CHAPTER ONE

Basics of Android, First App: HelloAndroid

CHAPTER CONTENTS

Introduction

Today, it seems that almost everybody has a smartphone and is using apps. There are apps to check our email, check the weather, play games, calculate a mortgage payment, translate one language into another, learn algebra, or even apps for websites, social media, or news organizations such as Facebook, Twitter, or CNN. In this chapter, we learn how to develop our first app for Android devices.

1.1 Smartphones and Their Operating Systems

1.1.1 Smartphones

A smartphone is an intelligent cellular phone, or a cellular phone with a computer inside, as well as some extra hardware. Thus, programmers can write application software for them; such application software is called an app. A smartphone has the typical components of a regular computer: a CPU, memory, a storage device, an operating system, and other devices, such as a camera, an accelerometer, or a GPS.

The two most well-known operating systems are the Android operating system, from Google, and iOS, from Apple. Other popular operating systems for smartphones include BlackBerry, Windows, and Symbian. The annual number of smartphones sold worldwide is now over one billion. Furthermore, smartphones comprise an increasing percentage of the mobile phones sold.

1.1.2 Android Phones

There are over 100 different varieties of Android phones or tablets. They can have a different CPU, a different screen resolution, and a different amount of memory available. This makes it difficult for developers to test their apps on actual devices. The dimensions of the various components of the user interface of an app can be different depending on the Android phone or tablet. Furthermore, in complex games where speed is an issue, an app can run differently on an old and slow Android device as compared to a more recent one. This is something to keep in mind as we develop apps for the Android market.

1.1.3 Apps and Google Play

Android apps are distributed via Google Play (https://play.google.com), which used to be called the Android Market. Google Play is actually a store for digital content, much like the Apple's app store, not just apps. In order to distribute apps on Google Play, you need to become a registered developer, which costs $25.

There are now over one million apps in Google Play, and the vast majority of those apps are free. Top categories include entertainment (games), personalization, tools, books, and references. Most of the downloaded apps are free apps. One should also be aware that the Android operating

system is open, so anyone can easily copy an app from one Android device to another. There is little protection for intellectual property.

1.2 Development Environment for Android Apps

The typical and recommended development environment for Android apps is comprised of the following:

- Java Development Kit (JDK)
- Android Studio
- Android Standard Development Kit (Android SDK)

It is not a requirement to use Android Studio to develop Android apps, we can run our code from the command line or use another integrated development environment, such as Eclipse. However, Android Studio is Google's official development environment and is likely to quickly become the industry standard; we use Android Studio in this book.

In order to set up a complete Android app development environment, we need to:

- Download and install the latest Java SDK (if we have not done it yet), which we can find at http://www.oracle.com/technetwork/java/javase/downloads/index.html.
- Download and install Android Studio, including the IDE, SDK tools, and the emulator system, which we can find at http://developer.android.com/sdk/index.html

1.3 Our First App: HelloAndroid

Let's create our first Android app.

1.3.1. Skeleton App

Open Android Studio. The first time we run Android Studio, the environment checks that it is up to date, asks us to download components if needed. Once that is done and we click on Finish (see **FIGURE 1.1**), we see the screen shown in **FIGURE 1.2**. Recent projects are listed on the left side; there will not be any the first time we use Android Studio. To start a new project, click on Start a new Android Studio project.

In the dialog box shown in **FIGURE 1.3**, we type in our project name (we choose HelloAndroid) and our domain (we chose jblearning.com; if we do not have a domain name, we can choose any name); the other two fields (package name and project location) are automatically filled. If we prefer a different project location, we can edit that field. Note that the package name is our domain name in reverse. It is typical for developers to name their package with their domain name in reverse; it insures that it is unique. When we are done, we click Next.

The next dialog box (**FIGURE 1.4**) is used to specify the minimum SDK for this project; depending on our app, this can be important. For example, if we incorporate advertising, this may require a higher SDK level than the default level; however, the more recent the SDK level we

