

Section 1

Get started fast with Android

This section gets you started fast with Android programming. Chapter 1 introduces you to some important terms, concepts, and background information. Chapter 2 shows you how to use Eclipse to work with Android projects. Chapter 3 introduces you to the basic skills that you need for developing Android apps. And chapter 4 shows you how to test and debug Android apps.

To illustrate these skills, this section shows how to write, test, and debug a simple Tip Calculator app. This app calculates the tip you would give based on the amount of a bill. When you complete these chapters, you'll be able to write, test, and debug simple applications of your own.

Chapter 1

An introduction to Android

This chapter introduces you to the concepts and terms that you need for developing Android apps. To give you a clear idea of how the code for an Android app works, this chapter presents some of the code for a simple app that calculates tips. When you finish this chapter, you'll be ready to get started with Android development.

Android overview	3
Types of devices	3
Types of apps	4
A brief history	5
Versions	6
System architecture	7
How apps are compiled and run	8
A simple Android app	9
The user interface	9
The XML for the user interface	10
The XML for the display text	12
The Java source code	13
The Android manifest	15
Perspective	16

Android overview

Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers. Before you begin developing apps that run on Android, it's helpful to take a moment to consider the types of Android devices and apps that are available today. In addition, it's helpful to understand the history, versions, and architecture of Android.

Types of devices

Figure 1-1 starts by showing two of the most popular Android devices, a smartphone and a tablet. However, the code for Android is open-source. As a result, it can be customized to work with other types of electronic devices such as eBook readers, cameras, home automation systems, home appliances, vehicle systems, game consoles, and so on.

Types of apps

Android has a large community of developers writing *applications*, more commonly referred to as *apps*, that extend the functionality of Android devices. Figure 1-2 lists some categories of different types of apps and describes some of the functionality available from each category. As you review this list, keep in mind that these categories are only intended to give you a general idea of the various types of Android apps. More categories exist, and the functionality that's available from Android apps is constantly expanding.

If you have used a smartphone or tablet, you should be familiar with some of the apps listed in this figure. For example, you probably use apps to send and receive text messages and email. You probably use apps to take pictures, listen to music, and watch video. You probably use apps to get directions and navigate to a location. You probably use apps to check the weather, read news, and browse the web.

In addition, you may use apps from social media companies like Facebook and Twitter to work with social media. You may use apps to play games like Angry Birds. The main point here is that there are many different types of apps and that application developers are constantly creating new apps that use the capabilities of Android phones in new ways.

Some apps come preinstalled on the device. For example, most phones include apps for managing contacts, using the phone, sending and receiving text messages and email, working with photos and video, and web browsing. Other apps are available for download through Google Play or third-party sites. Some apps are free. Other apps are expensive. All apps were created by somebody like you who learned how to develop Android apps.

A brief history

Figure 1-3 summarizes the history of Android. In 2003, a handful of entrepreneurs in Palo Alto, California, founded Android, Inc. In 2005, Google bought Android, Inc. Then, in 2007, the *Open Handset Alliance* was announced. This alliance consists of a consortium of hardware, software, and telecommunication companies devoted to advancing open standards for mobile devices. That same year, Google released Android code as open source under the Apache License. In addition, Google helped to create the *Android Open Source Project (AOSP)* and put it in charge of the maintenance and further development of Android.

In 2008, the first version of the *Android Software Development Kit (SDK)* was released. This SDK contains all of the tools that Android developers need to develop apps. Later in 2008, the first Android phones became available.

Since 2008, new versions of the Android SDK have continued to be released, and Android devices have continued to proliferate. During that time, hundreds of thousands of apps have been developed, and billions of apps have been downloaded.

Versions

Figure 1-4 describes all major releases of Android starting with version 1.6 and ending with version 4.3. Here, each version of Android corresponds with an API (Application Programming Interface) number. For example, version 4.2 corresponds with API 17, and version 2.2 corresponds with API 8. In addition, each version has a code name that's based on something sweet. For example, version 2.2 carries the code name Froyo, while version 4.1 is known as Jelly Bean.

As you develop an Android app, you must decide the minimum API level that your app supports. As of April 2013, many developers choose 7 or 8 as the minimum API level to support since that covers a high percentage of all Android devices.

