WHAT IS ANDROID?

Android is a mobile operating system that is based on a modified version of Linux. It was originally developed by a startup of the same name, Android, Inc. In 2005, as part of its strategy to enter the mobile space, Google purchased Android, Inc. and took over its development work (as well as its development team).

Google wanted the Android OS to be open and free, so most of the Android code was released under the open source Apache License. That means anyone who wants to use Android can do so by downloading the full Android source code. Moreover, vendors (typically hardware manufacturers) can add their own proprietary extensions to Android and customize Android to differentiate their products from others. This development model makes Android very attractive to vendors, especially those companies affected by the phenomenon of Apple's iPhone, which was a hugely successful product that revolutionized the smartphone industry. When the iPhone was launched, many smartphone manufacturers had to scramble to find new ways of revitalizing their products. These manufacturers saw Android as a solution, meaning they will continue to design their own hardware and use Android as the operating system that powers it. Some companies that have taken advantage of Android's open source policy include Motorola and Sony Ericsson, which have been developing their own mobile operating systems for many years.

The main advantage to adopting Android is that it offers a unified approach to application development. Developers need only develop for Android in general, and their applications should be able to run on numerous different devices, as long as the devices are powered using Android. In the world of smartphones, applications are the most important part of the success chain.

Android Versions

Android has gone through quite a number of updates since its first release. Table 1-1 shows the various versions of Android and their codenames.

TABLE 1-1: A Brief History of Android Versions

ANDROID VERSION	RELEASE DATE	CODENAME
1.1	February 9, 2009	
1.5	April 30, 2009	Cupcake
1.6	September 15, 2009	Donut
2.0/2.1	October 26, 2009	Éclair
2.2	May 20, 2010	Froyo
2.3	December 6, 2010	Gingerbread
3.0/3.1/3.2	February 22, 2011	Honeycomb
4.0	October 18, 2011	Ice Cream Sandwich

ANDROID VERSION	RELEASE DATE	CODENAME
4.1	July 9, 2012	Jelly Bean
4.4	October 31, 2013	KitKat
5.0	November 12, 2014	Lollipop
6.0	October 5, 2015	Marshmallow
7.0	TBD	Nougat

In 2016, Google released Android 7.0; the following are the key changes in Android 7.0:

- Split-screen multi-window mode
- Redesigned notification shade
- Refined "Doze" feature
- Switch from JRE (Java Runtime Environment) to OpenJDK

One important thing to keep in mind as you are looking at Android versions is that each version has its own features and APIs (application programming interfaces). Therefore, if your application is written for the newest version of Android, and it uses an API that was not present in an older version of Android, then only devices running that newer version of Android will be able to use your application.

Features of Android

Because Android is open source and freely available to manufacturers for customization, there are no fixed hardware or software configurations. However, the base Android OS supports many features, including

- **Storage**—SQLite, a lightweight relational database, for data storage. Chapter 7 discusses data storage in more detail.
- **Connectivity**—GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth (includes A2DP and AVRCP), Wi-Fi, LTE, and WiMAX. Chapter 11 discusses networking in more detail.
- **Messaging**—Both SMS and MMS. Chapter 9 discusses messaging in more detail.
- **Media support** H.263, H.264 (in 3GP or MP4 container), MPEG-4 SP, AMR, AMR-WB (in 3GP container), AAC, HE-AAC (in MP4 or 3GP container), MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP.
- **Hardware support**—Accelerometer sensor, camera, digital compass, proximity sensor, and GPS.
- **Multi-touch**—Multi-touch screens.
- **Multi-tasking**—Multi-tasking applications.
- **Tethering**—Sharing of Internet connections as a wired/wireless hotspot.

Android's web browser is based on the open source WebKit and Chrome's V8 JavaScript engine.

Architecture of Android

To understand how Android works, take a look at Figure 1-1, which shows the various layers that make up the Android operating system (OS).

The Android OS is roughly divided into five sections in four main layers:

- **Linux kernel**—This is the kernel on which Android is based. This layer contains all the low-level device drivers for the various hardware components of an Android device.
- **Libraries**—These contain the code that provides the main features of an Android OS. For example, the SQLite library provides database support so that an application can use it for data storage. The WebKit library provides functionalities for web browsing.
- **Android runtime**—The Android runtime is located in the same layer with the libraries and provides a set of core libraries that enable developers to write Android apps using the Java programming language. The Android runtime also includes the Dalvik virtual machine, which enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine. (Android applications are compiled into Dalvik executables). Dalvik is a specialized virtual machine designed specifically for Android and optimized for battery-powered mobile devices with limited memory and CPU power.
- **Application framework**—The application framework exposes the various capabilities of the Android OS to application developers so that they can make use of them in their applications.
- **Applications**—At this top layer are the applications that ship with the Android device (such as Phone, Contacts, Browser, and so on), as well as applications that you download and install from the Android Market. Any applications that you write are located at this layer.

Android Devices in the Market

Android devices come in all shapes and sizes including, but not limited to, the following types of devices:

- Smartphones
- Tablets
- E-reader devices
- Internet TVs
- Automobiles
- Smartwatches

Chances are good that you own at least one of the preceding devices. Figure 1-2 shows the Samsung Galaxy Edge 7.

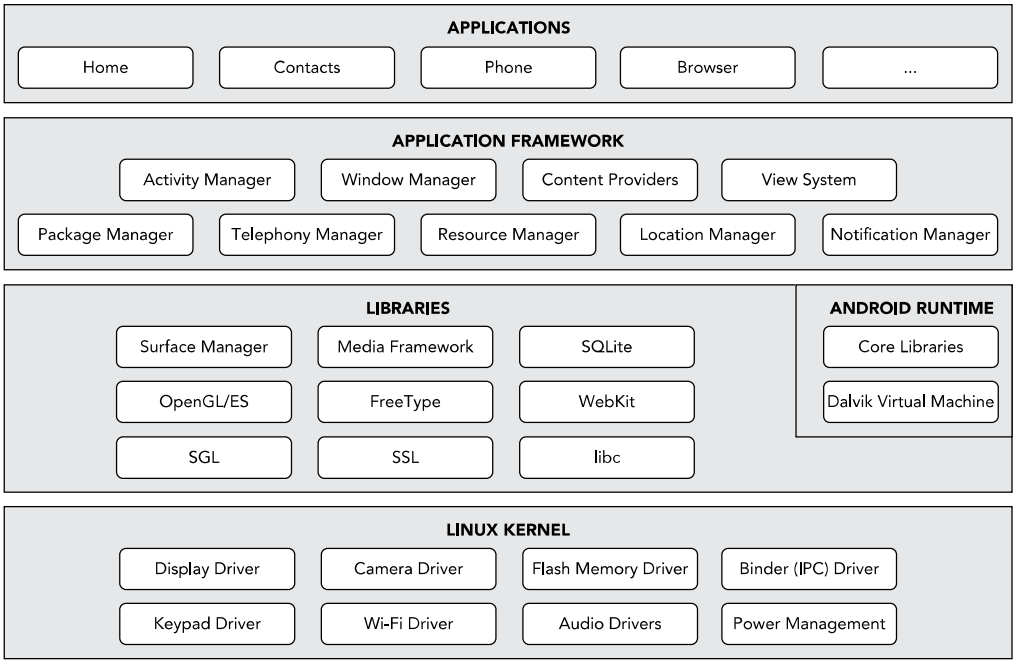
APPLICATIONS
Home
Contacts
Phone
Browser
...
APPLICATION FRAMEWORK
Activity Manager
Window Manager
Content Providers
View System
Package Manager
Telephony Manager
Resource Manager
Location Manager
Notification Manager
LIBRARIES
Surface Manager
Media Framework
SQLite
OpenGL/ES
FreeType
WebKit
SGL
SSL
libc
ANDROID RUNTIME
Core Libraries
Dalvik Virtual Machine
LINUX KERNEL
Display Driver
Camera Driver
Flash Memory Driver
Binder (IPC) Driver
Keypad Driver
Wi-Fi Driver
Audio Drivers
Power Management

FIGURE 1-1

FIGURE 1-2

Another popular category of devices is the tablet. Tablets typically come in two sizes: 7″ and 10″, measured diagonally.

Besides smartphones and tablets, Android is used in dedicated devices, such as e-book readers. Figure 1-4 shows the Barnes and Noble's NOOK Color running the Android OS.

In addition to the popular mobile devices I've already mentioned, Android is finding its way onto your wrist. Smartwatches, and "wearables" in general, have become a major segment of the Android population. Figure 1-3 shows the Motorola Moto 360 Smartwatch, which runs Android Wear (a version of Android OS specifically designed for wearables).

At the time of writing, the Samsung Galaxy Nexus (see Figure 1-4) is the only device running a pure version of Android. Many manufacturers add their own modifications to the Android OS for use on their specific devices. Motorola devices

FIGURE 1-3

have Motoblur, HTC devices have HTC Sense, and so on. However, the Nexus devices always run a clean version of Android with no modifications.

FIGURE 1-4

The Android Market

As mentioned earlier, one of the main factors determining the success of a smartphone platform is the applications that support it. It is clear from the success of the iPhone that applications play a very vital role in determining whether a new platform swims or sinks. Also, making these applications accessible to the general user is extremely important.

Users can simply use the Google Play application that is preinstalled on their Android devices to directly download third-party applications to their devices. Both paid and free applications are available in the Google Play Store, although paid applications are available only to users in certain countries because of legal issues.

> ***NOTE*** *Chapter 13 discusses more about Google Play Store and how you can sell your own applications in it.*

OBTAINING THE REQUIRED TOOLS

Now that you know what Android is and what its feature set contains, you are probably anxious to get your hands dirty and start writing some applications! Before you write your first app, however, you need to download the required tools.

For Android development, you can use a Mac, a Windows PC, or a Linux machine. You can freely download all the necessary tools. Most of the examples provided in this book are written to work on Android Studio. For this book, I am using a Windows 10 computer to demonstrate all the code samples. If you are using a Mac or Linux computer, the screenshots should look similar. Some minor differences might be present, but you should be able to follow along without problems.

Let the fun begin!

JAVA JDK 8

The Android Studio 2 makes use of the Java SE Development Kit 8 (JDK). If your computer does not have the JDK 8 installed, you should start by downloading it from `www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html` and installing it prior to moving to the next section.

Android Studio

The first and most important piece of software you need to download is Android Studio 2. After you have downloaded and installed Android Studio 2, you can use the SDK Manager to download and install multiple versions of the Android SDK. Having multiple versions of the SDK available enables you to write programs that target different devices. For example, you can write one version of an application that specifically targets Android Nougat, but because that flavor of Android is on less than 1% of devices, with multiple versions of the SDK you can also write a version of your app that uses older features and targets Marshmallow or Lollipop users. You can use the Android Device Manager to set up device emulators.

You can download Android Studio 2 from `http://developer.android.com/sdk/index.html` (see Figure 1-5).

Android Studio 2 is packaged in an executable. Run the install process to set up Android Studio 2. After you've downloaded and run the setup executable, use the following steps to go through the installation process:

1. Accept the terms and conditions shown in Figure 1-6.
2. If you have an older version of Android Studio already installed on your computer, the Android Studio Setup prompts you to automatically uninstall it. Even though the old version of Android Studio will be uninstalled, the settings and configurations are retained. You have an opportunity to reapply those settings and configurations to Android Studio 2 after the setup has completed. Figure 1-7 shows the screen where you are prompted to uninstall an old version of Android Studio.

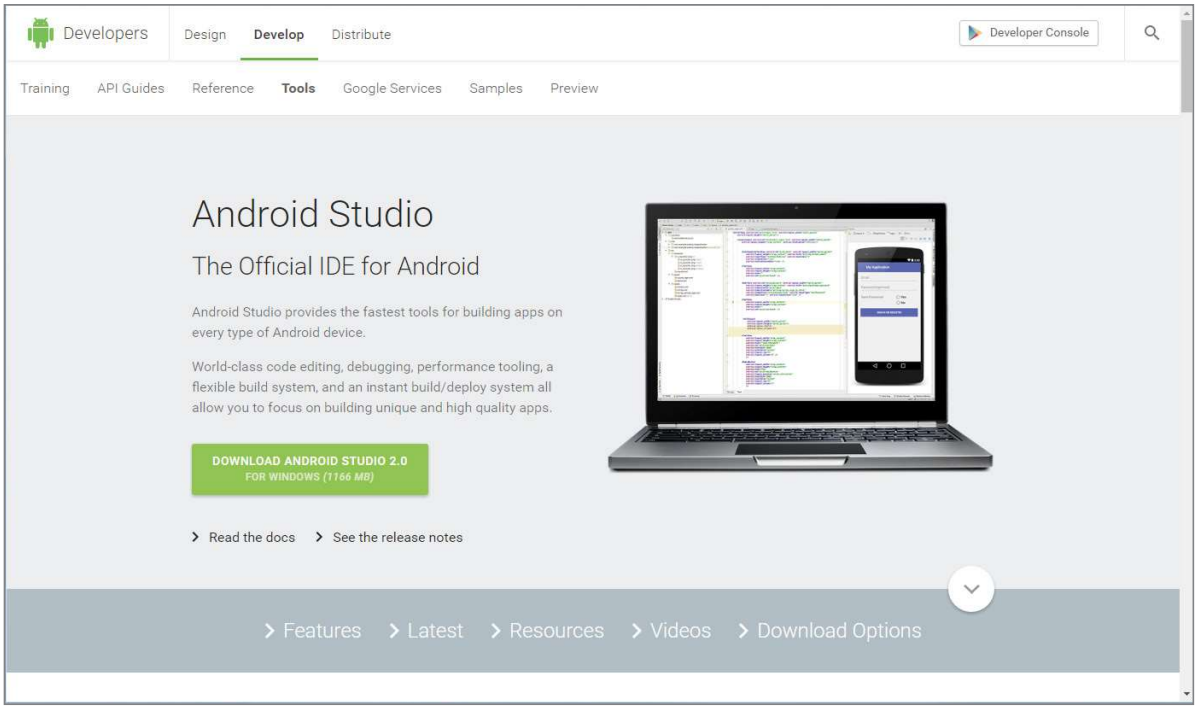

FIGURE 1-5

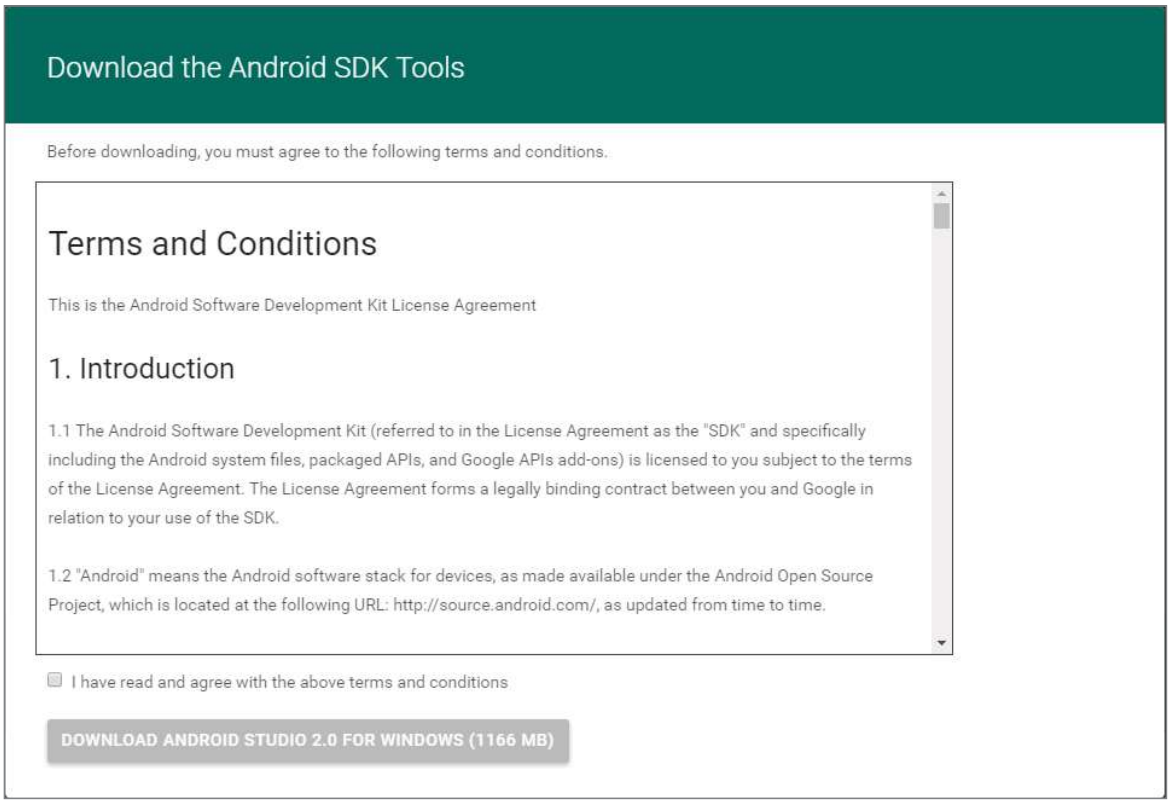

FIGURE 1-6

FIGURE 1-7

3. Click Next on the Welcome to Android Studio Setup screen (see Figure 1-8).

FIGURE 1-8

4. Pick which components of Android Studio you want to install from the screen shown in Figure 1-9. Android Studio is selected by default (and cannot be deselected), which makes sense given that you are going through all of this trouble for the distinct purpose of installing Android Studio. Android SDK and Android Virtual Device are also selected by default. Click Next to accept the default choices and continue.