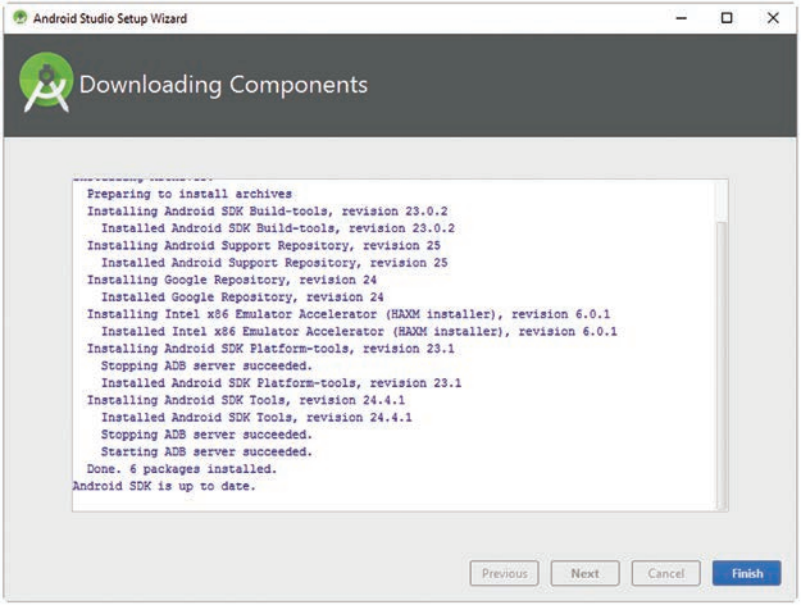

FIGURE 1.1 **Downloading components**

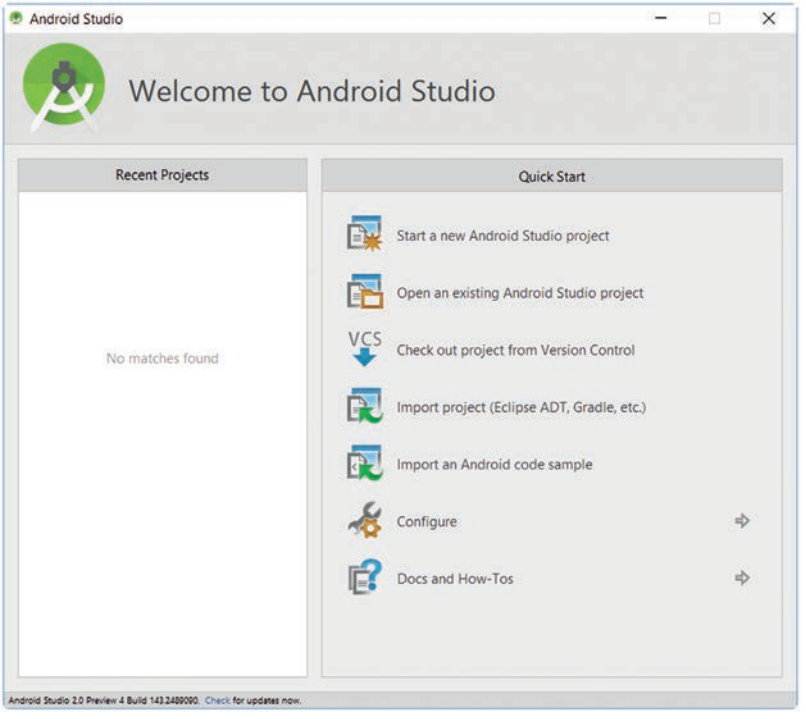

FIGURE 1.2 **Welcome screen**

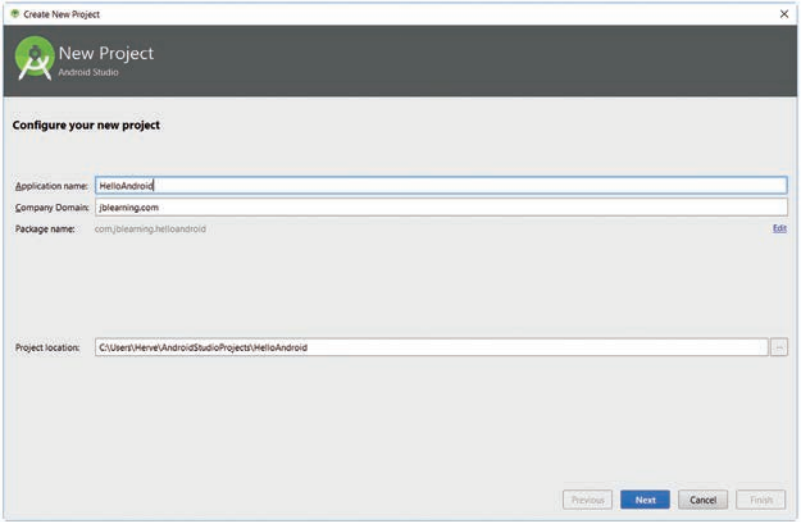

FIGURE 1.3 **Dialog box for new project**

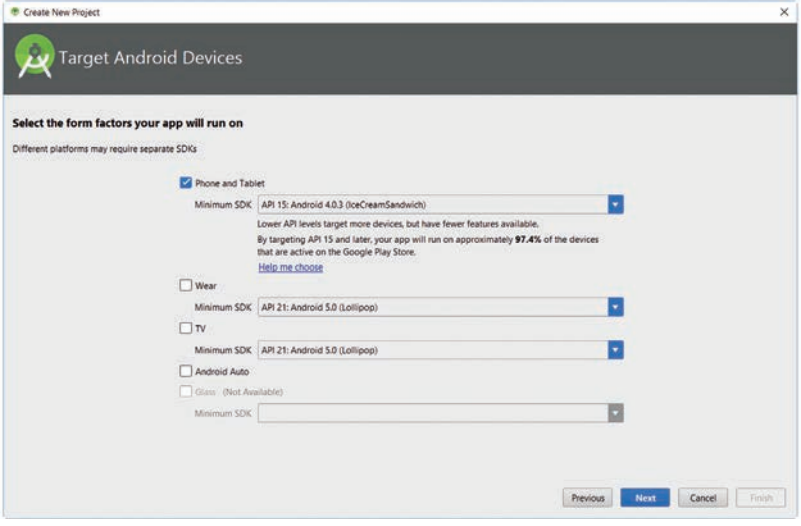

FIGURE 1.4 **Dialog box to specify the minimum SDK**

specify, the more we restrict the potential number of users for our app. For this app, we keep the default SDK level and click Next.

In the next dialog box (**FIGURE 1.5**), we can choose among several templates; a template creates skeleton code with some predefined user-interface functionality; often, it