The distribution percentages shown here are from April 2013. As time progresses, more users will upgrade from older devices to newer ones. As a result, you should check the URL shown in this figure to get current percentages before you decide the minimum API level for your app.

As you review this figure, you may notice that it doesn't include some versions of Android such as 1.0 and 1.1. That's because there are virtually no devices left that run on these versions of Android. As time marches on, it's inevitable that the same fate will befall other older versions of Android too.

System architecture

Figure 1-5 shows the Android system architecture, which is also known as the *Android stack*. This stack has four layers.

The bottom layer of the stack is Linux, an open-source operating system that's portable and secure. This operating system provides low-level drivers for hardware, networking, file system access, and inter-process communication (IPC).

The second layer up in the stack contains the native libraries. These libraries are written in C or C++. They include the *Dalvik virtual machine (VM)*, which works similarly to the *Java virtual machine (JVM)*. However, the Dalvik VM was designed specifically for mobile devices and their inherent limitations, such as battery life and processing power.

The third layer up in the stack contains the application framework. This layer is written mostly in Java, and it provides libraries that can be used by the top layer of the stack. In this book, you'll learn how to use some of these libraries, such as the libraries for the notification manager, content providers, and the location manager.

The top layer of the stack contains Android apps. These apps include pre-installed apps such as the apps that you can use to manage the home screen, manage your contacts, make and receive calls, browse the web, and so on. In addition, these apps include other apps that you can download and install. These types of apps are written in Java, and they are the type of apps that you'll learn to develop in this book.

How apps are compiled and run

When you develop an Android app, you typically use an *IDE* (*Integrated Development Environment*) such as Eclipse to create a *project*. A project contains all of the files for the app including the files for the Java source code. In the next chapter, you'll learn how to use Eclipse to work with Android projects.

When you're ready to test a project, you can run it. Figure 1-6 shows how this works. When you run a project, the IDE typically compiles and packages the project automatically before running it. This is known as *building* the project.

When the IDE builds a project, it compiles the *Java source code* (.java files) into *Java bytecodes* (.class files). Then, it compiles the Java bytecodes into *Dalvik executable files* (.dex files) that can be run by the Dalvik virtual machine that's available from all Android devices.

When the IDE builds a project, it puts the .dex files and the rest of the files for the project into an *Android package* (.apk file). This file contains all of the files necessary to run your app including the .dex files and other compiled resources, uncompiled resources, and a binary version of the Android manifest. The *Android manifest* is a file that specifies some essential information about an app that the Android system must have before it can run the app. In its non-binary version, the Android manifest is stored in a file named `AndroidManifest.xml`.

For security reasons, all Android apps must be digitally *signed* with a certificate. During development, the IDE typically signs the app for you automatically using a special debug key. Then, it runs the app on the specified physical device such as a smartphone or tablet. Or, it runs the app on the specified *emulator*, which is a piece of software that runs on your computer and mimics an Android device. An Android emulator can also be called an *Android Virtual Device (AVD)*.

The *Android debug bridge (ADB)* lets your IDE communicate with an emulator or a physical Android device. This is necessary to provide the debugging capabilities described in chapter 4.

When you are ready to release the app, you must sign the app in release mode, using your own private key. For more information about this, please see chapter 17.

A simple Android app

To give you a better idea of how an Android app works, this topic presents a basic Tip Calculator app. For a simple app like this one, the programmer only needs to modify three files: the XML file for the user interface, the XML file for the display text, and the Java source code for the activity. In addition, the programmer may sometimes need to view or modify the Android manifest file for the app.

The user interface

Figure 1-7 shows the user interface for the Tip Calculator app after it has been displayed in an emulator for Android 4.2 (API 17). Of course, this app can also be run on emulators for other Android platforms. Similarly, this app can also be run on a physical device that's configured for debugging.

The emulator shown in the figure includes a message that indicates that the hardware keyboard is not enabled in AVD and that you should use the on-screen keyboard. As a result, you can use the on-screen keyboard, known as a *soft keyboard*, to enter text as shown in this figure. To do that, you click on the Bill Amount text box. When you do, the emulator should display a soft keyboard that's optimized for entering a number with a decimal point. When you're done using that keyboard to enter an amount for the bill, you can click the Done key. When you do, the app calculates the tip and total for the bill.

