed by a forward slash and a name. An end tag's last character is a closing angle bracket.

  The third tag in Listing 4-2 (the `</RelativeLayout>` tag on the last line of the listing) is an end tag. Its name is `RelativeLayout`.

- **An XML *element* either has both a start tag and an end tag, or it has an empty element tag.**

  In Listing 4-2, the document's `RelativeLayout` element has both a start tag and an end tag. (Both the start and end tags have the same name, `RelativeLayout`, so the name of the entire element is `RelativeLayout`.)

  In Listing 4-2, the document's `TextView` element has only one tag: an empty element tag.

- **Elements are either nested inside one another or have no overlap.**

  For example, in the following code, a `TableLayout` element contains two `TableRow` elements:

```
<TableLayout xmlns:android=
      "http://schemas.
   android.com/apk/res/
   android"
      android:layout_
   width="fill_parent"
      android:layout_
   height="fill_parent" >

  <TableRow>

    <TextView
        android:layout_
   width="wrap_content"
        android:layout_
   height="wrap_content"
```

```
        android:text="@
    string/name" />

  </TableRow>

  <TableRow>

    <TextView
        android:layout_
    width="wrap_content"
        android:layout_
    height="wrap_content"
        android:text="@
    string/address" />

  </TableRow>

</TableLayout>
```

The preceding code works because the first `TableRow` ends before the second `TableRow` begins. But the following XML code is illegal:

```
<!-- The following code isn't
   legal XML code. -->
<TableRow>

    <TextView
        android:layout_
   width="wrap_content"
        android:layout_
   height="wrap_content"
        android:text="@
   string/name" />
<TableRow>

</TableRow>

    <TextView
        android:layout_
   width="wrap_content"
        android:layout_
   height="wrap_content"
        android:text="@
   string/address" />
</TableRow>
```

With two start tags followed by two end tags, this new XML code doesn't pass muster.

- **Each XML document contains a *root element* — one element in which all other elements are nested.**

  In Listing 4-2, the root element is the `RelativeLayout` element. The listing's only other element (the `TextView` element) is nested inside that `RelativeLayout` element.

- **Different XML documents use different element names.**

  In every HTML document, the `<br />` element stands for *line break*. But in XML, the names `RelativeLayout` and `TextView` are particular to Android layout documents. And the names `portfolio` and `trade` are particular to financial product XML (`FpML`) documents. The names `prompt` and `phoneme` are peculiar to voice XML (`VoiceXML`). Each kind of document has its own list of element names.

- **The text in an XML document is case-sensitive.**

  For example, if you change `RelativeLayout` to `relativelayout` in Listing 4-2, the app won't run.

- **Start tags and empty element tags may contain attributes.**

  An *attribute* is a name-value pair. Each attribute has the form *name="value"*. The quotation marks around the *value* are required.

  In Listing 4-2, the start tag (`RelativeLayout`) has five attributes, and the empty element tag (`TextView`) has five of its own attributes. For example, in the `TextView` empty element tag, the text `android:layout_width="wrap_content"` is the first attribute. This attribute has the name `android:layout_width` and the value `"wrap_content"`.

(continued)

(continued)

- **A non-empty XML element may contain *content*.**

  For example, in the element `<string name="hello_world">Hello world!</string>` in Listing 4-1, the content `Hello world!` is sandwiched between the start tag (`<string name="hello_world">`) and the end tag (`</string>`).

The gen directory

The directory name `gen` stands for *gen*erated. The `gen` directory contains `R.java`. Listing 4-3 shows that part of the `R.java` file generated for you when you create a brand-new project.

Listing 4-3: Don't Even Look at This File

```
/* AUTO-GENERATED FILE.  DO NOT MODIFY.
 *
 * This class was automatically generated by the
 * aapt tool from the resource data it found.  It
 * should not be modified by hand.
 */

package com.example.myfirstandroidapp;

public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int ic_launcher=0x7f020000;
    }
    public static final class id {
        public static final int menu_settings=0x7f070000;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int activity_main=0x7f060000;
    }
    public static final class string {
        public static final int app_name=0x7f040000;
        public static final int hello_world=0x7f040001;
        public static final int menu_settings=0x7f040002;
    }
    // ... (There's more!)
```

The values in `R.java` are the jumping-off points for the resource management mechanism in Android. Android uses these numbers for quick and easy loading of the items you store in the `res` directory.