FIGURE 1-9

5. You are presented with the License Agreement, as shown in Figure 1-10. Click I Agree to continue.

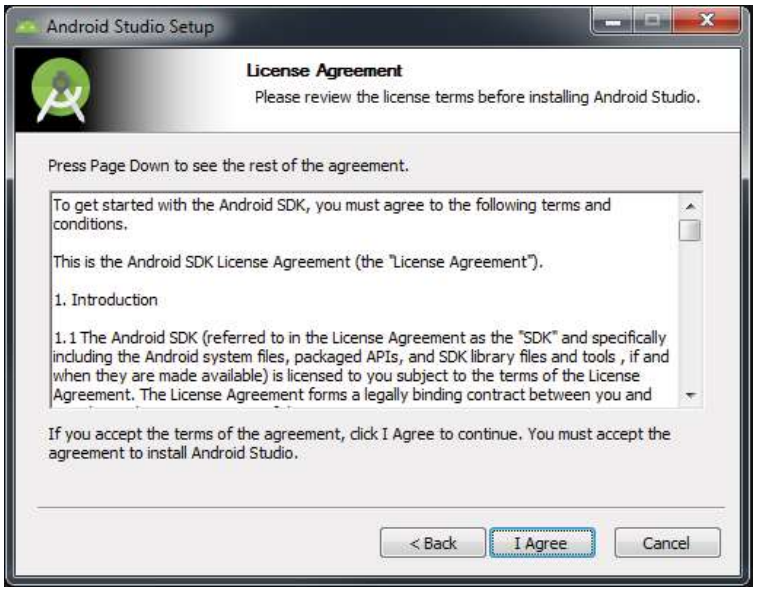

FIGURE 1-10

6. On the configuration settings screen, it is best to accept the default locations specified by the setup process and click Next to continue. You see the Choose Start Menu Folder screen (shown in Figure 1-11). Click Install to kick off the Android Studio 2 installation.

FIGURE 1-11

7. Installing Android Studio 2 could take a few minutes, depending on the speed of your computer. You are presented with a progress bar to help you track the state of the installation. Android Studio 2 is installed with a default SDK (Software Development Kit), in this case Marshmallow. Later in the process you have the opportunity to install other SDKs. The Android SDK allows you to develop and write applications geared for a specific version of Android. In other words, applications written with the Marshmallow SDK run on Android devices running Marshmallow, but they also possibly run on other versions depending on which features of the SDK you used in the application.

8. When the install is complete, you will see a Completing Android Studio Setup screen (shown in Figure 1-12). Leave the Start Android Studio box checked and click Finish.

FIGURE 1-12

9. Android Studio 2 prompts you to either import settings from a previous version of Android Studio or continue with new settings. If you uninstalled a previous version in the first step of the installation process, Android Studio offers you a chance to recover the settings used in that previous version and apply them to Android Studio 2 (see Figure 1-13).

FIGURE 1-13

Now that Android Studio 2 is installed, you need to adjust the settings and options using the following steps:

1. Click Continue at the Welcome screen and choose Standard from the Install Type selection screen shown in Figure 1-14. Click Next to continue.

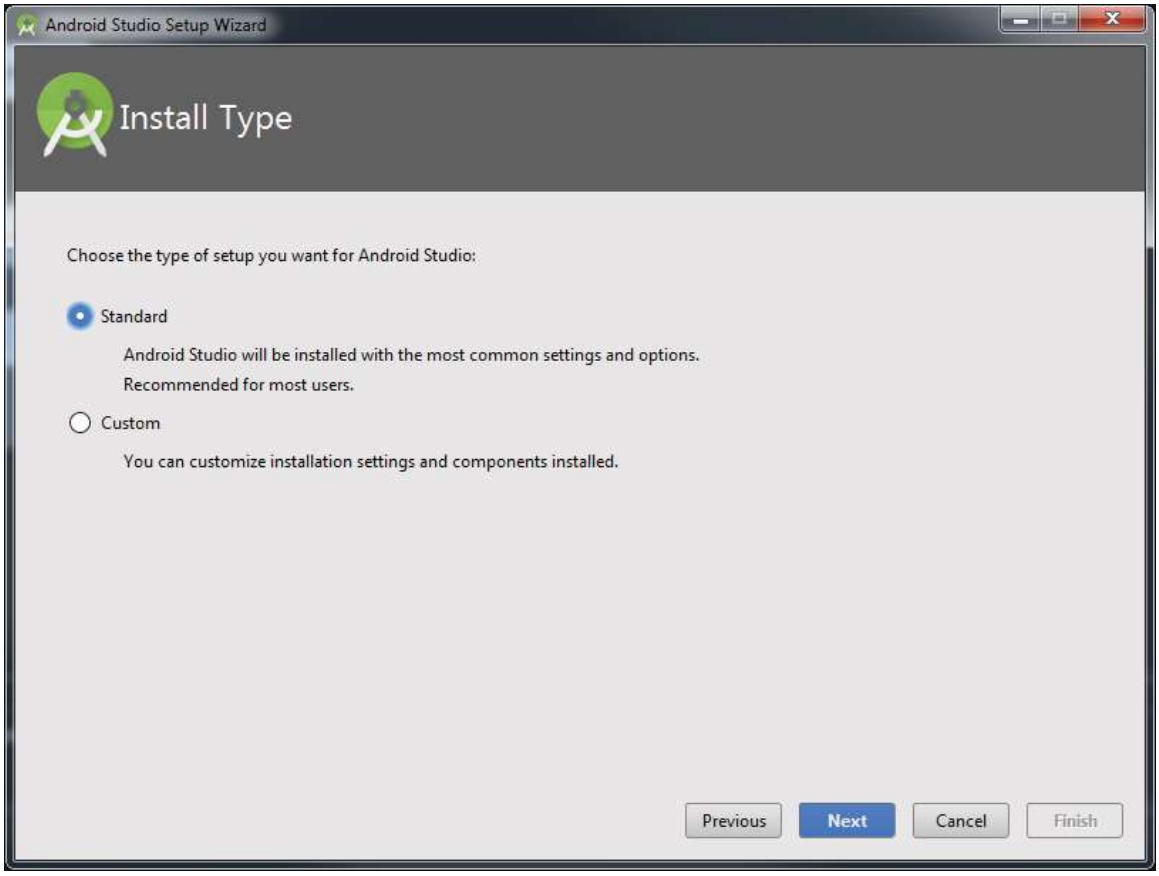

FIGURE 1-14

2. Click Finish on the Verify Settings screen, and Android Studio 2 finalizes the setup process. You know the process is complete when you are greeted with the Welcome to Android Studio screen (see Figure 1-15).

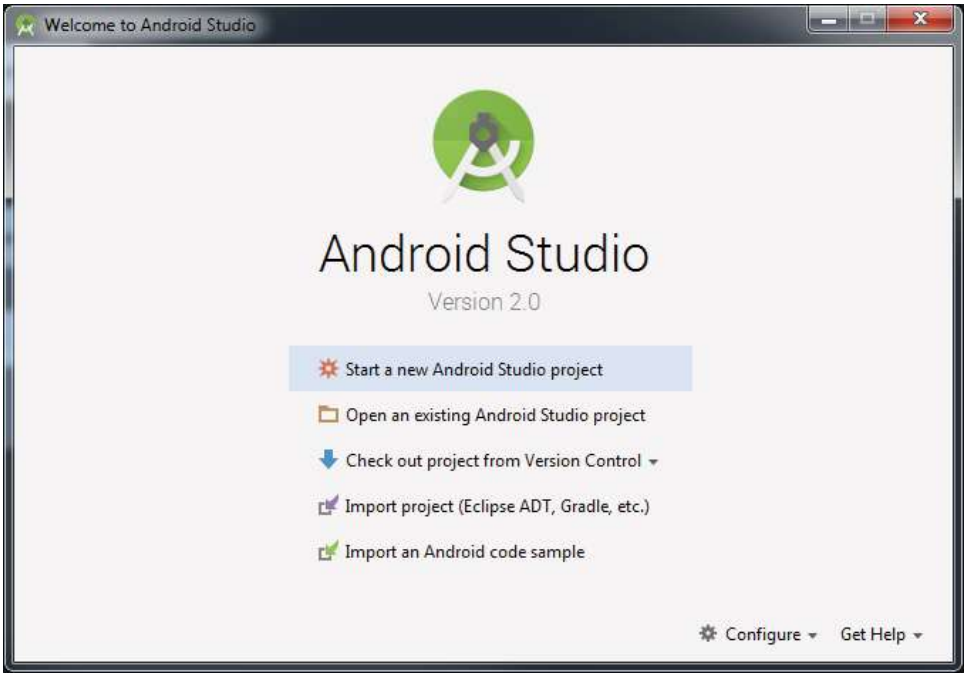

FIGURE 1-15

Now that Android Studio is set up, it's time to install the latest and greatest Android SDK.

Android SDK

The most important piece of software you need to download is, of course, the Android SDK. The Android SDK contains all of the packages and tools required to develop a functional Android application. The SDKs are named after the version of Android OS to which they correspond. By default, the Marshmallow SDK was installed with Android Studio 2, which means you can develop applications that will run seamlessly on devices with Android Marshmallow.

However, if you want to install a different Android SDK, you can do so using the SDK Manager from the Android Studio welcome screen (shown in Figure 1-15). From this screen, click the Configure drop-down menu in the lower-right corner. The Configure selection menu opens. Choose SDK Manager from this menu.

The SDK configuration screen, shown in Figure 1-16, shows that the Marshmallow SDK is already installed. Android N is available to be installed (as of the writing of this book Android Nougat was in a finalized beta, so it might be named differently now).

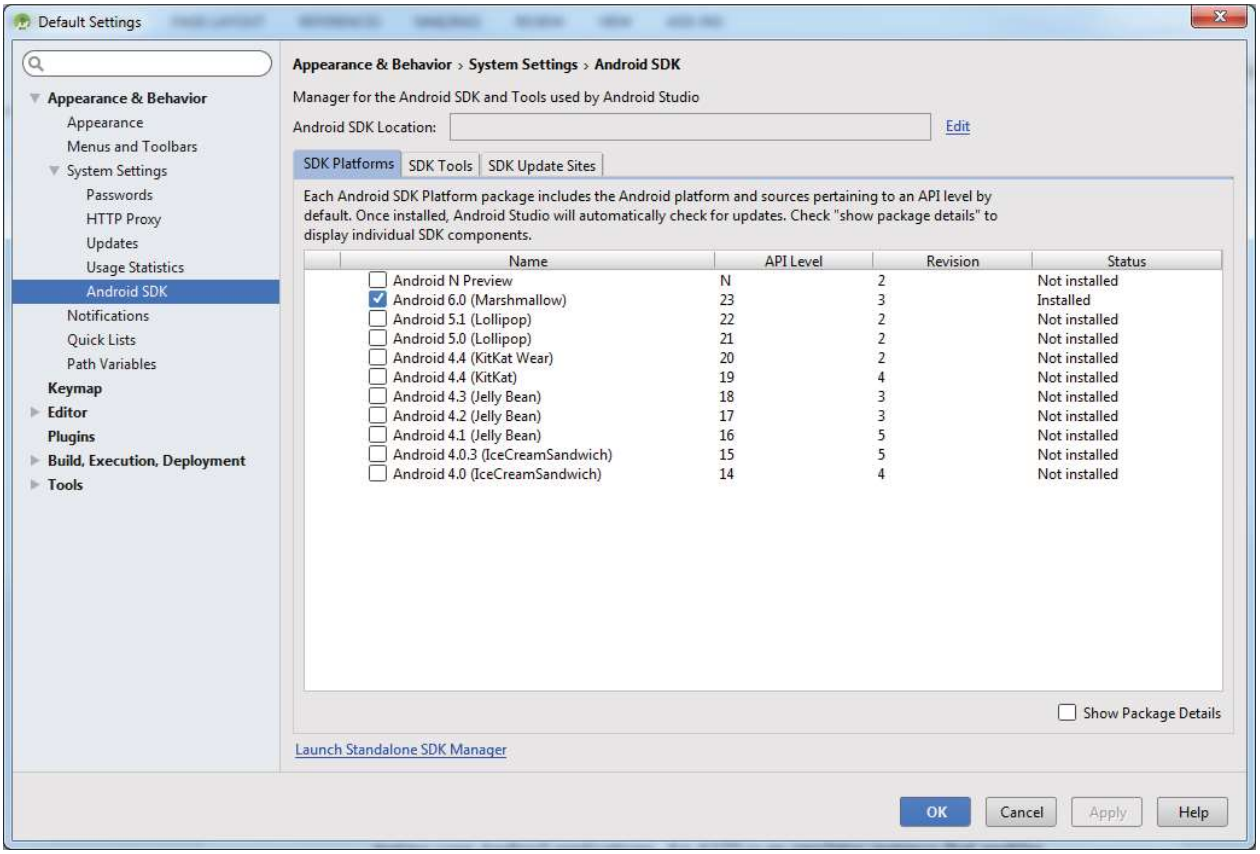

FIGURE 1-16

Select Android Nougat, click Apply, and then click OK. However, before the SDK is installed you must accept the licensing agreement as shown in Figure 1-17.

The setup process for Android Studio is now complete. The next section explains how to set up an Android Virtual Device that you can use to test your applications.

Creating Android Virtual Devices (AVDs)

The next step is to create an Android Virtual Device (AVD) you can use for testing your Android applications. An AVD is an emulator instance that enables you to model an actual device. Each AVD consists of a hardware profile; a mapping to a system image; and emulated storage, such as a secure digital (SD) card. One important thing to remember about emulators is that they are not perfect. There are some applications, such as games (which are GPU heavy) or applications that use sensors such as the GPS or accelerometer. These types of applications cannot be simulated with the same speed or consistency within an emulator as they can when running on an actual device. However, the emulator is good for doing some generalized testing of your applications.

You can create as many AVDs as you want to test your applications with different configurations. This testing is important to confirm the behavior of your application when it is run on different devices with varying capabilities.

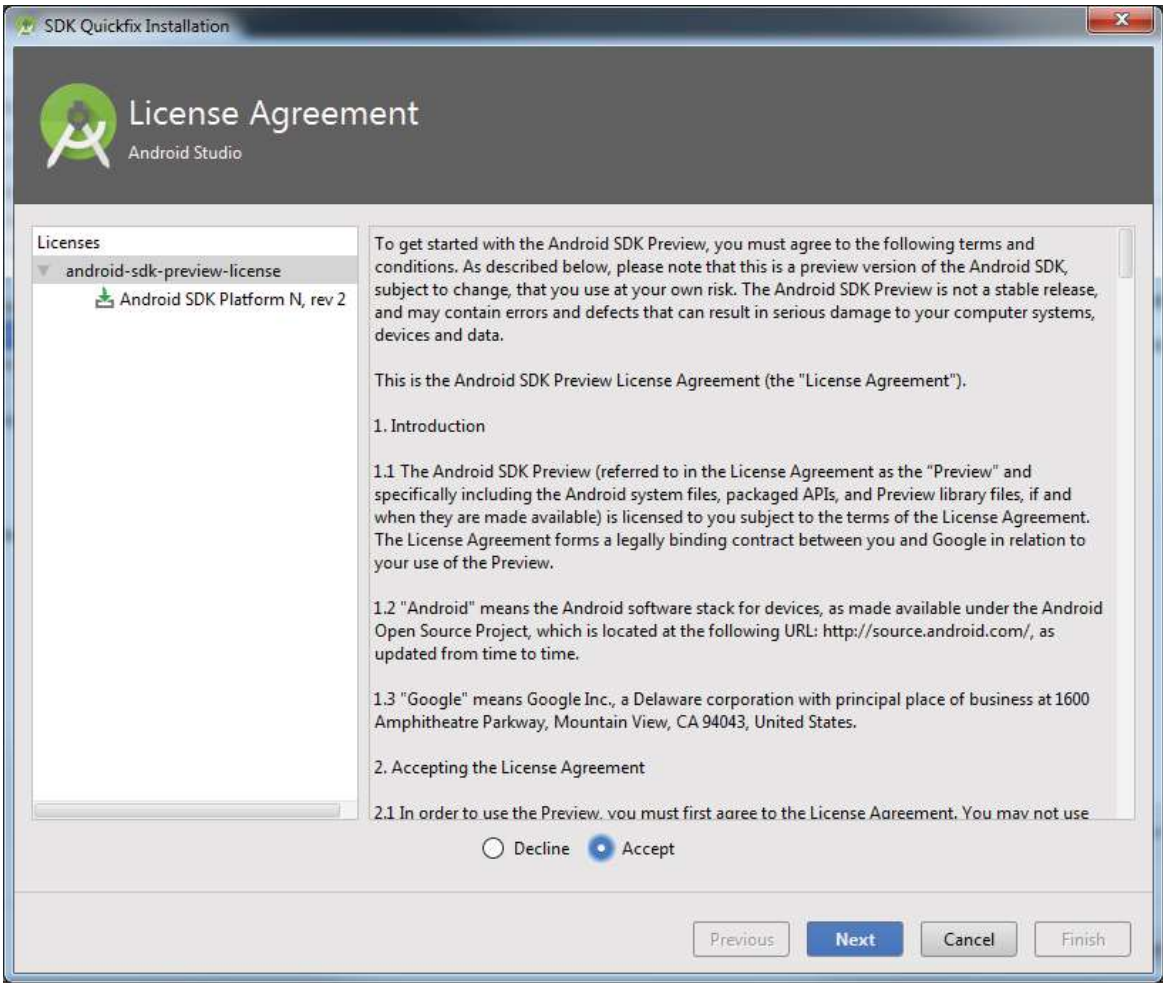

FIGURE 1-17

Use the following steps to create an AVD. This example demonstrates creating an AVD (put simply, an Android emulator) that emulates an Android device running Android N on the Nexus 5x hardware specs.