provides a user interface that is similar to what can be found in native apps. For this app, we choose the Empty Activity template—it creates a minimum skeleton code.

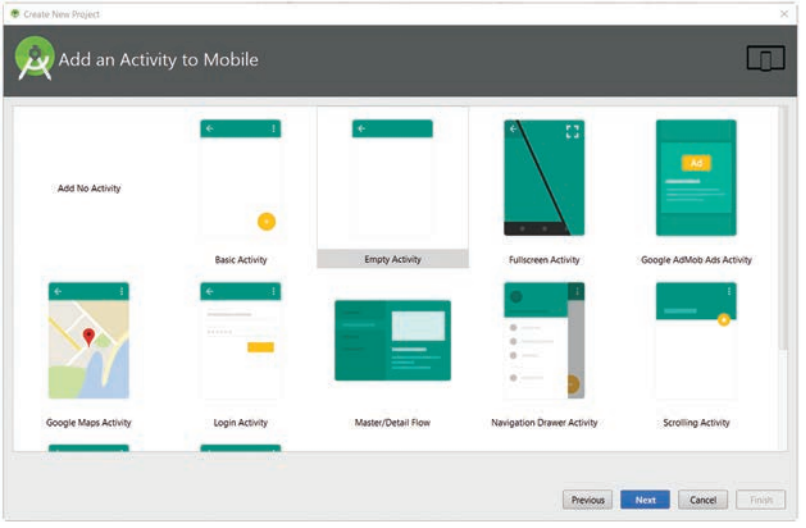

FIGURE 1.5 **Dialog box to choose the type of activity**

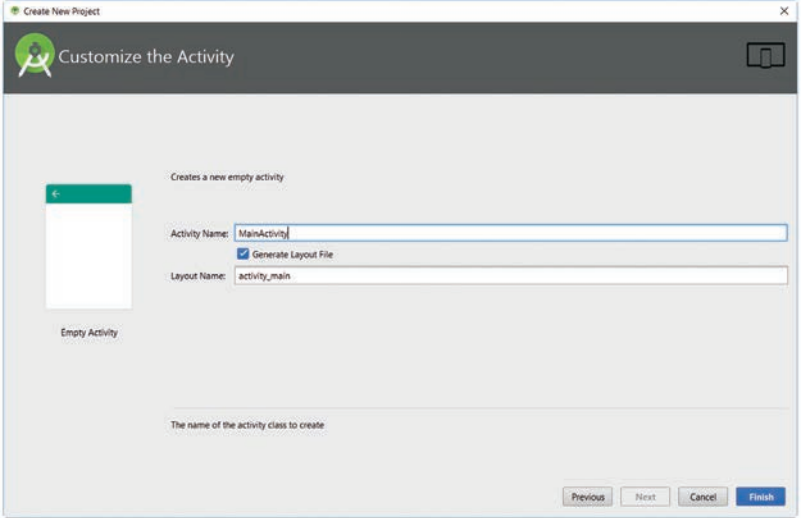

FIGURE 1.6 **File names dialog box**

In the next dialog box (**FIGURE 1.6**), we can specify various names for the first activity class and layout files. For this first app, we keep the default names, `MainActivity` for the class, and activity_main for the layout file. After we click on Finish, our project is created.