You can't make changes to the `R.java` file. Long after the creation of a project, Android continues to monitor (and, if necessary, update) the contents of the `R.java` file. If you delete `R.java`, Android re-creates the file. If you edit `R.java`, Android undoes the edit. If you answer Yes in the dialog box named Do You Really Want to Edit This File?, Eclipse accepts the change — but immediately afterward, Android clobbers your change.

The Android 4.2 branch

The tree shown in Figure 4-13 has an `Android 4.2` branch, but it isn't a directory on the computer's file system. In the Package Explorer view, the `Android 4.2` branch (or `Android 3.0` branch or `Android` *`whatever`* branch) reminds you that the project includes prewritten Android code (the Android API).

A `.jar` file is a compressed archive containing a useful bunch of Java classes. In fact, a `.jar` file is a `.zip` archive. You can open any `.jar` file by using WinZip or StuffIt Expander or the operating system's built-in unzipping utility. (You may or may not have to change the filename from *`whatever`*`.jar` to *`whatever`*`.zip`.) Anyway, an `android.jar` file contains prewritten Android code (the Android API) for a particular version of Android. In Figure 4-13, a Package Explorer branch reminds you that your project contains a reference to another location on the hard drive (to one containing the `.jar` file for Android 4.2).

R.java and the legend of the two vaudevillians

According to legend, two friends named Herkimer and Jake once worked together for 50 years as a comedy team in vaudeville. Year after year, they practiced and refined their act, adding a new joke here and removing an old joke there. As time went on, they adopted a kind of shorthand to refer to the jokes in their act. "Let's move Joke Number 35 to the end of the first song," said Herkimer. And Jake responded, "I'd rather do Joke Number 119 when the song ends."

Eventually, both Herkimer and Jake retired to an old-age home. Day after day, they sat side by side in the TV room, staring at reruns of Milton Berle's show and *The Ed Sullivan Show*. Occasionally, something on the screen would remind Herkimer of one the team's old jokes. "Fifty-one," Herkimer would call out. And upon hearing this number, Jake would start laughing hysterically.

(continued)

(continued)

Many elements of the code in an Android app are numbered. For example, an item on the screen can be in one of three states: 0, 4, or 8. To help you (the developer) remember what the numbers mean, the creators of Android provide synonyms for each number. So rather than write `0` in your Java code, you can write `View.VISIBLE`. An item in this state is in plain sight on the user's screen. On the other hand, an item in state 4 (with the synonym `View.INVISIBLE`) occupies space on the screen but doesn't light up any pixels. The user doesn't see this item, but its spooky presence might force other items to move one way or another. Finally, an item in state 8 (with the synonym `View.GONE`) has no presence on the screen. This item might have once appeared in the center of the screen, and it might later appear again on the screen. But now, in the `View.GONE` state, this item has no influence on the layout of the screen.

When dealing with state numbers, and with other code numbers, the creators of Java use hexadecimal notation. In Java, numbers starting with `0x` are hexadecimal (base 16) numbers. For example, the number `0x00000004` stands for 4×16^0 — which (in the conventional base 10 system) is plain old 4. And the number `0x00000024` stands for $2 \times 16^1 + 4 \times 16^0$ — which (in base 10) is 36. Finally, the number `0x0000001b` stands for $1 \times 16^1 + 11 \times 16^0$ — which (in base 10) is 27. As an Android developer, I seldom have to convert a hexadecimal value into its conventional base 10 representation. So don't worry about doing it.

Anyway, the app you see in Figure 4-11 displays the text *Hello world!* When you create an Android app, you seldom put actual words such as "Hello World!" in the app's Java code. Instead, you refer to the words indirectly. You give the words *Hello World!* a number, and you put that number in the Java code. More precisely, these things happen:

- You have the line `<string name="hello_world">Hello world!</string>` in the `strings.xml` file, which is in the `values` subdirectory of the project's `res` directory.
- Eclipse generates a code number, such as `0x7f040001`. (Refer to Listing 4-3.)
- Android associates the number `0x7f040001` with the synonym `R.string.hello_world` by having the text `hello_world=0x7f040001` in the `string` portion of the `R.java` file. (Refer to Listing 4-3).
- You have the text `R.string.hello_world` in the Java code. Alternatively, you have the text `@string/hello_world` in the `activity_main.xml` file.