1. Start Android Studio so that the Welcome screen is visible (refer to Figure 1-15). Click Start a New Android Studio Project. You see the Create New Project Wizard shown in Figure 1-18.
2. Set up a HelloWorld project (that you will use in the final section of this chapter). Type **Chapter1Helloworld** in the Application Name field.

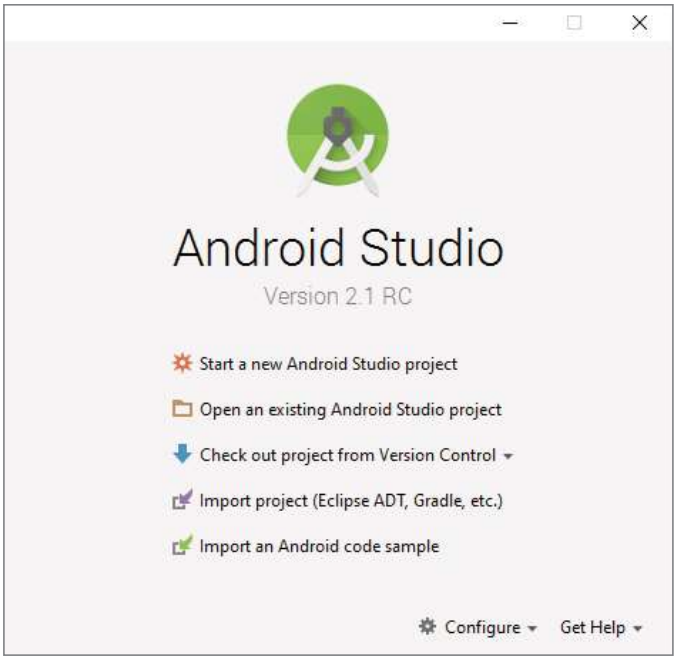

FIGURE 1-18

3. You can keep the default values for the other fields on the New Project screen (they will be explained in more detail in later chapters). Click Next.

> ***NOTE*** *For the purposes of setting up a quick Hello World project and creating an AVD, you will be accepting many of the default values, without explanation, during the project setup process. This is fine for now, as all of the settings are explained in much greater detail in subsequent chapters.*

4. You should see the Targeted Android Devices screen. By default, the Create New Project Wizard selects for you the Android SDK level that has the greatest activity based on statistics gathered from Google Play. At the time this book was written 74.3 percent of the active devices on Google Play were written using Android Jelly Bean. For now, accept the default, as shown in Figure 1-19, and click Next.
5. On the Add an Activity to Mobile screen, accept the default choice—Empty Activity (see Figure 1-20)—and click Next.

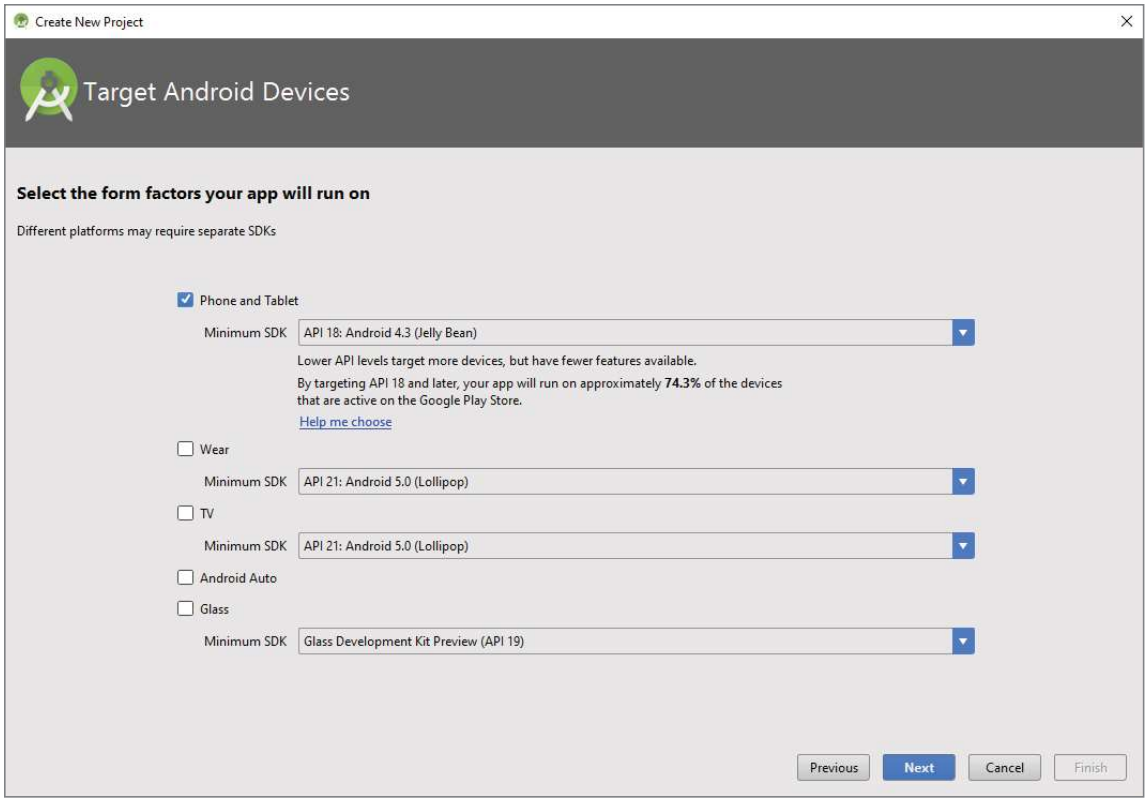

FIGURE 1-19

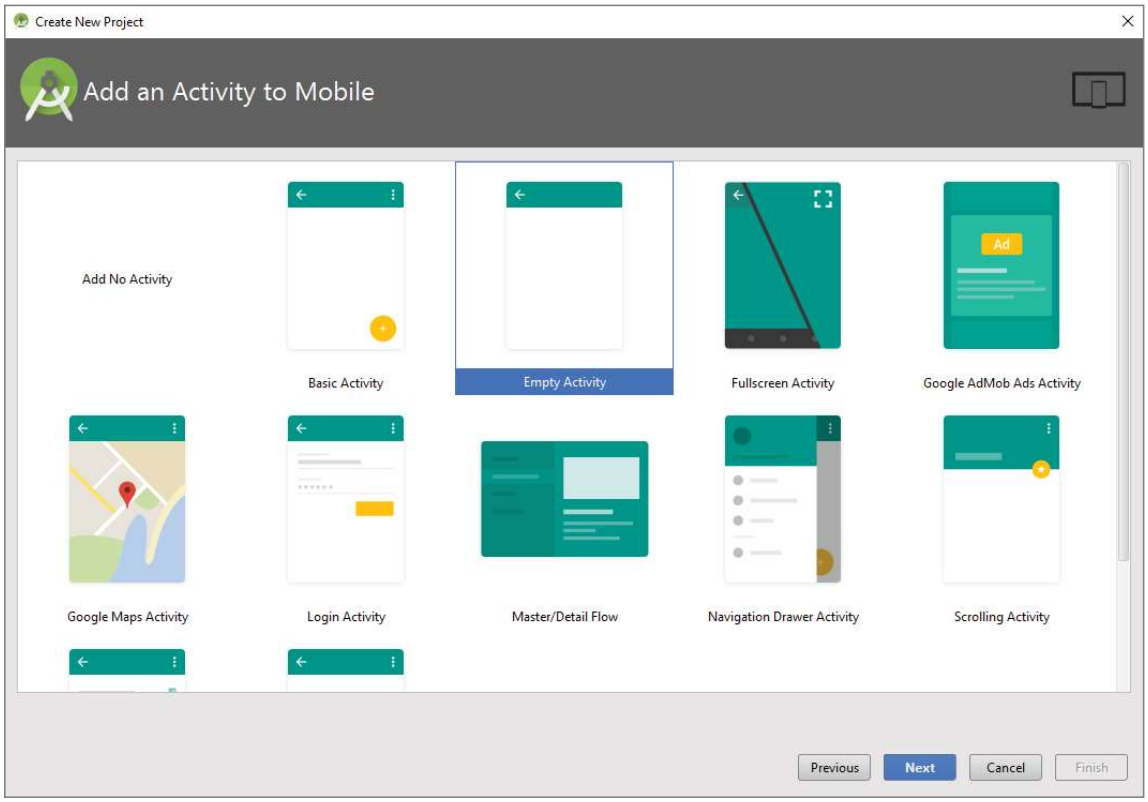

FIGURE 1-20

6. Accept all of the defaults on the Customize the Activity screen, as shown in Figure 1-21, and click Finish. Figure 1-22 shows the open Android Studio IDE.

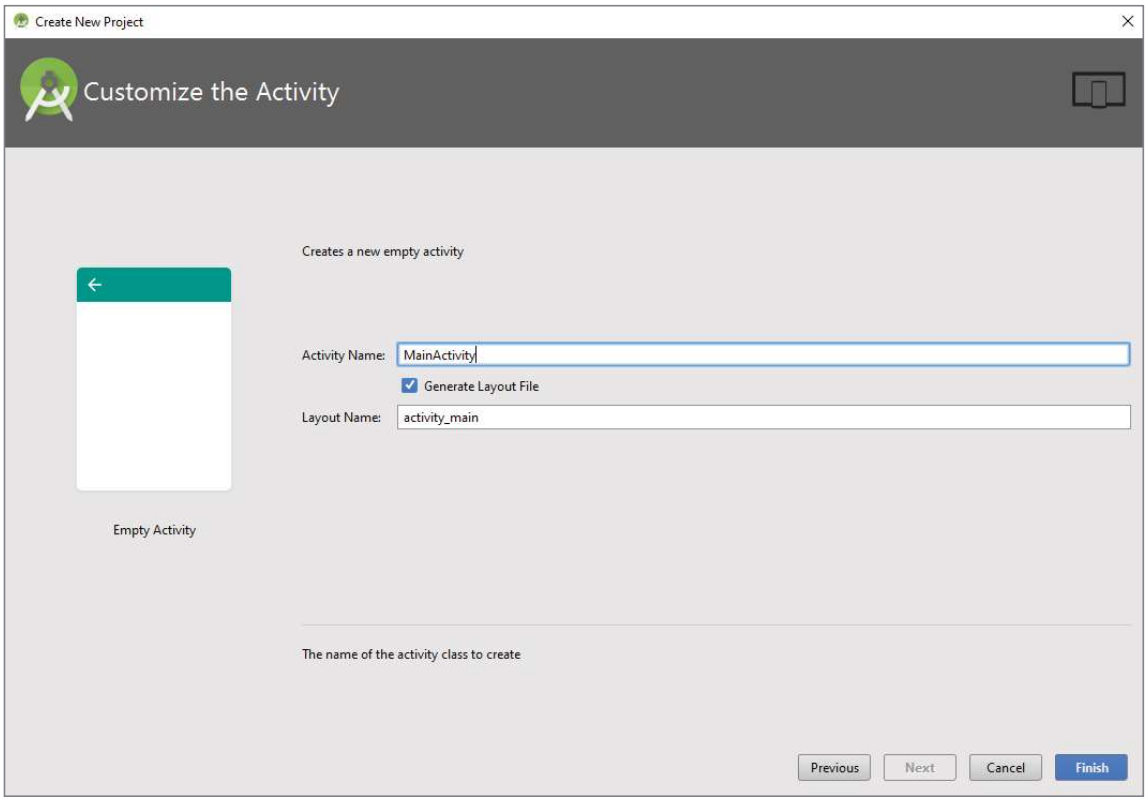

FIGURE 1-21

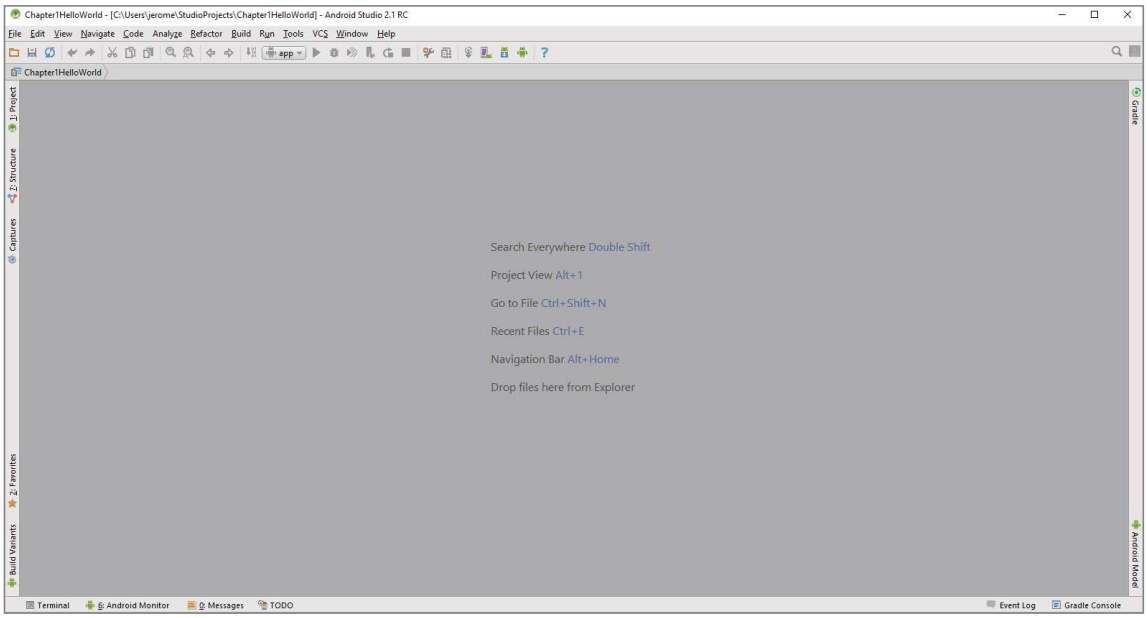

FIGURE 1-22

7. Launch the AVD Manager by selecting Tools ➪ Android ➪ AVD Manager or using the AVD Manager button from the toolbar. Figure 1-23 shows the Android Virtual Device Manager Wizard, which is where you set up AVDs to be used when you emulate your application in Android on your desktop.

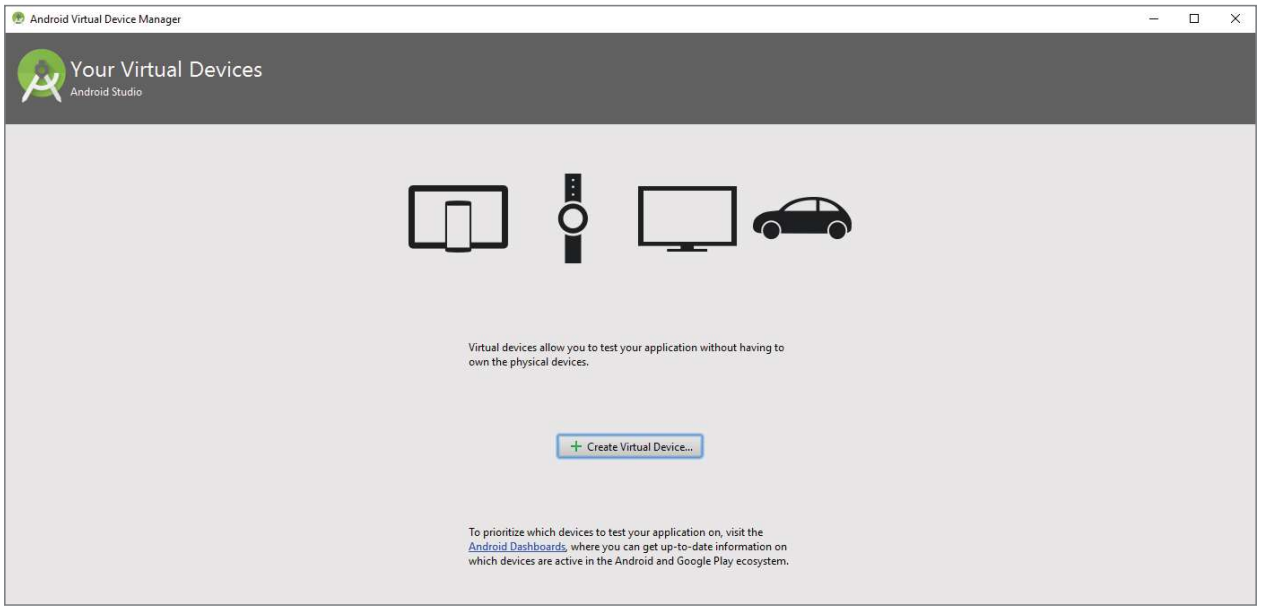

FIGURE 1-23

8. Click the + Create Virtual Device button to create a new AVD. The Virtual Device Configuration screen opens as shown in Figure 1-24.
9. Select the Nexus 5x hardware profile and click Next. Although none of the emulators offers the same performance as its actual hardware counterpart, the Nexus 5x should run well on most x86-based desktops, and it still offers some of the mid- to high-end Android device specs.
10. For the system image, select and install the latest option, which at the time this book was written is Android Nougat. Click the x86 Images tab (see Figure 1-25), select N from the list of images, and then click Next.