If the Hardware Keyboard option is enabled for an emulator, the emulator should display a message that indicates that you can use your computer's keyboard to enter text. In that case, the emulator won't display a soft keyboard when you click on the Bill Amount text box. Instead, you use your computer's keyboard to enter text. Although this provides an easier way to enter text, it doesn't accurately emulate touchscreen devices.

By default, this Tip Calculator app uses a tip percent of 15%. However, if you want to increase or decrease the tip amount, you can click the Increase (+) or Decrease (-) buttons. When you do, the app recalculates the tip and total amounts for the bill.

The XML for the user interface

Figure 1-8 shows the XML for the user interface. This code defines an *activity*, which is a screen that users can interact with, and it's stored in a file named `activity_tip_calculator.xml`.

If you have experience with XML or HTML, you may be able to understand much of this code already. If not, don't worry! You'll learn the details about how this code works in chapter 3. For now, here's a brief explanation.

An XML file consists of one or more *elements*. These elements can be nested within each other. In this file, the `RelativeLayout` element is the *parent element*, and it has ten *child elements*: seven `TextView` elements, one `EditText` element, and two `Button` elements.

Each of these elements has multiple *attributes*. For example, the `RelativeLayout` element has a padding attribute that determines how much space there is on the screen between the `RelativeLayout` element and its child elements. Many of these attributes are set automatically for you when you use a graphical tool, like Eclipse to create the layout. As a result, you often don't need to set these attributes. However, you sometimes need to modify them.

The `RelativeLayout` element defines a *layout*, which is a container that determines where its child elements are displayed on the screen. In Android, there are several types of layouts that use different techniques to control where the child elements are displayed. The *relative layout* organizes its child elements in relation to one another.

The `RelativeLayout` element has several attributes that determine how it works. Here, the first two attributes are set automatically, and you can usually ignore them. Then, the `layout_width` and `layout_height` attributes determine that the layout should use the whole screen. Finally, the padding attribute determines that there should be some *padding*, or space, between the edge of the layout and its child elements.

To specify the amount of space, the padding attribute specifies a measurement of "10dp". Here, *dp* stands for *density-independent pixel*. In general, it's considered a best practice to use density-independent pixels for padding and margins whenever possible as it allows your app to scale properly on various devices.

The child elements define the *controls* for the bill amount. In Android, controls are typically referred to as *widgets*.

The first `TextView` element displays "Bill Amount" on the user interface. In Android terms, this element is often referred to as a *text view*. Since this widget displays text that labels another widget, it can also be referred to as a *label*.

The first attribute of this widget defines an ID for the widget. Here, the value for the `id` attribute begins with "@+id/". This indicates that the following text is the ID for the widget. In this case, the widget has an ID of `billAmountLabel`. The next three attributes for this widget set its height, width, and padding.

The fifth attribute sets the text that's displayed on the widget to the value that's stored in the `bill_amount_label` string of the `strings.xml` file, which you'll learn about in a moment. To get a string from the `strings.xml` file, the value for this attribute begins with `"@string/"`. This indicates that the following text is the name of the string element that contains the text.

To specify the size of the text, the `textSize` attribute specifies a measurement of `"20sp"`. Here, *sp* stands for *scale-independent pixel*. In general, it's considered a best practice to use scale-independent pixels for text sizes whenever possible as it allows your text sizes to scale properly. Finally, the `textStyle` attribute sets the style of this text to bold.

The `EditText` element that follows displays the widget that allows the user to enter a bill amount. The Android documentation refers to this type of widget as an *editable text view*, but you can also refer to it as a *text box*. Most of the attributes of this `EditText` element work similarly to those of the previous `TextView` element. However, there are some differences. For example, the alignment attributes align the text box relative to its label. Then, the `ems` attribute specifies that the text box width should be 8 ems (8 of the letter *m*). Next, the `inputType` element specifies that the soft keyboard should only allow the user to enter decimal numbers.

To make it easy to test this app, the `text` attribute for the `EditText` element specifies a default value of `"34.60"`. During development, you can leave this attribute set to a default value like this one. However, once the application is working, you can set this attribute to an empty string.