The project directory structure has been created along with many source files, which we can see on the left pane of the Android Studio development environment (shown in **FIGURE 1.7**).

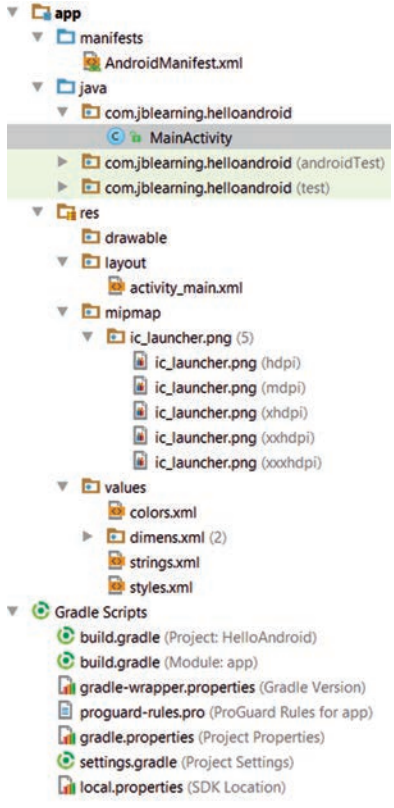

FIGURE 1.7 The app directory structure in the IDE

The program generates a lot of directories and files automatically.

- The manifests directory contains the AndroidManifest.xml file, automatically generated and editable. Among other things, it specifies the resources that this app uses, such as activities, the file system, the Internet, the device's hardware resources. . . When a user downloads an app, this file tells the user about the resources that this app uses (e.g., if it writes to the file system of the device).
- The java directory contains the Java source files. We can add more Java source files as our app gets more complex.
- The res (`res` stands for "resources") directory contains resources such as utility files (to define strings, menus, layouts, colors, styles) and images, sounds, etc. Ids are automatically generated for these resources inside a file named R.java. R.java should not be modified.
- Inside the res directory, the drawable directory is meant to contain images and other drawable resources. It can contain jpegs, pngs, gifs, files to define gradients, etc. As needed, we can add resources to this directory.

- Inside the res directory, the mipmap directory contains the icon for the app. As needed, we can add resources to this directory.
- Inside the res directory, the layout directory contains XML files defining screen layouts. At this point, it contains the activity_main.xml file, an automatically generated layout file for our empty activity. We can edit this file to define the Graphical User Interface (GUI) for this app.
- Inside the res directory, the values directory contains XML files defining various resources such as colors (in the colors.xml file), dimensions (in the dimens.xml file), styles (in the styles.xml file), or strings (in the strings.xml file). We can edit these files to define more color, dimension, style, or string resources.
- The gradle scripts directory contains the scripts used to build the app.

In this chapter, we look at the following files in detail: AndroidManifest.xml, MainActivity.java, dimens.xml, strings.xml, styles.xml, colors.xml, and activity_main.xml. We also add an icon for the app.

1.3.2 GUI Preview

A nice feature of Android Studio is that it lets us preview a screen without running the app. A layout file needs to be selected for the preview to show. To select activity_main.xml, we double click on the file name. In the menu, if we select View, Tool Windows, Preview, a preview of the app shows on the right side of the environment, as shown in **FIGURE 1.8**. In this way, we can edit the GUI and preview it without running the app. We can preview a screen in both portrait and landscape orientations by choosing from a menu (**FIGURE 1.9**). **FIGURE 1.10** shows the preview in landscape orientation. We can also preview the app choosing a different theme by clicking on the AppTheme drop-down list (**FIGURE 1.11**) and choosing among many devices (**FIGURE 1.12**) by clicking on the device drop-down list.

1.3.3 XML Files: activity_main.xml, colors.xml, styles.xml, strings.xml, dimens.xml

The GUI for this app is defined in the activity_main.xml file. Note that we can also set the GUI by code.