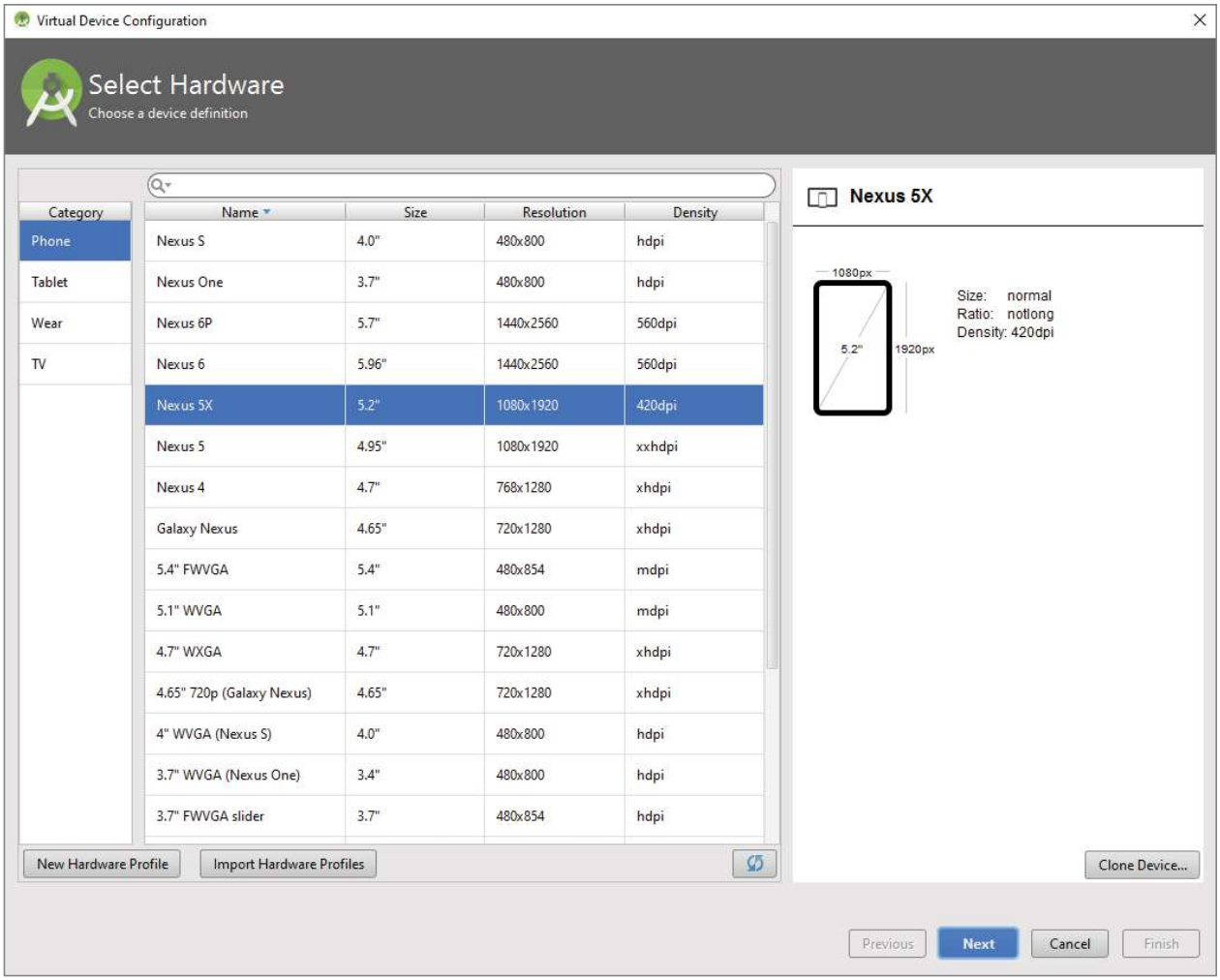

FIGURE 1-24

11. In the Android Virtual Device (AVD) dialog, accept the defaults as shown in Figure 1-26. Click the Finish button to begin building the AVD.

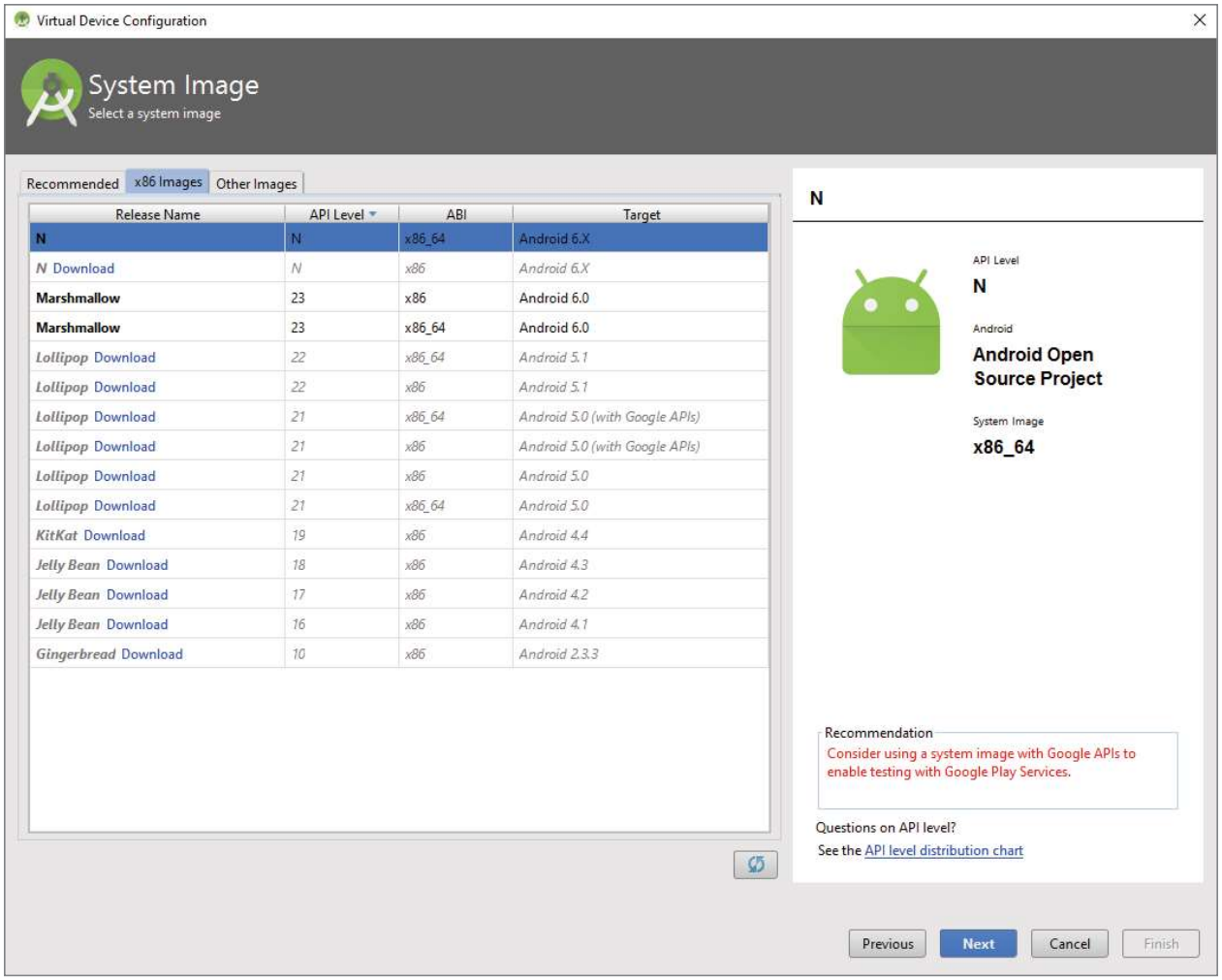

FIGURE 1-25

> ***TIP*** *It is preferable to create a few AVDs with different API levels and hardware configurations so that your application can be tested on different versions of the Android OS.*

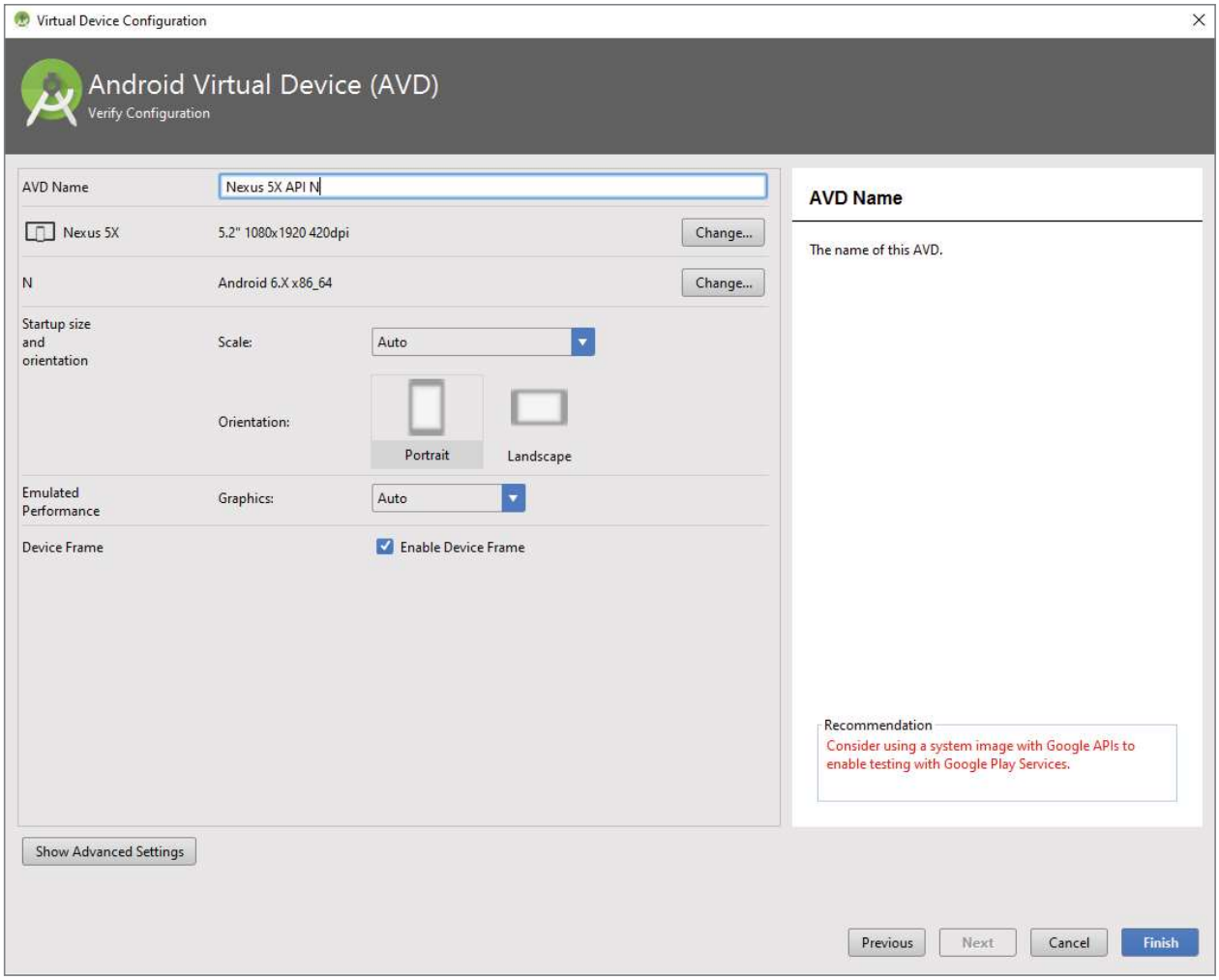

FIGURE 1-26

TRY IT OUT Creating a Jellybean Emulator

When you created your first Android project earlier in this section, the setup process determined that Jelly Bean was the most active version of Android on Google Play. In this Try It Out, you create an AVD for Android Jelly Bean.

1. Launch the AVD Manager by selecting Tools ➪ Android ➪ AVD Manager or using the AVD Manager button from the toolbar.
2. In the Android Virtual Device Manager Wizard, click the + Create Virtual Device button.

3. Select the Nexus 5x hardware profile and click Next.
4. Click the x86 Images tab, select Jelly Bean from the list of images, and then click Download.
5. Accept the agreement and download the Jelly Bean SDK.
6. After the SDK has downloaded, click Jelly Bean once again (on the x86 Images tab) and click Next.
7. In the Android Virtual Device (AVD) dialog, accept the defaults and click the Finish button.

After you have created your ADV, it is time to test it. There is no better way to do this than to create and launch the ubiquitous Hello World application.

The Android Developer Community

Now that Android is in its seventh version, there is a large developer community all over the world. It is easy to find solutions to problems and to find like-minded developers with whom to share app ideas and experiences.

The following are some developer communities and websites that you can turn to for help if you run into problems while working with Android:

- **Stack Overflow** (`www.stackoverflow.com`)—Stack Overflow is a collaboratively edited question-and-answer site for developers. If you have a question about Android, chances are someone at Stack Overflow is probably already discussing the same question. It's also likely that someone else has already provided the answer. Best of all, other developers can vote for the best answer so that you can know which are the answers that are most trustworthy.
- **Google Android Training** (`http://developer.android.com/training/index.html`)—Google has launched the Android Training site, which contains a number of useful classes grouped by topics. At the time of writing, the classes mostly contain code snippets that are useful to Android developers who have started with the basics. After you have learned the basics in this book, I strongly suggest you take a look at the classes.
- **Android Discuss** (`http://groups.google.com/group/android-discuss`)—Android Discuss is a discussion group hosted by Google using the Google Groups service. Here, you will be able to discuss the various aspects of Android programming. This group is monitored closely by the Android team at Google, so this is good place to clarify your doubts and to learn new tips and tricks.

LAUNCHING YOUR FIRST ANDROID APPLICATION

With all the tools and the SDK downloaded and installed, it is now time to start your engine. As in most programming books, the first example uses the ubiquitous Hello World application. This will give you a detailed look at the various components that make up an Android project. This is also the easiest Android project you will ever make.

Believe it or not, the Hello World application is already finished. By default, when you create a new application in Android Studio, it creates a Hello World application. Let's launch this application and, in the process, also launch the Android emulator to see how everything works.

1. Select Run ➪ Run *app* from the Android Studio menu bar. You should see the Select Deployment Target dialog as shown in Figure 1-27.

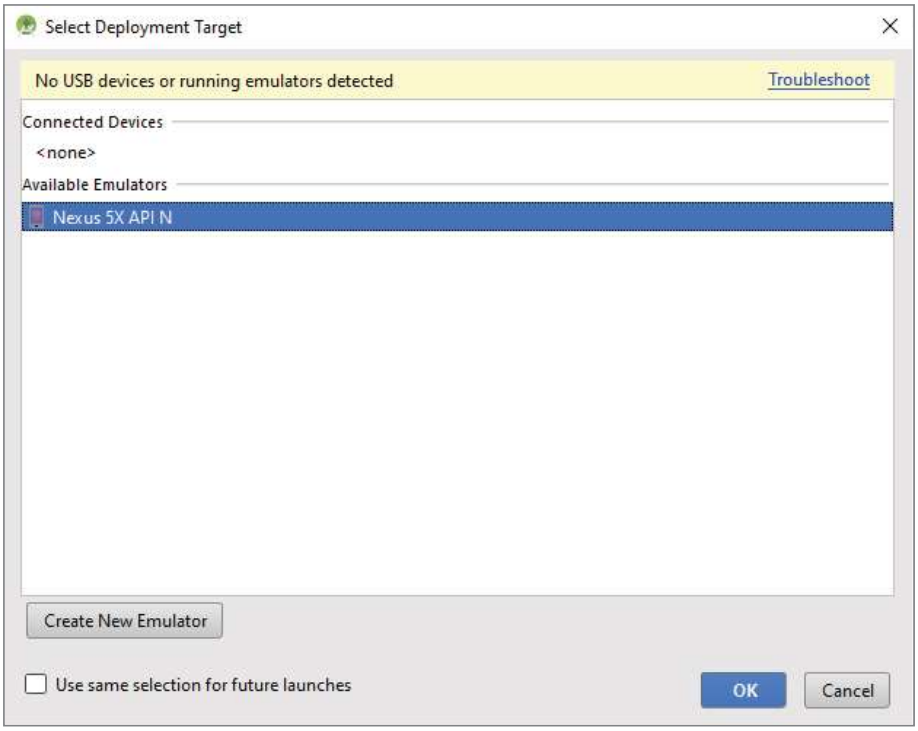

FIGURE 1-27

2. Select the Nexus 5X API N (feel free to select the Nexus 5x API 18, which is the Jelly Bean emulator that you created in the Try It Out for the last section), and click Next.