The next two `TextView` elements define the widgets that display the tip percent. By now, you should have a general idea of what most of the attributes for these elements do. The main point to notice is that the layout attributes align these controls relative to one another. For example, the label for the tip percent is aligned with the left side of the label for the bill amount. Similarly, the percent label is displayed below the bill amount label.

The `Button` elements define the buttons that allow the user to increase or decrease the tip percent. These widgets are 45dp wide and high and they're aligned to the right of the `TextView` widgets for the tip percent.

The last four `TextView` elements define the widgets that display the tip and total amounts. These elements work like the previous `TextView` elements shown in this figure.

The XML for the display text

Part 3 of figure 1-8 also shows the XML for the display text. This XML is stored in a file named `strings.xml`, and it defines the names and values for the display text. For example, the string element with a name of `app_name` has a value of “Tip Calculator”. The string element with a name of `bill_amount_label` has a value of “Bill Amount”. The string element with a name of `bill_amount` has a value of “34.60”. And so on. To differentiate between the widgets that display labels and the widgets that display results, I have appended “_label” to the names of the labels.

At this point, you may be wondering why Android provides a mechanism for separating the display text from the user interface. The reason is because Android apps are likely to run on devices used around the world. As a result, Android attempts to make it easy to develop apps that are appropriate for different locales. This is known as *localization*. For example, you can provide one `strings.xml` file for English and another for French. Then, if Android determines that the locale for the device is French, it will use the `strings.xml` file that contains the French text. You’ll learn more about how that works in chapter 3.

The Java source code

Figure 1-9 shows the Java source code that handles events that occur on the user interface, gets input from the user interface, performs calculations, and updates the user interface. This source code is stored in a file named `TipCalculatorActivity.java`.

If you have experience with Java, you may be able to understand much of this code already. If not, don't worry! You'll learn the details about how this code works in chapter 3. For now, here's a brief explanation.

This source code declares a class named `TipCalculatorActivity`. To start, the package statement stores this class in the `com.murach.tipcalculator` package. Then, it imports all the Java and Android classes and interfaces that are needed for this class. As you can see, most of these classes and interfaces are stored in the `android.view` or `android.widget` packages.

The `TipCalculatorActivity` class inherits the `Activity` class that's provided by Android. This class defines the methods that are called during the lifecycle of an activity. This allows the `TipCalculatorActivity` class to override these lifecycle methods to perform tasks at different points in the activity's life.

Of these methods, the most important one is the `onCreate` method, which is called when Android creates an activity. This method is the only lifecycle method that's used by the activity shown in this chapter. However, in chapter 3, you'll learn how to use two other activity lifecycle methods to save and restore data if the activity is interrupted by another activity such as a phone call.

The `TipCalculatorActivity` class also implements two interfaces. These interfaces allow this class to handle the events that occur on the user interface. First, the `OnEditorActionListener` interface makes the class a listener for the `EditorAction` event. As a result, this class must implement the `onEditorAction` method. Then, the `OnClickListener` interface makes the class a listener for the `Click` event. As a result, this class must implement the `onClick` method.

Within the class, the first six statements define the instance variables for the widgets that the class needs to work with. Then, the seventh statement defines an instance variable for the tip percent and sets its default value to `.15`. Since this statement uses the float type, this code appends an *f* to the number to indicate that the literal is of the float type, not the double type.

The `onCreate` method begins by passing its `Bundle` parameter to the superclass (the `Activity` class). Then, it uses the `setContentView` method that's available from the `Activity` superclass to display the user interface that's defined in the `activity_tip_calculator.xml` file. To do that, this method uses the `R` class to access the compiled resources that are available to the Android app. This class is created and compiled automatically when you build a project. In other words, Android converts the XML in the `activity_tip_calculator.xml` file into Java code, compiles it, and makes it available through the `R` class.

After displaying the user interface, the `onCreate` method gets references to the six widgets that it declared earlier. To do that, it calls the `findViewById` method for each widget and uses the `R` class to pass the ID of the widget to that method. Then, it casts the widget object that's returned to the appropriate type. These types correspond with the names of the elements in the XML file for the user interface.