This indirect way to refer to the words *Hello world!* might seem to be needlessly complicated. But the indirectness is exactly what helps you create apps that appeal to people all over the world. Look at the discussion of localization in the earlier section "The res directory." By creating a new `values-fr` directory, you allow a user's device to automatically localize to another language, and to display *Bonjour tout le monde!* or *Hallo Welt!* or *Hej Verden!* instead of the Anglocentric *Hello world!* phrase.

The `android.jar` file contains code grouped into Java packages, and each package contains Java classes. Figures 4-14 and 4-15 show you the tip of the `android.jar` iceberg. The `android.jar` file contains classes specific to Android and classes that simply help Java do its job. Figure 4-14 shows some Android-specific packages in `android.jar`. Figure 4-15 displays some general-purpose Java packages in the `android.jar` file.

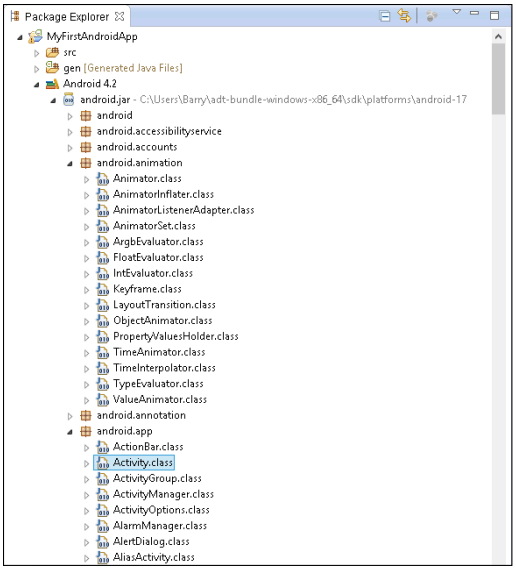

Figure 4-14: Some packages and classes in `android.jar`.

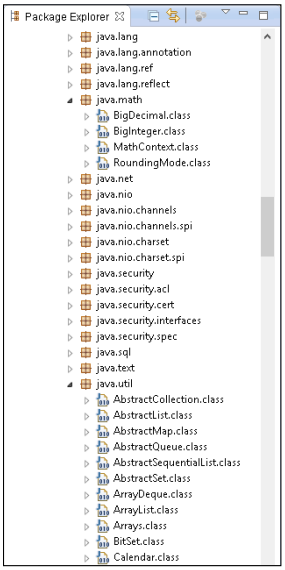

Figure 4-15: The `android.jar` file includes general-purpose Java packages.

The AndroidManifest.xml file

If you followed the instructions earlier in this chapter, you've already tinkered with an `AndroidMaifest.xml` file. Keep in mind that every Android app has an `AndroidManifest.xml` file. The `AndroidManifest.xml` file provides information that a device needs in order to run the

app. The `AndroidManifest.xml` file in Listing 4-4 stores some options that you choose when you create a brand-new Android project. For example, the listing contains the package name, the minimum required SDK (the `android:minSdkVersion` attribute), and the target SDK (the `android:targetSdkVersion` attribute).

Listing 4-4: An AndroidManifest.xml File

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android=
    "http://schemas.android.com/apk/res/android"
  package="com.example.myfirstandroidapp"
  android:versionCode="1"
  android:versionName="1.0" >

  <uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="16" />

  <application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
      android:name=
        "com.example.myfirstandroidapp.MainActivity"
      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>
```

For my money, the most important items in an `AndroidManifest.xml` file are the `activity` elements. The code in Listing 4-4 has only one `activity` element. But a single Android app can have many activities, and each activity must have its own `activity` element in the app's `AndroidManifest.xml` file.

For the scoop on Android activities, see Chapter 5.

An Android activity is one "screenful" of components. (Refer to Chapter 5 for more about Android activities.) If you add an activity's Java code to an Android application, you must also add an `activity` element to the application's `AndroidManifest.xml` file. If you forget to add an `activity` element, you see an `ActivityNotFoundException` when you try to run the application. (Believe me. I've made this mistake many, many times.)

Within an `activity` element, an `intent-filter` element describes the kinds of duties that this activity can fulfill for apps on the same device. (Intent filters are complicated, so in this book I don't dare open that whole can of worms.) But to give you an idea, the action `android.intent.action.MAIN` indicates that this activity's code can be the starting point of an app's execution. And the category `android.intent.category.LAUNCHER` indicates that this activity's icon can appear on the device's Apps screen.