> ***NOTE*** *Note that If there's ever a time when you have not already created the emulator, you can create an emulator at this point.*

3. It can take up to five minutes, and sometimes longer (depending on the hardware specs of your desktop) for the emulator to start and fully load. During this time (the first time you launch the emulator) the application might time out. If a message pops up in Android Studio telling you that the application timed out waiting for the ADB (Android Debugging Bridge) to start, or another similar message, just wait for the emulator to fully load, and then once again select Run ➪ Run *app* from the Android Studio menu bar.

With the emulator fully loaded and started, Android Studio can install your Hello World application. The application will display as shown in Figure 1-28.

FIGURE 1-28

This was a very quick example of how to create and launch your first Android applications. However, what this example has really done for you is introduce you, on a general scale, to most of the major skills you will fine tune throughout this book.

SUMMARY

This chapter provided a brief overview of Android and highlighted some of its capabilities. If you have followed the sections on downloading the tools and the Android SDK, you should now have a working system—one that is capable of developing Android applications that are more interesting than the Hello World application. In the next chapter, you find out about the inner workings of Android Studio before moving on to more complex Android application development concepts.

EXERCISES

1. What is an AVD?
2. Why was Jelly Bean selected for you by default in the Targeted Android Devices dialog?
3. What does SDK stand for?
4. What tool is used to download new Android SDKs?

You can find answers to the exercises in the appendix.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Android OS	Android is an open source mobile operating system based on the Linux operating system. It is available to anyone who wants to adapt it to run on their own devices.
Languages used for Android application development	You use the Java programming language to develop Android applications. Written applications are compiled into Dalvik executables, which are then run on top of the Dalvik virtual machine.
Google Play	Google Play hosts all the various Android applications written by third-party developers.
Tools for Android application development	Android Studio, Android SDK, and virtual devices.

3

Activities, Fragments, and Intents

WHAT YOU WILL LEARN IN THIS CHAPTER

- The life cycles of an activity
- Using fragments to customize your UI
- Applying styles and themes to activities
- How to display activities as dialog windows
- Understanding the concept of intents
- Displaying alerts to the user using notifications

> ***CODE DOWNLOAD*** *The wrox.com code downloads for this chapter are found at* `www.wrox.com/go/beginningandroidprog` *on the Download Code tab. The code is in the chapter 03 download and individually named according to the names throughout the chapter.*

An Android application can have zero or more *activities*. Typically, applications have one or more activities. The main purpose of an activity is to interact with the user. From the moment an activity appears on the screen to the moment it is hidden, it goes through a number of stages. These stages are known as an activity's *life cycle*. Understanding the life cycle of an activity is vital to ensuring that your application works correctly. In addition to activities, Android N also supports *fragments*, a feature that was introduced for tablets in Android 3.0 and for phones in Android 4.0. Think of fragments as "miniature" activities that can be grouped to form an activity. In this chapter, you find out how activities and fragments work together.

Apart from activities, another unique concept in Android is that of an *intent*. An intent is basically the "glue" that enables activities from different applications to work together seamlessly, ensuring that tasks can be performed as though they all belong to one single application. Later in this chapter, you learn more about this very important concept and how you can use it to call built-in applications such as the Browser, Phone, Maps, and more.

UNDERSTANDING ACTIVITIES

This chapter begins by showing you how to create an activity. To create an activity, you create a Java class that extends the `Activity` base class:

```
package com.jfdimarzio.chapter1helloworld;

        import android.support.v7.app.AppCompatActivity;
        import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Your activity class loads its user interface (UI) component using the XML file defined in your `res/layout` folder. In this example, you would load the UI from the `main.xml` file:

```
        setContentView(R.layout.activity_main);
```

Every activity you have in your application must be declared in your `AndroidManifest.xml` file, like this:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android
    package="com.jfdimarzio.chapter1helloworld">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
         <activity android:name=".MainActivity">
             <intent-filter>
                 <action android:name="android.intent.action.MAIN" />

                 <category android:name="android.intent.category.LAUNCHER" />
             </intent-filter>
         </activity>
    </application>

</manifest>
```

The `Activity` base class defines a series of events that govern the life cycle of an activity. Figure 3-1 shows the lifecycle of an `Activity`.

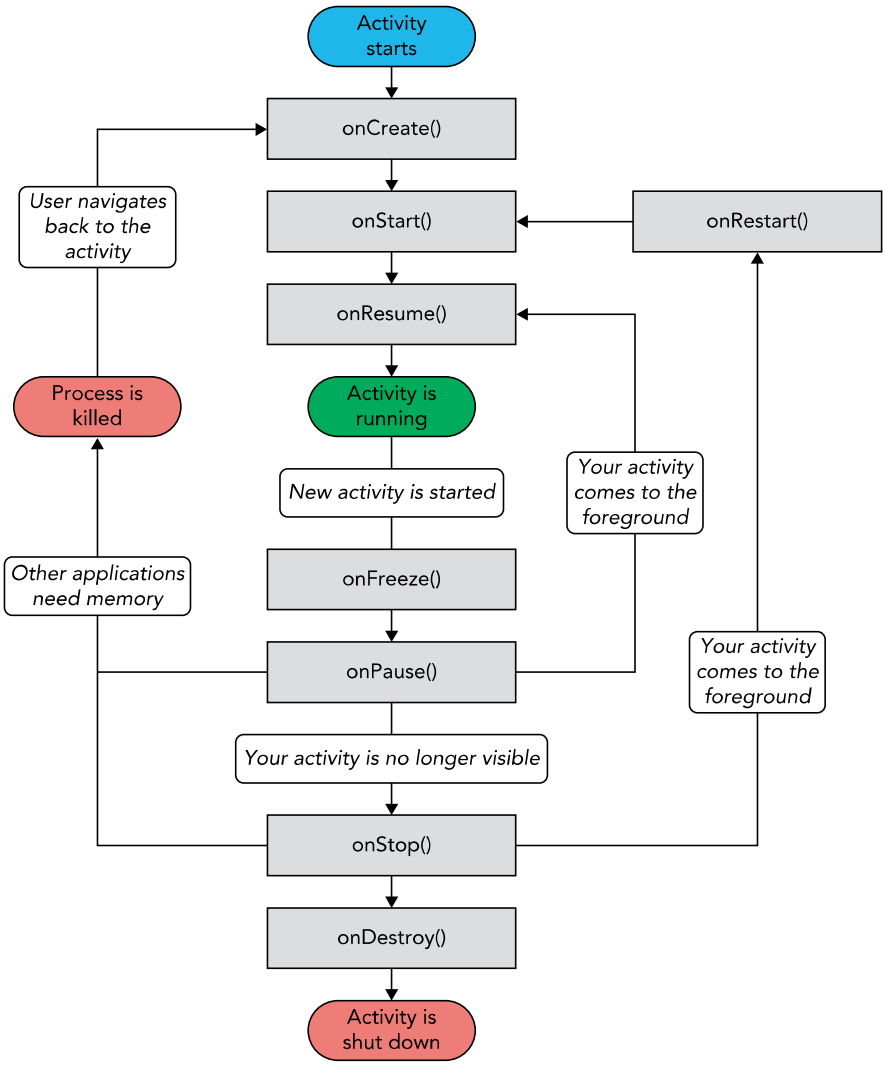

FIGURE 3-1

The `Activity` class defines the following events:

- `onCreate()`—Called when the activity is first created
- `onStart()`—Called when the activity becomes visible to the user
- `onResume()`—Called when the activity starts interacting with the user
- `onPause()`—Called when the current activity is being paused and the previous activity is being resumed
- `onStop()`—Called when the activity is no longer visible to the user

- `onDestroy()`—Called before the activity is destroyed by the system (either manually or by the system to conserve memory)
- `onRestart()`—Called when the activity has been stopped and is restarting again

By default, the activity created for you contains the `onCreate()` event. Within this event handler is the code that helps to display the UI elements of your screen.

Figure 3-2 shows the life cycle of an activity and the various stages it goes through—from when the activity is started until it ends.

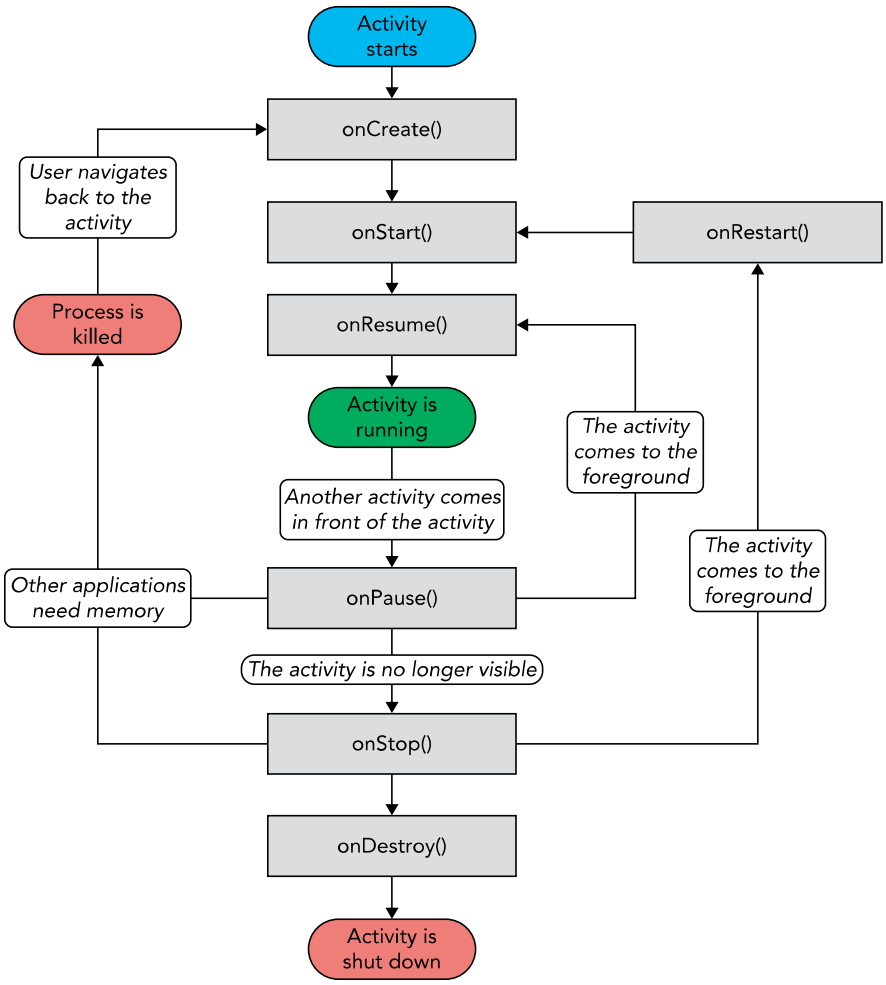

FIGURE 3-2

The best way to understand the various stages of an activity is to create a new project, implement the various events, and then subject the activity to various user interactions.

TRY IT OUT Understanding the Life Cycle of an Activity (Activity101.zip)

1. Using Android Studio, create a new Android project and name it **Activity101**.
2. In the `Activity101Activity.java` file, add the following highlighted statements. (Please note: Throughout this example, be sure to change all references to `"com.jfdimarzio"` to whatever package name your project is using.)

```
package com.jfdimarzio.activity101;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.Log;

public class MainActivity extends AppCompatActivity
{
    String tag = "Lifecycle Step";
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Log.d(tag, "In the onCreate() event");
    }

    public void onStart()
    {
        super.onStart();
        Log.d(tag, "In the onStart() event");
    }

    public void onRestart()
    {
        super.onRestart();
        Log.d(tag, "In the onRestart() event");
    }

    public void onResume()
    {
        super.onResume();
        Log.d(tag, "In the onResume() event");
    }

    public void onPause()
    {
        super.onPause();
        Log.d(tag, "In the onPause() event");
    }
```

```
    public void onStop()
    {
        super.onStop();
        Log.d(tag, "In the onStop() event");
    }

    public void onDestroy()
    {
        super.onDestroy();
        Log.d(tag, "In the onDestroy() event");
    }
}
```

3. Press Shift+F9 to debug the application, or select Run ➪ Debug. Then select one of your Android Virtual Devices from the pop-up window.

4. When the activity is first loaded, you should see something very similar to the following in the logcat console (see Figure 3-3). If you do not see the logcat console, click Android Monitor at the bottom of the Android Studio window:

```
11-16 06:25:59.396: D/Lifecycle Step(559): In the onCreate() event
11-16 06:25:59.396: D/Lifecycle Step(559): In the onStart() event
11-16 06:25:59.396: D/Lifecycle Step(559): In the onResume() event
```

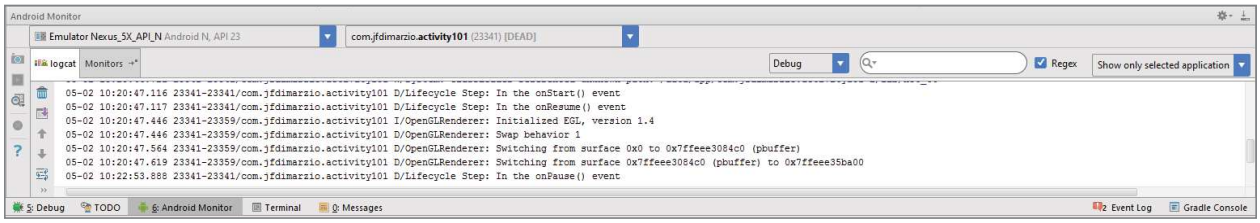

FIGURE 3-3

5. If you click the Back button on the Android emulator, you see the following:

```
11-16 06:29:26.665: D/Lifecycle Step(559): In the onPause() event
11-16 06:29:28.465: D/Lifecycle Step(559): In the onStop() event
11-16 06:29:28.465: D/Lifecycle Step(559): In the onDestroy() event
```

6. Click the Home button, click the `Overview` icon, select the Activity101 application, and observe the following:

```
11-16 06:31:08.905: D/Lifecycle Step(559): In the onCreate() event
11-16 06:31:08.905: D/Lifecycle Step(559): In the onStart() event
11-16 06:31:08.925: D/Lifecycle Step(559): In the onResume() event
```

7. Click the Home button and then click the Phone button on the Android emulator so that the activity is pushed to the background. Observe the output in the logcat window:

```
11-16 06:32:00.585: D/Lifecycle Step(559): In the onPause() event
11-16 06:32:05.015: D/Lifecycle Step(559): In the onStop() event
```

8. Notice that the `onDestroy()` event is not called, indicating that the activity is still in memory. Exit the phone dialer by clicking the Back button. The activity is now visible again. Observe the output in the logcat window:

```
11-16 06:32:50.515: D/Lifecycle(559): In the onRestart() event
11-16 06:32:50.515: D/Lifecycle(559): In the onStart() event
11-16 06:32:50.515: D/Lifecycle(559): In the onResume() event
```

The `onRestart()` event is now fired, followed by the `onStart()` and `onResume()` methods.

How It Works

As you can see from this simple example, an activity is destroyed when you click the Back button. This is crucial to understand because whatever state the activity is currently in will be lost. This means you need to write additional code in your activity to preserve its state when the activity is destroyed (Chapter 4 shows you how). At this point, note that the `onPause()` method is called in both scenarios:

- When an activity is sent to the background
- When a user kills an activity by tapping the Back button

When an activity is started, the `onStart()` and `onResume()` methods are always called, regardless of whether the activity is restored from the background or newly created. When an activity is created for the first time, the `onCreate()` method is called.

From the preceding example, you can derive the following guidelines:

- Use the `onCreate()` method to create and instantiate the objects that you will be using in your application.
- Use the `onResume()` method to start any services or code that needs to run while your activity is in the foreground.
- Use the `onPause()` method to stop any services or code that does not need to run when your activity is not in the foreground.
- Use the `onDestroy()` method to free up resources before your activity is destroyed.

> ***NOTE*** *Even if an application has only one activity and the activity is killed, the application is still running in memory.*

Applying Styles and Themes to an Activity

By default, an activity is themed to the default Android theme. However, there has been a push in recent years to adopt a new theme known as Material. The Material theme has a much more modern and clean look to it.

There are two versions of the Material theme available to Android developers: Material Light and Material Dark. Either of these themes can be applied from the `AndroidManifest.xml`.

To apply one of the Material themes to an activity, simply modify the `<Application>` element in the `AndroidManifest.xml` file by changing the default `android:theme` attribute. (Please be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.jfdimarzio.activity101">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@android:style/Theme.Material">
        <activity android:name=".MainActivity">

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

Changing the default theme to `@android:style/Theme.Material`, as in the highlighted code in the preceding snippet, applies the Material Dark theme and gives your application a darker look as shown in Figure 3-4.