To open activity_main.xml, we click on res, then click on layout, then double click on activity_main.xml. In order to see the actual XML code, we may need to click on the Text tab at the bottom of the panel.

Activity_main.xml, shown in **EXAMPLE 1.1**, is an **eXtensible Markup Language (XML)** file. The Android development environment uses XML extensively, in particular to define the user interface of the various screens of an app and also to define various resources such as strings, dimensions, menus, styles, shapes, or gradients.

We may already be familiar with **HyperText Markup Language (HTML)** and its syntax. HTML is the language that web browsers use, and has a fixed set of tags and attributes. XML uses essentially the same syntax as HTML, but we can create our own tags and attributes.

FIGURE 1.8 **A preview of the app in the environment**

FIGURE 1.9 **Choosing the orientation for the preview**

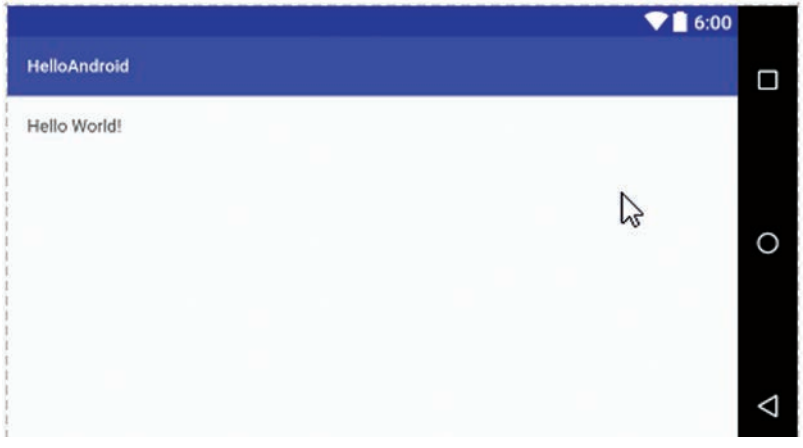

FIGURE 1.10 **A preview of the app in landscape orientation**

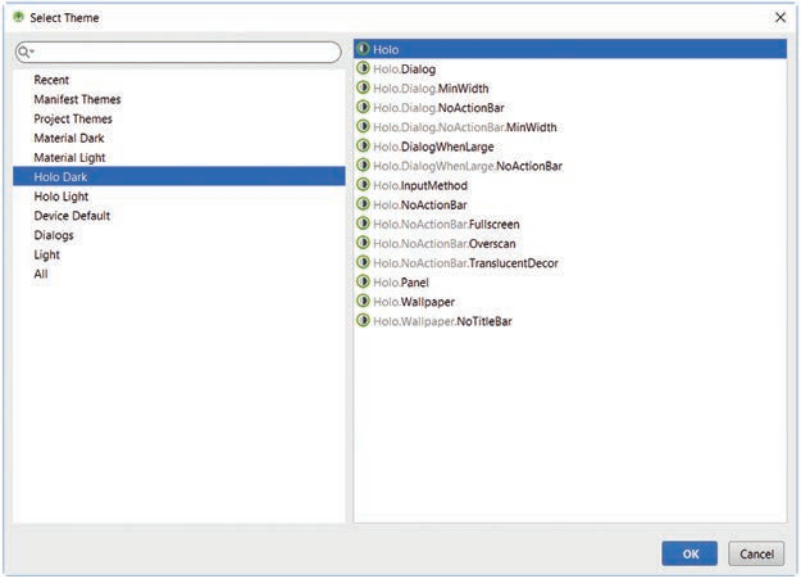

FIGURE 1.11 Choosing a theme for the preview (after selecting Holo Dark)

We can find the full XML documentation at http://www.w3.org/XML/.

We will next outline the main characteristics of XML. An XML document is made up of elements, and each element can have 0 or more attribute/value pairs. Elements can be nested. An element can have content or not. A non-empty element begins with a start tag and ends with an end tag. The text between the start tag and the end tag is called the element content.