After getting references to the widgets, this code sets the listeners. First, it sets the current class (`TipCalculatorActivity`) as the listener for the `EditorAction` event on the bill amount text box. As a result, the `onEditorAction` method in this class is executed when the user uses the bill amount text box to enter an amount for the bill. Then, this code sets the current class as the listener for the `Click` event on both of the buttons. As a result, the `onClick` method is executed when the user clicks on either button.

After setting the listeners, this code calls the `calculateAndDisplay` method. This method calculates the tip and total amounts and displays all current data on the user interface. To start, it gets the string that was entered into the bill amount text box. Then, it checks whether the string is empty. If so, it sets the bill amount to zero. Otherwise, it converts the string to a float value. Next, it calculates the tip and total amounts. Finally, it formats and displays the results of the calculations. To do that, it uses the `NumberFormat` class to convert the float values into string values that have currency and percent formatting.

The `onEditorAction` method is executed whenever the user presses an action key on a soft keyboard such as the Done key. This method calls the `calculateAndDisplay` method to calculate the new tip amount and total and display it on the user interface.

The `onClick` method is executed whenever the user clicks on either of the buttons. This method begins by using a switch statement to check which button is clicked. Then, if the Decrease (-) button is clicked, this code decreases the tip percent by 1 percent and calls the `calculateAndDisplay` method to refresh the data that's on the user interface. Conversely, if the Increase (+) button is clicked, this code increases the tip percent by 1 percent and calls the `calculateAndDisplay` method to refresh the data.

The Android manifest

Figure 1-10 shows the `AndroidManifest.xml` file, which is known as the *Android manifest*. This file specifies some essential information about an app that the Android system must have before it can run the app. Within the manifest, there is always one application element. Within the application element, there may be one or more activity elements. However, for this app, there is only a single activity. This activity is identified as the main activity, and it is launched when the app starts.

Like the XML file for the user interface, the manifest file can access strings in the `strings.xml` file. For example, the `label` attribute of the application element specifies the `app_name` string in the `strings.xml` file. Similarly, the `label` attribute of the activity element specifies the `title_activity_tip_calculator` string in the `strings.xml` file.

This manifest file sets the `screenOrientation` attribute of the activity to “portrait”. As a result, this application can only be displayed in portrait orientation, not landscape orientation. That’s important because this activity doesn’t include the code that’s necessary to gracefully handle the switch between the portrait and landscape orientations. In chapter 3, you’ll learn how to handle changes in orientation.

Perspective

The goal of this chapter has been to introduce you to some concepts and terms that you should understand before getting started with Android programming. However, this chapter is just an overview of Android programming. In the next few chapters, you'll learn the hands-on skills you need to develop a simple Android app like one shown in this chapter.

Terms

Android application	Android Virtual Device (AVD)
app	Android debug bridge (ADB)
Open Handset Alliance	soft keyboard
Android Open Source Project (AOSP)	activity
Software Development Kit (SDK)	element
Android stack	parent element
Dalvik virtual machine (VM)	child element
Java virtual machine (JVM)	attribute
Integrated Development Environment (IDE)	layout
project	relative layout
building	padding
Java source code	density-independent pixel (dp)
Java bytecodes	control
Dalvik executable files	widget
Android package	text view
Android manifest	label
signed app	scale-independent pixel (sp)
emulator	editable text view
	text box
	localization

Summary

- *Android* is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers. It was first released in 2008. Android code is open-source.
- Android developers write *applications*, or *apps*.
- Android system architecture, known as the *Android stack*, consists of four layers: Linux, native libraries, the application framework, and Android apps.
- An Android app is typically developed using an *IDE (Integrated Development Environment)* like Eclipse, using Java as the programming language.
- Android apps can be run on a physical Android device or on an *emulator*, also called an *Android Virtual Device (AVD)*.
- An Android project must be built before it is run, compiling the *Java source code* (.java files) into *Java bytecodes* (.class files) and then into *Dalvik executable files* (.dex files).
- All of the files for an Android project are put into an *Android package* (.apk file), which includes a binary version of the AndroidManifest.xml file.
- To run an app on an emulator or device, it must be digitally *signed* with a certificate.
- The *Android debug bridge (ADB)* lets your IDE communicate with an emulator or a physical Android device.