Hiding the Activity Title

You can also hide the title of an activity if desired (such as when you just want to display a status update to the user). To do so, use the `requestWindowFeature()` method and pass it the `Window.FEATURE_NO_TITLE` constant, like this:

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.Window;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        requestWindowFeature(Window.FEATURE_NO_TITLE);
    }

}
```

Now you need to change the theme in the `AndroidManifest.xml` to a theme that has no title bar. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
package com.jfdimarzio.activity101;

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.jfdimarzio.activity101">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@android:style/Theme.NoTitleBar">
        <activity android:name=".MainActivity">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

FIGURE 3-4

This hides the title bar, as shown in Figure 3-5.

FIGURE 3-5

Displaying a Dialog Window

There are times when you need to display a dialog window to get a confirmation from the user. In this case, you can override the `onCreateDialog()` protected method defined in the `Activity` base class to display a dialog window. The following Try It Out shows you how.

TRY IT OUT Displaying a Dialog Window Using an Activity (Dialog.zip)

1. Using Android Studio, create a new Android project and name it Dialog. When presented with the option, name the main activity `DialogActivity`.
2. Add the following theme in bold to the `AndroidManifest.xml` file. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.jfdimarzio.dialog" >
```

```
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".DialogActivity"
            android:label="@string/app_name"
            android:theme="@style/Theme.AppCompat.Dialog" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

3. Compare your `DialogActivity.java` file to this:

```
package com.jfdimarzio.dialog;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;

public class DialogActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_dialog);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

       FloatingActionButton fab = (FloatingActionButton) findViewById(R.id.fab);
        fab.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                Snackbar.make(view, "Replace with your own action",
                  Snackbar.LENGTH_LONG)
                        .setAction("Action", null).show();
            }
        });
    }

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
```

```
        getMenuInflater().inflate(R.menu.menu_dialog, menu);
        return true;
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        // Handle action bar item clicks here. The action bar will
        // automatically handle clicks on the Home/Up button, so long
        // as you specify a parent activity in AndroidManifest.xml.
        int id = item.getItemId();

        //noinspection SimplifiableIfStatement
        if (id == R.id.action_settings) {
            return true;
        }

        return super.onOptionsItemSelected(item);
    }
}
```

4. Press Shift+F9 to debug the application on the Android emulator. Click the button to display the dialog (see Figure 3-6).

FIGURE 3-6

How It Works

Android uses the `AppCompat.Dialog` theme to draw your standard activity as a free-floating dialog box. It would be very easy to modify this dialog to add some buttons if you needed to provide an OK or Cancel choice.

Notice also that the theme is applied to the Activity, not the project. Therefore, you could have a project with multiple activities, and apply the dialog theme to just one of them.

Displaying a Progress Dialog

One common UI feature in an Android device is the "Please wait" dialog that you typically see when an application is performing a long-running task. For example, the application might be logging in to a server before the user is allowed to use it, or it might be doing a calculation before displaying the result to the user. In such cases, it is helpful to display a dialog, known as a *progress dialog*, so that the user is kept in the loop.

Android provides a `ProgressDialog` class you can call when you want to display a running meter to the user. `ProgressDialog` is easy to call from an activity.

The following Try It Out demonstrates how to display such a dialog.

TRY IT OUT Displaying a Progress (Please Wait) Dialog

1. Using the Activity101 project created earlier in this chapter, make sure you are using the Material theme in the `AndroidManifest.xml` file. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    package="com.jfdimarzio.activity101">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@android:style/Theme.Material">
        <activity android:name=".MainActivity">

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

2. Add the bolded statements from the following code to the `MainActivity.java` file:

```
package com.jfdimarzio.activity101;
import android.app.Activity;
import android.app.ProgressDialog;
import android.os.CountDownTimer;
import android.os.Bundle;


public class MainActivity extends Activity {

    ProgressDialog progressDialog;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

    }

    public void onStart()
    {
        super.onStart();
        progressDialog = ProgressDialog.show(this,"Please Wait",
           "Processing...",true);
        CountDownTimer timer = new CountDownTimer(3000,1000) {
            @Override
            public void onTick(long millisUntilFinished) {

            }

            @Override
            public void onFinish() {
                progressDialog.dismiss();
            }
        }.start();
    }
}
```

3. Press Shift+F9 to debug the application on the Android emulator. You see the progress dialog, as shown in Figure 3-7. It disappears after three seconds.

How It Works

To create a progress dialog, you create an instance of the `ProgressDialog` class and call its `show()` method:

```
progressDialog = ProgressDialog.show(this,"Please Wait", "Processing...",true);
```

This displays the progress dialog shown in Figure 3-7. Because this is a modal dialog, it will block the UI until it is dismissed. To close the dialog, you create a timer that calls the `dismiss()` method after three seconds. (Chapter 12 covers threads and calling methods that can do extraneous work from a thread while a progress dialog is displayed.)

```
CountDownTimer timer = new CountDownTimer(3000,1000) {
            @Override
            public void onTick(long millisUntilFinished) {
```

```
        }

        @Override
        public void onFinish() {
            progressDialog.dismiss();
        }
    }.start();
```

After the three seconds have elapsed, you dismiss the dialog by calling the `dismiss()` method.

FIGURE 3-7

The next section explains using Intents, which help you navigate between multiple Activities.

LINKING ACTIVITIES USING INTENTS

An Android application can contain zero or more activities. When your application has more than one activity, you often need to navigate from one to another. In Android, you navigate between activities through what is known as an intent.

The best way to understand this very important but somewhat abstract concept is to experience it firsthand and see what it helps you achieve. The following Try It Out shows how to add another activity to an existing project and then navigate between the two activities.

TRY IT OUT Linking Activities with Intents (UsingIntent.zip)

1. Using Android Studio, create a new Android project with an empty Activity named `MainActivity`; name the project **UsingIntent**.
2. Right-click your package name under the `app>>app>>src>>main>>java` folder in the Project Files windows and select New ➪ Java Class
3. Name the new class **SecondActivity** and click OK.
4. Add the bolded statements from the following code to the `AndroidManifest.xml` file. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.jfdimarzio.usingintent">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".SecondActivity" >
         <intent-filter >
             <action android:name="com.jfdimarzio.usingintent.SecondActivity" />
              <category android:name="android.intent.category.DEFAULT" />
         </intent-filter>
     </activity>
    </application>

</manifest>
```

5. Make a copy of the `activity_main.xml` file (in the `res/layout` folder) by right-clicking it and selecting Copy. Then right-click the `res/layout` folder and select Paste. Name the file activity_second.xml.
6. Modify the `activity_second.xml` file as follows:

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

```
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.usingintent.SecondActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is the Second Activity!" />
</RelativeLayout>
```

7. In the `SecondActivity.java` file, add the bolded statements from the following code:

```
package com.jfdimarzio.usingintent;

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

public class SecondActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }

}
```

8. Add the bolded lines in the following code to the `activity_main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.usingintent.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Main Activity!"
        android:id="@+id/textView" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Display second activity"
        android:onClick="onClick"
        android:id="@+id/button"
        android:layout_below="@+id/textView"
        android:layout_alignParentStart="true"
        android:layout_marginTop="56dp" />
</RelativeLayout>
```

9. Modify the `MainActivity.java` file as shown in the bolded lines in the following code:

```
package com.jfdimarzio.usingintent;

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

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClick(View view) {
        startActivity(new Intent("com.jfdimarzio.usingintent.SecondActivity"));
    }

}
```

10. Press Shift+F9 to debug the application on the Android emulator. When the first activity is loaded, click the button and the second activity also loads (see Figures 3-8 and 3-9).

How It Works

As previously described, an activity is made up of a UI component (for example, `activity_main.xml`) and a class component (for example, `MainActivity.java`). If you want to add another activity to a project, you need to create these two components.

Specifically, you need to add the following to the `AndroidManifest.xml` file:

```
</activity>
        <activity android:name=".SecondActivity" >
        <intent-filter >
            <action android:name="com.jfdimarzio.usingintent.SecondActivity" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
</activity>
```

When you add a new activity to the application, be sure to note the following:

- The name (class) of the new activity is `SecondActivity`.
- The intent filter name for the new activity is `<Your Package Name>.SecondActivity`. Other activities that want to call this activity invoke it via this name. Ideally, you should use the reverse domain name of your company as the intent filter name to reduce the chances of another application having the same intent filter name.
- The category for the intent filter is `android.intent.category.DEFAULT`. You need to add this to the intent filter so that this activity can be started by another activity using the `startActivity()` method (more on this shortly).

FIGURE 3-8

FIGURE 3-9

When the Display Second Activity button is clicked, you use the `startActivity()` method to display `SecondActivity` by creating an instance of the `Intent` class and passing it the intent filter name of `SecondActivity` (`net.learn2develop.SecondActivity`):

```
public void onClick(View view) {
    startActivity(new Intent("net.learn2develop.SecondActivity"));
}
```

Activities in Android can be invoked by any application running on the device. For example, you can create a new Android project and then display `SecondActivity` by using its `net.learn2develop.SecondActivity` intent filter. This is one of the fundamental concepts in Android that enables an application to easily invoke another application.

If the activity you want to invoke is defined within the same project, you can rewrite the preceding statement like this:

```
startActivity(new Intent(this, SecondActivity.class));
```

However, this approach is applicable only when the activity you want to display is within the same project as the current activity.

Returning Results from an Intent

The `startActivity()` method invokes another activity but does not return a result to the current activity. For example, you might have an activity that prompts the user for username and password. The information entered by the user in that activity needs to be passed back to the calling activity for further processing. If you need to pass data back from an activity, you should instead use the `startActivityForResult()` method. The following Try It Out demonstrates this.

TRY IT OUT Obtaining a Result from an Activity

1. Using the same project from the previous section, modify the `secondactivity.xml` file to look like the following code. Please be sure to change all references from `"com.jfdimarzio"` to whatever package name your project is using:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.usingintent.SecondActivity">


    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="This is the Second Activity!"
        android:id="@+id/textView2" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Please enter your name"
        android:id="@+id/textView3" />

    <EditText
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:id="@+id/txtUsername" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="OK"
        android:onClick="onClick"
        android:id="@+id/button2" />
</LinearLayout>
```

2. Add the bolded statements in the following code to `SecondActivity.java`:

```
package com.jfdimarzio.usingintent;

import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class SecondActivity extends Activity {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
    }
    public void onClick(View view) {
         Intent data = new Intent();
         //---get the EditText view---
         EditText txt_username = (EditText)findViewById(R.id.txtUsername);
         //---set the data to pass back---
         data.setData(Uri.parse( txt_username.getText().toString()));
         setResult(RESULT_OK, data);
         //---closes the activity---
         finish();
    }
}
```

3. Add the bolded statements in the following code to the `MainActivity.java` file:

```
package com.jfdimarzio.usingintent;

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

public class MainActivity extends Activity {
    int request_Code = 1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);
    }
    public void onClick(View view) {
        startActivityForResult(new Intent("com.jfdimarzio.usingintent.
          SecondActivity"),request_Code);
    }
    public void onActivityResult(int requestCode, int resultCode, Intent data)
    {
        if (requestCode == request_Code) {
            if (resultCode == RESULT_OK) {
                Toast.makeText(this,data.getData().toString(),
                   Toast.LENGTH_SHORT).show();
            }
        }
    }

}
```

4. Press Shift+F9 to debug the application on the Android emulator. When the first activity is loaded, click the button to load `SecondActivity`. Enter your name (see Figures 3-10, 3-11, and 3-12) and click the OK button. The first activity displays the name you have entered using the `Toast` class.

FIGURE 3-10

How It Works

To call an activity and wait for a result to be returned from it, you need to use the `startActivityForResult()` method, like this:

```
startActivityForResult(new Intent("com.jfdimarzio.usingintent.
  SecondActivity"),request_Code);
```

In addition to passing in an `Intent` object, you need to pass in a request code as well. The request code is simply an integer value that identifies an activity you are calling. This is needed because when an activity returns a value, you must have a way to identify it. For example, you might be calling multiple activities at the same time, though some activities might not return immediately (for example, waiting for a reply from a server). When an activity returns, you need this request code to determine which activity is actually returned.

FIGURE 3-11

FIGURE 3-12

> ***NOTE*** *If the request code is set to –1 then calling it using the* `startActivityForResult()` *method is equivalent to calling it using the* `startActivity()` *method. That is, no result is returned.*

In order for an activity to return a value to the calling activity, you use an `Intent` object to send data back via the `setData()` method:

```
Intent data = new Intent();
//---get the EditText view---
EditText txt_username =
    (EditText) findViewById(R.id.txt_username);
//---set the data to pass back---
data.setData(Uri.parse(
    txt_username.getText().toString()));
setResult(RESULT_OK, data);
//---closes the activity---
finish();
```

The `setResult()` method sets a result code (either `RESULT_OK` or `RESULT_CANCELLED`) and the data (an `Intent` object) to be returned back to the calling activity. The `finish()` method closes the activity and returns control to the calling activity.

In the calling activity, you need to implement the `onActivityResult()` method, which is called whenever an activity returns:

```
public void onActivityResult(int requestCode, int resultCode,
Intent data)
{
    if (requestCode == request_Code) {
        if (resultCode == RESULT_OK) {
            Toast.makeText(this,data.getData().toString(),
                Toast.LENGTH_SHORT).show();
        }
    }
}
```

Here, you check for the appropriate request and result codes and display the result that is returned. The returned result is passed in via the `data` argument; and you obtain its details through the `getData()` method.

Passing Data Using an Intent Object

Besides returning data from an activity, it is also common to pass data to an activity. For example, in the previous example, you might want to set some default text in the `EditText` view before the activity is displayed. In this case, you can use the `Intent` object to pass the data to the target activity.

The following Try It Out shows you the various ways in which you can pass data between activities.

TRY IT OUT Passing Data to the Target Activity

1. Using Eclipse, create a new Android project and name it **PassingData**.
2. Add the bolded statements in the following code to the `activity_main.xml` file. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.passingdata.MainActivity">


    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click to go to Second Activity"
        android:id="@+id/button"
        android:onClick="onClick"/>
</LinearLayout>
```

3. Add a new XML file to the `res/layout` folder and name it **activity_second.xml**. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.passingdata.MainActivity">


    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Welcome to the Second Activity"
        android:id="@+id/textView" />

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click to go to Main Activity"
        android:id="@+id/button"
        android:onClick="onClick"/>
</LinearLayout>
```

4. Add a new `Class` file to the package and name it **SecondActivity**. Populate the `SecondActivity.java` file as follows:

```
package com.jfdimarzio.passingdata;
```

```
import android.app.Activity;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class SecondActvity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_second);
        //---get the data passed in using getStringExtra()---
        Toast.makeText(this,getIntent().getStringExtra("str1"),
                Toast.LENGTH_SHORT).show();
        //---get the data passed in using getIntExtra()---
        Toast.makeText(this,Integer.toString(
                getIntent().getIntExtra("age1", 0)),
                Toast.LENGTH_SHORT).show();
        //---get the Bundle object passed in---
        Bundle bundle = getIntent().getExtras();
        //---get the data using the getString()---
        Toast.makeText(this, bundle.getString("str2"),
                Toast.LENGTH_SHORT).show();
        //---get the data using the getInt() method---
        Toast.makeText(this,Integer.toString(bundle.getInt("age2")),
                Toast.LENGTH_SHORT).show();
    }
    public void onClick(View view) {
        //---use an Intent object to return data---
        Intent i = new Intent();
        //---use the putExtra() method to return some
        // value---
        i.putExtra("age3", 45);
        //---use the setData() method to return some value---
        i.setData(Uri.parse("Something passed back to main activity"));
        //---set the result with OK and the Intent object---
        setResult(RESULT_OK, i);
        //---destroy the current activity---
        finish();
    }

}
```

5. Add the bolded statements from the following code to the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.jfdimarzio.passingdata">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
```

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

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".SecondActvity" >
         <intent-filter >
             <action android:name="com.jfdimarzio.passingdata.SecondActivity" />
              <category android:name="android.intent.category.DEFAULT" />
         </intent-filter>

    </activity>
    </application>

</manifest>
```

6. Add the bolded statements from the following code to the `MainActivity.java` file:

```
package com.jfdimarzio.passingdata;

import android.content.Intent;
import android.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class MainActivity extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClick(View view) {
         Intent i = new
                 Intent("com.jfdimarzio.passingdata.SecondActivity");
         //---use putExtra() to add new name/value pairs---
         i.putExtra("str1", "This is a string");
         i.putExtra("age1", 25);
         //---use a Bundle object to add new name/values
         // pairs---
         Bundle extras = new Bundle();
         extras.putString("str2", "This is another string");
         extras.putInt("age2", 35);
         //---attach the Bundle object to the Intent object---
         i.putExtras(extras);
         //---start the activity to get a result back---
         startActivityForResult(i, 1);
    }

    public void onActivityResult(int requestCode,
                                 int resultCode, Intent data)
```

```
    {
        //---check if the request code is 1---
        if (requestCode == 1) {
            //---if the result is OK---
            if (resultCode == RESULT_OK) {
                //---get the result using getIntExtra()---
                Toast.makeText(this, Integer.toString(
                        data.getIntExtra("age3", 0)),
                        Toast.LENGTH_SHORT).show();
                //---get the result using getData()---
                Toast.makeText(this, data.getData().toString(),
                        Toast.LENGTH_SHORT).show();
            }
        }
    }

}
```

7. Press Shift+F9 to debug the application on the Android emulator. Click the button on each activity and observe the values displayed.

How It Works

While this application is not visually exciting, it does illustrate some important ways to pass data between activities.

First, you can use the `putExtra()` method of an `Intent` object to add a name/value pair:

```
//---use putExtra() to add new name/value pairs---
i.putExtra("str1", "This is a string");
i.putExtra("age1", 25);
```

The preceding statements add two name/value pairs to the `Intent` object: one of type `string` and one of type `integer`.

Besides using the `putExtra()` method, you can also create a `Bundle` object and then attach it using the `putExtras()` method. Think of a `Bundle` object as a dictionary object—it contains a set of name/value pairs. The following statements create a `Bundle` object and then add two name/value pairs to it. The `Bundle` object is then attached to the `Intent` object:

```
//---use a Bundle object to add new name/values pairs---
Bundle extras = new Bundle();
extras.putString("str2", "This is another string");
extras.putInt("age2", 35);
//---attach the Bundle object to the Intent object---
i.putExtras(extras);
```

To obtain the data sent using the `Intent` object, you first obtain the `Intent` object using the `getIntent()` method. Then, call its `getStringExtra()` method to get the string value set using the `putExtra()` method:

```
//---get the data passed in using getStringExtra()---
Toast.makeText(this,getIntent().getStringExtra("str1"),
        Toast.LENGTH_SHORT).show();
```

In this case, you have to call the appropriate method to extract the name/value pair based on the type of data set. For the integer value, use the `getIntExtra()` method (the second argument is the default value in case no value is stored in the specified name):

```
//---get the data passed in using getIntExtra()---
Toast.makeText(this,Integer.toString(
        getIntent().getIntExtra("age1", 0)),
        Toast.LENGTH_SHORT).show();
```

To retrieve the `Bundle` object, use the `getExtras()` method:

```
//---get the Bundle object passed in---
Bundle bundle = getIntent().getExtras();
```

To get the individual name/value pairs, use the appropriate method. For the string value, use the `getString()` method:

```
//---get the data using the getString()---
Toast.makeText(this, bundle.getString("str2"),
        Toast.LENGTH_SHORT).show();
```

Likewise, use the `getInt()` method to retrieve an integer value:

```
//---get the data using the getInt() method---
Toast.makeText(this,Integer.toString(bundle.getInt("age2")),
        Toast.LENGTH_SHORT).show();
```

Another way to pass data to an activity is to use the `setData()` method (as used in the previous section), like this:

```
//---use the setData() method to return some value---
i.setData(Uri.parse(
        "Something passed back to main activity"));
```

Usually, you use the `setData()` method to set the data on which an `Intent` object is going to operate, such as passing a URL to an `Intent` object so that it can invoke a web browser to view a web page. (For more examples, see the section "Calling Built-In Applications Using Intents," later in this chapter.)

To retrieve the data set using the `setData()` method, use the `getData()` method (in this example `data` is an `Intent` object):

```
//---get the result using getData()---
Toast.makeText(this, data.getData().toString(),
    Toast.LENGTH_SHORT).show();
```

FRAGMENTS

In the previous section, you learned what an activity is and how to use it. In a small-screen device (such as a smartphone), an activity typically fills the entire screen, displaying the various views that make up the user interface of an application. The activity is essentially a container for views. However, when an activity is displayed in a large-screen device, such as on a tablet, it is somewhat

out of place. Because the screen is much bigger, all the views in an activity must be arranged to make full use of the increased space, resulting in complex changes to the view hierarchy. A better approach is to have "mini-activities," each containing its own set of views. During runtime, an activity can contain one or more of these mini-activities, depending on the screen orientation in which the device is held. In Android 3.0 and later, these mini-activities are known as *fragments*.

Think of a fragment as another form of activity. You create fragments to contain views, just like activities. Fragments are always embedded in an activity. For example, Figure 3-13 shows two fragments. Fragment 1 might contain a `ListView` showing a list of book titles. Fragment 2 might contain some `TextViews` and `ImageViews` showing some text and images.

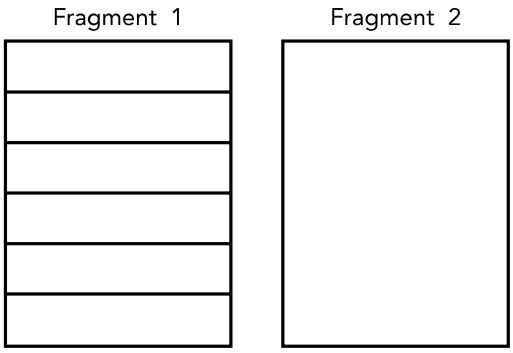

FIGURE 3-13

Now imagine the application is running on an Android tablet (or on an Android smartphone) in portrait mode. In this case, Fragment 1 might be embedded in one activity, whereas Fragment 2 might be embedded in another activity (see Figure 3-14). When users select an item in the list in Fragment 1, Activity 2 is started.

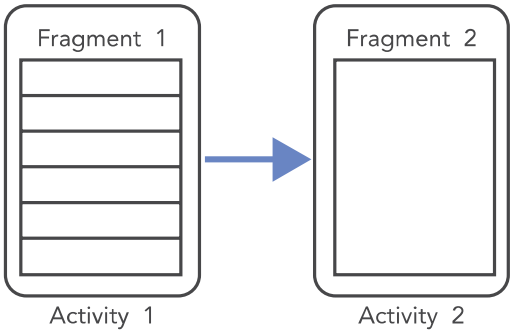

FIGURE 3-14

If the application is now displayed in a tablet in landscape mode, both fragments can be embedded within a single activity, as shown in Figure 3-15.

From this discussion, it becomes apparent that fragments present a versatile way in which you can create the user interface of an Android application. Fragments form the atomic unit of your user interface, and they can be dynamically added (or removed) to activities in order to create the best user experience possible for the target device.

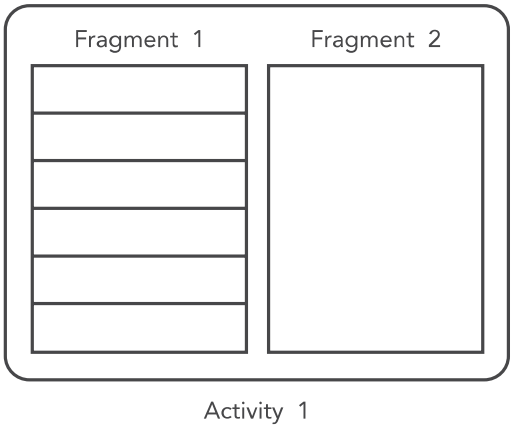

FIGURE 3-15

The following Try It Out shows you the basics of working with fragments.

TRY IT OUT Using Fragments (Fragments.zip)

1. Using Android Studio, create a new Android project and name it **Fragments**.
2. In the `res/layout` folder, add a new layout resource file and name it `fragment1.xml`. Populate it with the following code. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#00FF00"
    >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="This is fragment #1"
    android:textColor="#000000"
    android:textSize="25sp" />
</LinearLayout>
```

3. Also in the `res/layout` folder, add another new layout resource file and name it `fragment2.xml`. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#FFFE00"
    >
```

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="This is fragment #2"
    android:textColor="#000000"
    android:textSize="25sp" />
</LinearLayout>
```

4. In `activity_main.xml`, add the bolded lines in the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.fragments.MainActivity">

    <fragment
        android:name="com.jfdimarzio.fragments.Fragment1"
        android:id="@+id/fragment1"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
    <fragment
        android:name="com.jfdimarzio.fragments.Fragment2"
        android:id="@+id/fragment2"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

5. Under the `<Your Package Name>/fragments` package name, add two Java class files and name them `Fragment1.java` and `Fragment2.java`.

6. Add the following code to `Fragment1.java`:

```
package com.jfdimarzio.fragments;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
public class Fragment1 extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container, Bundle savedInstanceState) {
        //---Inflate the layout for this fragment---
        return inflater.inflate(
                R.layout.fragment1, container, false);
    }
}
```

7. Add the following code to `Fragment2.java`:

```
package com.jfdimarzio.fragments;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
public class Fragment2 extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container, Bundle savedInstanceState) {
        //---Inflate the layout for this fragment---
        return inflater.inflate(
                R.layout.fragment2, container, false);
    }
}
```

8. Press Shift+F9 to debug the application on the Android emulator. Figure 3-16 shows the two fragments contained within the activity.

FIGURE 3-16

How It Works

A fragment behaves very much like an activity: It has a Java class and it loads its UI from an XML file. The XML file contains all the usual UI elements that you expect from an activity: `TextView`, `EditText`, `Button`, and so on. The Java class for a fragment needs to extend the `Fragment` base class:

```
public class Fragment1 extends Fragment {
}
```

> ***NOTE*** *Besides the* `Fragment` *base class, a fragment can also extend a few other subclasses of the* `Fragment` *class, such as* `DialogFragment`, `ListFragment`, *and* `PreferenceFragment`. *Chapter 6 discusses these types of fragments in more detail.*

To draw the UI for a fragment, you override the `onCreateView()` method. This method needs to return a `View` object, like this:

```
    public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
        //---Inflate the layout for this fragment---
        return inflater.inflate(
            R.layout.fragment1, container, false);
    }
Here, you use a LayoutInflater object to inflate the UI from the
   specified XML file (R.layout.fragment1 in this case). The
   container argument refers to the parent ViewGroup, which
   is the activity in which you are trying to embed the
   fragment. The savedInstanceState argument enables you
   to restore the fragment to its previously saved state.
To add a fragment to an activity, you use the <fragment> element:
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res
   /android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <fragment
        android:name=" com.jfdimarzio.fragments.Fragment1"
        android:id="@+id/fragment1"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
    <fragment
        android:name=" com.jfdimarzio.fragments.Fragment2"
        android:id="@+id/fragment2"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

Note that each fragment needs a unique identifier. You can assign one via the `android:id` or `android:tag` attribute.

Adding Fragments Dynamically

Although fragments enable you to compartmentalize your UI into various configurable parts, the real power of fragments is realized when you add them dynamically to activities during runtime. In the previous section, you saw how you can add fragments to an activity by modifying the XML file during design time. In reality, it is much more useful if you create fragments and add them to activities during runtime. This enables you to create a customizable user interface for your application. For example, if the application is running on a smartphone, you might fill an activity with a single fragment; if the application is running on a tablet, you might then fill the activity with two or more fragments, as the tablet has much more screen real estate compared to a smartphone.

The following Try It Out shows how you can programmatically add fragments to an activity during runtime.

TRY IT OUT Adding Fragments During Runtime

1. Using the same project created in the previous section, modify the `main.xml` file by commenting out the two `<fragment>` elements. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.fragments.MainActivity">
<!--
    <fragment
        android:name="com.jfdimarzio.fragments.Fragment1"
        android:id="@+id/fragment1"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
    <fragment
        android:name="com.jfdimarzio.fragments.Fragment2"
        android:id="@+id/fragment2"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
-->
</LinearLayout>
```

2. Add the bolded lines in the following code to the `MainActivity.java` file:

```
package com.jfdimarzio.fragments;

import android.app.Activity;
import android.app.FragmentManager;
import android.app.FragmentTransaction;
import android.os.Bundle;
import android.util.DisplayMetrics;

public class MainActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction =
                fragmentManager.beginTransaction();
        //---get the current display info---
        DisplayMetrics display = this.getResources().getDisplayMetrics();

        int width = display.widthPixels;
        int height = display.heightPixels;
        if (width> height)
        {
            //---landscape mode---
            Fragment1 fragment1 = new Fragment1();
            // android.R.id.content refers to the content
            // view of the activity
            fragmentTransaction.replace(
                    android.R.id.content, fragment1);
        }
        else
        {
            //---portrait mode---
            Fragment2 fragment2 = new Fragment2();
            fragmentTransaction.replace(
                    android.R.id.content, fragment2);
        }
        fragmentTransaction.commit();
    }
}
```

3. Press Shift + F9 to run the application on the Android emulator. Observe that when the emulator is in portrait mode, Fragment 2 is displayed (see Figure 3-17). If you press Ctrl+Left to change the orientation of the emulator to landscape, Fragment 1 is shown instead (see Figure 3-18).

FIGURE 3-17

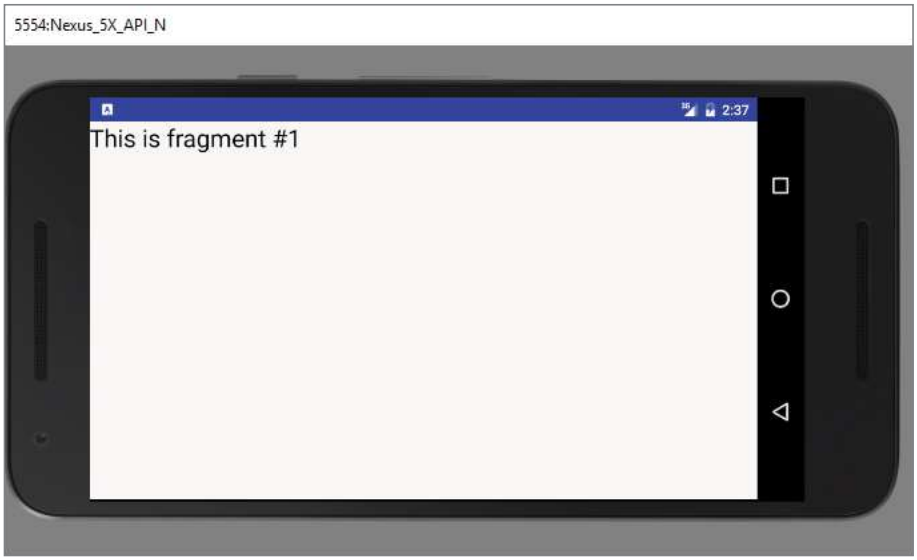

FIGURE 3-18

How It Works

To add fragments to an activity, you use the `FragmentManager` class by first obtaining an instance of it:

```
FragmentManager fragmentManager = getFragmentManager();
```

You also need to use the `FragmentTransaction` class to perform fragment transactions (such as add, remove, or replace) in your activity:

```
FragmentTransaction fragmentTransaction =
        fragmentManager.beginTransaction();
```

In this example, the `WindowManager` is used to determine whether the device is currently in portrait mode or landscape mode. Once that is determined, you can add the appropriate fragment to the activity by creating the fragment. Next, you call the `replace()` method of the `FragmentTransaction` object to add the fragment to the specified view container. In this case, `android.R.id.content` refers to the content view of the activity.

```
//---landscape mode---
Fragment1 fragment1 = new Fragment1();
// android.R.id.content refers to the content
// view of the activity
fragmentTransaction.replace(
        android.R.id.content, fragment1);
```

Using the `replace()` method is essentially the same as calling the `remove()` method followed by the `add()` method of the `FragmentTransaction` object. To ensure that the changes take effect, you need to call the `commit()` method:

```
fragmentTransaction.commit();
```

Life Cycle of a Fragment

Like activities, fragments have their own life cycle. Understanding the life cycle of a fragment enables you to properly save an instance of the fragment when it is destroyed, and restore it to its previous state when it is re-created.

The following Try It Out examines the various states experienced by a fragment.

TRY IT OUT Understanding the Life Cycle of a Fragment (Fragments.zip)

1. Using the same project created in the previous section, add the following bolded code to the `Fragment1.java` file. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
package com.jfdimarzio.fragments;
import android.app.Activity;
import android.app.Fragment;
import android.os.Bundle;
import android.util.Log;
```

```
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
public class Fragment1 extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {

        Log.d("Fragment 1", "onCreateView");

        //---Inflate the layout for this fragment---
        return inflater.inflate(
            R.layout.fragment1, container, false);
    }

    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        Log.d("Fragment 1", "onAttach");
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Log.d("Fragment 1", "onCreate");
    }
    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        Log.d("Fragment 1", "onActivityCreated");
    }
    @Override
    public void onStart() {
        super.onStart();
        Log.d("Fragment 1", "onStart");
    }
    @Override
    public void onResume() {
        super.onResume();
        Log.d("Fragment 1", "onResume");
    }
    @Override
    public void onPause() {
        super.onPause();
        Log.d("Fragment 1", "onPause");
    }

    @Override
    public void onStop() {
```

```
        super.onStop();
        Log.d("Fragment 1", "onStop");
    }

    @Override
    public void onDestroyView() {
        super.onDestroyView();
        Log.d("Fragment 1", "onDestroyView");
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d("Fragment 1", "onDestroy");
    }

    @Override
    public void onDetach() {
        super.onDetach();
        Log.d("Fragment 1", "onDetach");
    }

}
```

2. Switch the Android emulator to landscape mode by pressing Ctrl+Left.
3. Press Shift+F9 in Android Studio to debug the application on the Android emulator.
4. When the application is loaded on the emulator, the following is displayed in the logcat console in Android Monitor:

```
12-09 04:17:43.436: D/Fragment 1(2995): onAttach
12-09 04:17:43.466: D/Fragment 1(2995): onCreate
12-09 04:17:43.476: D/Fragment 1(2995): onCreateView
12-09 04:17:43.506: D/Fragment 1(2995): onActivityCreated
12-09 04:17:43.506: D/Fragment 1(2995): onStart
12-09 04:17:43.537: D/Fragment 1(2995): onResume
```

5. Click the Home button on the emulator. The following output is displayed in the logcat console:

```
12-09 04:18:47.696: D/Fragment 1(2995): onPause
12-09 04:18:50.346: D/Fragment 1(2995): onStop
```

6. On the emulator, click the Home button and hold it. Launch the application again. This time, the following is displayed:

```
12-09 04:20:08.726: D/Fragment 1(2995): onStart
12-09 04:20:08.766: D/Fragment 1(2995): onResume
```

7. Click the Back button on the emulator. Now you should see the following output:

```
12-09 04:21:01.426: D/Fragment 1(2995): onPause
12-09 04:21:02.346: D/Fragment 1(2995): onStop
```

```
12-09 04:21:02.346: D/Fragment 1(2995): onDestroyView
12-09 04:21:02.346: D/Fragment 1(2995): onDestroy
12-09 04:21:02.346: D/Fragment 1(2995): onDetach
```

How It Works

Like activities, fragments in Android also have their own life cycle. As you have seen, when a fragment is being created, it goes through the following states:

- `onAttach()`
- `onCreate()`
- `onCreateView()`
- `onActivityCreated()`

When the fragment becomes visible, it goes through these states:

- `onStart()`
- `onResume()`

When the fragment goes into the background mode, it goes through these states:

- `onPause()`
- `onStop()`

When the fragment is destroyed (when the activity in which it is currently hosted is destroyed), it goes through the following states:

- `onPause()`
- `onStop()`
- `onDestroyView()`
- `onDestroy()`
- `onDetach()`

Like activities, you can restore an instance of a fragment using a `Bundle` object, in the following states:

- `onCreate()`
- `onCreateView()`
- `onActivityCreated()`

Most of the states experienced by a fragment are similar to those of activities. However, a few new states are specific to fragments:

- `onAttached()`—Called when the fragment has been associated with the activity
- `onCreateView()` Called to create the view for the fragment
- `onActivityCreated()`—Called when the activity's `onCreate()` method has been returned

- `onDestroyView()`—Called when the fragment's view is being removed
- `onDetach()`—Called when the fragment is detached from the activity

One of the main differences between activities and fragments is when an activity goes into the background, the activity is placed in the back stack. This allows the activity to be resumed when the user presses the Back button. In the case of fragments, however, they are not automatically placed in the back stack when they go into the background. Rather, to place a fragment into the back stack, you need to explicitly call the `addToBackStack()` method during a fragment transaction, like this:

```
//---get the current display info---
DisplayMetrics display = this.getResources().getDisplayMetrics();

int width = display.widthPixels;
int height = display.heightPixels;
if (width> height)
{
    //---landscape mode---
    Fragment1 fragment1 = new Fragment1();
    // android.R.id.content refers to the content
    // view of the activity
    fragmentTransaction.replace(
            android.R.id.content, fragment1);
}
else
{
    //---portrait mode---
    Fragment2 fragment2 = new Fragment2();
    fragmentTransaction.replace(
            android.R.id.content, fragment2);
}
//---add to the back stack---
fragmentTransaction.addToBackStack(null);
fragmentTransaction.commit();
```

The preceding code ensures that after the fragment has been added to the activity, the user can click the Back button to remove it.

Interactions Between Fragments

Very often, an activity might contain one or more fragments working together to present a coherent UI to the user. In this case, it is important for fragments to communicate with one another and exchange data. For example, one fragment might contain a list of items (such as postings from an RSS feed). Also, when the user taps on an item in that fragment, details about the selected item might be displayed in another fragment.

The following Try It Out shows how one fragment can access the views contained within another fragment.

TRY IT OUT Communication Between Fragments

1. Using the same project created in the previous section, add the following bolded statement to the `Fragment1.xml` file. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#00FF00" >
<TextView
     android:id="@+id/lblFragment1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="This is fragment #1"
    android:textColor="#000000"
    android:textSize="25sp" />
</LinearLayout>
```

2. Add the following bolded lines to `fragment2.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:background="#FFFE00" >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="This is fragment #2"
    android:textColor="#000000"
    android:textSize="25sp" />
<Button
     android:id="@+id/btnGetText"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="Get text in Fragment #1"
     android:textColor="#000000"
     android:onClick="onClick" />
</LinearLayout>
```

3. Return the two fragments to `main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical"
    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"
    android:paddingBottom="@dimen/activity_vertical_margin"
```

```
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.fragments.MainActivity">
    <fragment
        android:name="com.jfdimarzio.fragments.Fragment1"
        android:id="@+id/fragment1"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
    <fragment
        android:name="com.jfdimarzio.fragments.Fragment2"
        android:id="@+id/fragment2"
        android:layout_weight="1"
        android:layout_width="fill_parent"
        android:layout_height="match_parent" />
</LinearLayout>
```

4. Modify the `MainActivity.java` file by commenting out the code that you added in the earlier sections. It should look like this after modification:

```
public class MainActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        /*
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction =
                fragmentManager.beginTransaction();
        //---get the current display info---
        DisplayMetrics display = this.getResources().getDisplayMetrics();

        int width = display.widthPixels;
        int height = display.heightPixels;
        if (width> height)
        {
            //---landscape mode---
            Fragment1 fragment1 = new Fragment1();
            // android.R.id.content refers to the content
            // view of the activity
            fragmentTransaction.replace(
                    android.R.id.content, fragment1);
        }
        else
        {
            //---portrait mode---
            Fragment2 fragment2 = new Fragment2();
            fragmentTransaction.replace(
                    android.R.id.content, fragment2);
        }
        fragmentTransaction.commit();
        */
    }
}
```

5. Add the following bolded statements to the `Fragment2.java` file:

```
package com.jfdimarzio.fragments;
import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
public class Fragment2 extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater,
    ViewGroup container, Bundle savedInstanceState) {
        //---Inflate the layout for this fragment---
        return inflater.inflate(
            R.layout.fragment2, container, false);
    }

    @Override
    public void onStart() {
        super.onStart();
        //---Button view---
        Button btnGetText = (Button)
            getActivity().findViewById(R.id.btnGetText);
        btnGetText.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                TextView lbl = (TextView)
                    getActivity().findViewById(R.id.lblFragment1);
                Toast.makeText(getActivity(), lbl.getText(),
                    Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

6. Press Shift+F9 to debug the application on the Android emulator. In the second fragment on the right, click the button. You should see the `Toast` class displaying the text `This is fragment #1`.

How It Works

Because fragments are embedded within activities, you can obtain the activity in which a fragment is currently embedded by first using the `getActivity()` method and then using the `findViewById()` method to locate the view(s) contained within the fragment:

```
TextView lbl = (TextView)
    getActivity().findViewById(R.id.lblFragment1);
Toast.makeText(getActivity(), lbl.getText(),
    Toast.LENGTH_SHORT).show();
```

The `getActivity()` method returns the activity with which the current fragment is currently associated.

Alternatively, you can also add the following method to the `MainActivity.java` file:

```
public void onClick(View v) {
    TextView lbl = (TextView)
        findViewById(R.id.lblFragment1);
    Toast.makeText(this, lbl.getText(),
        Toast.LENGTH_SHORT).show();
}
```

Understanding the Intent Object

So far, you have seen the use of the `Intent` object to call other activities. This is a good time to recap and gain a more detailed understanding of how the `Intent` object performs its magic.

First, you learned that you can call another activity by passing its action to the constructor of an `Intent` object:

```
startActivity(new Intent("com.jfdimarzio.SecondActivity"));
```

The action (in this example `"com.jfdimarzio.SecondActivity"`) is also known as the *component name*. This is used to identify the target activity/application that you want to invoke. You can also rewrite the component name by specifying the class name of the activity if it resides in your project, like this:

```
startActivity(new Intent(this, SecondActivity.class));
```

You can also create an `Intent` object by passing in an action constant and data, such as the following:

```
Intent i = new
        Intent(android.content.Intent.ACTION_VIEW,
                Uri.parse("http://www.amazon.com"));
startActivity(i);
```

The action portion defines what you want to do, whereas the data portion contains the data for the target activity to act upon. You can also pass the data to the `Intent` object using the `setData()` method:

```
Intent i = new
        Intent("android.intent.action.VIEW");
i.setData(Uri.parse("http://www.amazon.com"));
```

In this example, you indicate that you want to view a web page with the specified URL. The Android OS will look for all activities that are able to satisfy your request. This process is

known as *intent resolution*. The next section discusses in more detail how your activities can be the target of other activities.

For some intents, there is no need to specify the data. For example, to select a contact from the Contacts application, you specify the action and then indicate the MIME type using the `setType()` method:

```
Intent i = new
    Intent(android.content.Intent.ACTION_PICK);
i.setType(ContactsContract.Contacts.CONTENT_TYPE);
```

> ***NOTE*** *Chapter 9 discusses how to use the Contacts application from within your application.*

The `setType()` method explicitly specifies the MIME data type to indicate the type of data to return. The MIME type for `ContactsContract.Contacts.CONTENT_TYPE` is `"vnd.android.cursor.dir/contact"`.

Besides specifying the action, the data, and the type, an `Intent` object can also specify a category. A category groups activities into logical units so that Android can use those activities for further filtering. The next section discusses categories in more detail.

To summarize, an `Intent` object can contain the following information:

- Action
- Data
- Type
- Category

Using Intent Filters

Earlier, you saw how an activity can invoke another activity using the `Intent` object. In order for other activities to invoke your activity, you need to specify the action and category within the `<intent-filter>` element in the `AndroidManifest.xml` file, like this:

```
<intent-filter >
       <action android:name="com.jfdimarzio.SecondActivity" />
       <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

This is a very simple example in which one activity calls another using the `"com.jfdimarzio.SecondActivity"` action.

DISPLAYING NOTIFICATIONS

So far, you have been using the `Toast` class to display messages to the user. While the `Toast` class is a handy way to show users alerts, it is not persistent. It flashes on the screen for a few seconds and then disappears. If it contains important information, users may easily miss it if they are not looking at the screen.

For messages that are important, you should use a more persistent method. In this case, you should use the `NotificationManager` to display a persistent message at the top of the device, commonly known as the *status bar* (sometimes also referred to as the *notification bar*). The following Try It Out demonstrates how.

TRY IT OUT Displaying Notifications on the Status Bar (Notifications.zip)

1. Using Android Studio, create a new Android project and name it **Notifications**.
2. Add a new class file named `NotificationView` to the package. In addition, add a new `notification.xml` layout resource file to the `res/layout` folder.
3. Populate the `notification.xml` file as follows. Be sure to change all instances of `"com.jfdimarzio"` to whatever package name your project is using)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Here are the details for the notification..." />
</LinearLayout>
```

4. Populate the `NotificationView.java` file as follows:

```
package com.jfdimarzio.notifications;

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

public class NotificationView extends Activity {

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

```
        //---look up the notification manager service---
        NotificationManager nm = (NotificationManager)
                getSystemService(NOTIFICATION_SERVICE);
        //---cancel the notification that we started---
        nm.cancel(getIntent().getExtras().getInt("notificationID"));
    }

}
```

5. Add the following statements in bold to the `AndroidManifest.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.jfdimarzio.notifications">
    <uses-permission android:name="android.permission.VIBRATE"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".NotificationView"
             android:label="Details of notification">
             <intent-filter>
                 <action android:name="android.intent.action.MAIN" />
                 <category android:name="android.intent.category.DEFAULT" />
             </intent-filter>
         </activity>
    </application>

</manifest>
```

6. Add the following statements in bold to the `activity_main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<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"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.jfdimarzio.notifications.MainActivity">

    <Button
        android:id="@+id/btn_displaynotif"
```

```
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Display Notification"
        android:onClick="onClick"/>
</RelativeLayout>
```

7. Add the following statements in bold to the `MainActivity.java` file:

```
package com.jfdimarzio.notifications;

import android.app.Activity;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.view.View;

public class MainActivity extends Activity {
    int notificationID = 1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClick(View view) {
        displayNotification();
    }

    protected void displayNotification()
    {
        //---PendingIntent to launch activity if the user selects
        // this notification---
        Intent i = new Intent(this, NotificationView.class);
        i.putExtra("notificationID", notificationID);
        PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, i, 0);
        NotificationManager nm = (NotificationManager)getSystemService
          (NOTIFICATION_SERVICE);
        NotificationCompat.Builder notifBuilder;
        notifBuilder = new NotificationCompat.Builder(this)
                .setSmallIcon(R.mipmap.ic_launcher)
                .setContentTitle("Meeting Reminder")
                .setContentText("Reminder: Meeting starts in 5 minutes");
        nm.notify(notificationID, notifBuilder.build());
    }
}
```

8. Press Shift+F9 to debug the application on the Android emulator.

9. Click the Display Notification button and a notification ticker text (set in the constructor of the `Notification` object) displays on the status bar.

10. Click and drag the status bar down to reveal the notification details set using the `setLatestEventInfo()` method of the `Notification` object (see Figure 3-19).

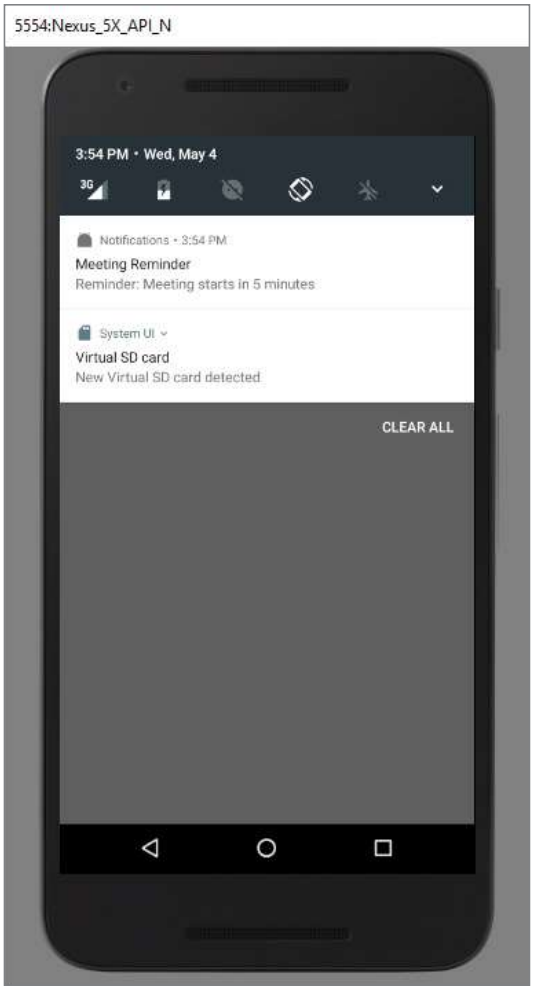

FIGURE 3-19

How It Works

To display a notification, you first created an `Intent` object to point to the `NotificationView` class:

```
Intent i = new Intent(this, NotificationView.class);
i.putExtra("notificationID", notificationID);
```

This intent is used to launch another activity when the user selects a notification from the list. In this example, you added a name/value pair to the `Intent` object so that you can tag the notification ID, identifying the notification to the target activity. Later, you will use this ID to dismiss the notification.

You also need to create a `PendingIntent` object. A `PendingIntent` object helps you to perform an action on your application's behalf, often at a later time, regardless of whether your application is running. In this case, you initialized it as follows:

```
PendingIntent pendingIntent =
    PendingIntent.getActivity(this, 0, i, 0);
```

The `getActivity()` method retrieves a `PendingIntent` object and you set it using the following arguments:

- **context**—Application context
- **request code**—Request code for the intent
- **intent**—The intent for launching the target activity
- **flags**—The flags in which the activity is to be launched

You then obtain an instance of the `NotificationManager` class and create an instance of the `NotificationCompat.Builder` class:

```
NotificationManager nm = (NotificationManager)getSystemService(NOTIFICATION
  _SERVICE);
 NotificationCompat.Builder notifBuilder;
 notifBuilder = new NotificationCompat.Builder(this)
         .setSmallIcon(R.mipmap.ic_launcher)
         .setContentTitle("Meeting Reminder")
         .setContentText("Reminder: Meeting starts in 5 minutes");
```

The `NotificationCompat.Builder` class enables you to specify the notification's main information.

Finally, to display the notification you use the `notify()` method:

```
nm.notify(notificationID, notifBuilder.build());
```

SUMMARY

This chapter first provided a detailed look at how activities and fragments work and the various forms in which you can display them. You also learned how to display dialog windows using activities.

The second part of this chapter demonstrated a very important concept in Android—the intent. The intent is the "glue" that enables different activities to be connected, and it is a vital concept to understand when developing for the Android platform.

EXERCISES

1. To create an activity, you create a Java class that extends what base class?
2. What attribute of the Application element is used to specify the theme?
3. What method do you override when displaying a dialog?
4. What is used to navigate between activities?
5. What method should you use if you plan on receiving information back from an activity?

You can find answers to the exercises in the appendix.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Creating an activity	All activities must be declared in the `AndroidManifest.xml` file.
Key life cycle of an activity	When an activity is started, the `onStart()` and `onResume()` events are always called. When an activity is killed or sent to the background, the `onPause()` event is always called.
Displaying an activity as a dialog	Use the `showDialog()` method and implement the `onCreateDialog()` method.
Fragments	Fragments are "mini-activities" that you can add or remove from activities.
Manipulating fragments programmatically	You need to use the `FragmentManager` and `FragmentTransaction` classes when adding, removing, or replacing fragments during runtime.
Life cycle of a fragment	Similar to that of an activity—you save the state of a fragment in the `onPause()` event, and restore its state in one of the following events: `onCreate()`, `onCreateView()`, or `onActivityCreated()`.
Intent	The "glue" that connects different activities.
Calling an activity	Use the `startActivity()` or `startActivityForResult()` method.
Passing data to an activity	Use the `Bundle` object.
Components in an `Intent` object	An `Intent` object can contain the following: action, data, type, and category.
Displaying notifications	Use the `NotificationManager` class.