Although the actual specification is a bit more complex, we will use the following simplified syntax to define a non-empty XML element:

```
<tagName attribute1="value1" attribute2="value2" ... >
Element Content
</tagName>
```

Here are two examples:

```
<price>46.00</price>
<app language="Java" version="7.0">Hello Android</app>
```

If an element has no content, then we can use an empty element tag with the following syntax:

```
<tagName attribute1="value1" attribute2="value2" ... />
```

Here are two examples:

```
<website name="twitter"/>
<app language="Java" version="7.0"/>
```

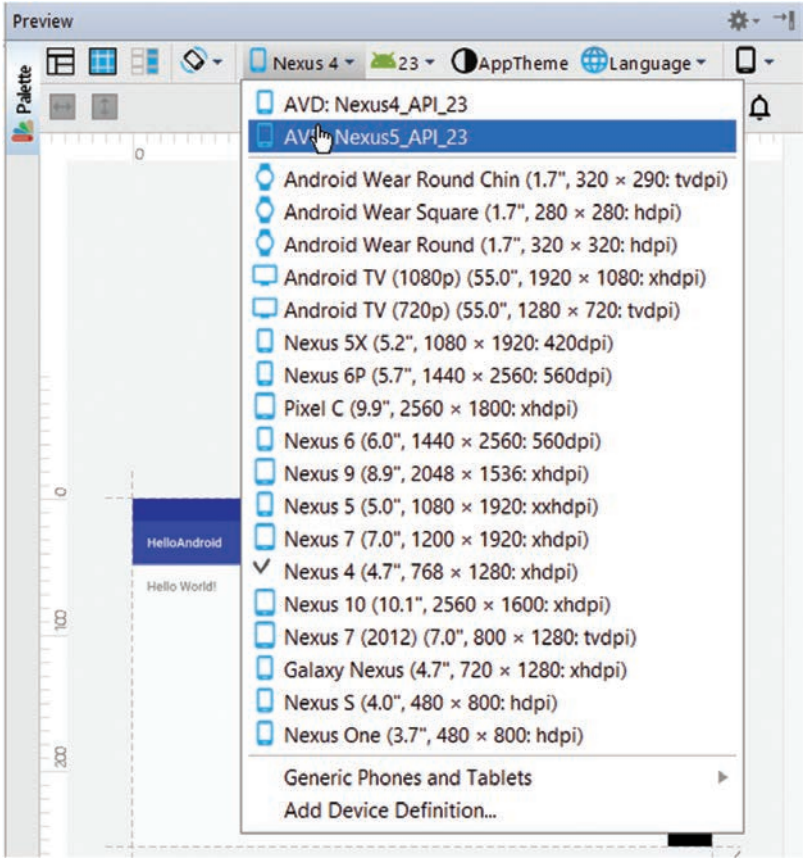

FIGURE 1.12 Choosing a device for the preview

There are rules for naming tags; we use only tag names that start with a letter or an underscore and are followed by 0 or more letters, underscores, or digits. The official specification for tag names is more complex than the earlier, and we can find it at http://www.w3.org/XML/Core/#Publications.

Comments use this syntax:

```
<!-- Write a comment here -->
```

The file activity_main.xml defines how the GUI should look like. The `RelativeLayout` element defines that the various graphical elements will be positioned and displayed in relation to each other or their parent graphical container.

In the activity_main.xml file the element `RelativeLayout` uses the first syntax and includes attributes `android:layout_width` and `android:layout_height`, which both have value `match_parent` (lines 5–6). This means that the `RelativeLayout` element will be as big as its parent element, which in this example is the screen. It also includes attributes `android:paddingBottom`,

android:paddingLeft, android:paddingRight, and android:paddingTop (lines 7–10). The android:paddingBottom element has value @dimen/activity_vertical_margin (line 7): this means that there will be some padding on the left side of the screen equal to the value of the dimen element named activity_vertical_margin (in this case 16 px) in the dimens.xml file located in the values directory of the res directory.

Generally, if a constant named constant_name is defined in a resource file named resource_types.xml, we can access the value of that constant using the syntax @resource_type/constant_name.

Lines 13–16 define a TextView element that is nested inside the RelativeLayout element; a TextView element is an instance of the TextView class, which encapsulates a label. This TextView element is an empty element and as such has no content; it has three attribute/value pairs:

- The attributes android:layout_width and android:layout_height have value wrap_content (lines 14 and 15). That means that the TextView element's width and height are just as big as necessary to make the contents of the TextView element fit; the element "wraps" around its contents.
- The attribute android:text defines the String displayed inside the TextView; it has value Hello World!.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <RelativeLayout
3      xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:tools="http://schemas.android.com/tools"
5      android:layout_width="match_parent"
6      android:layout_height="match_parent"
7      android:paddingBottom="@dimen/activity_vertical_margin"
8      android:paddingLeft="@dimen/activity_horizontal_margin"
9      android:paddingRight="@dimen/activity_horizontal_margin"
10     android:paddingTop="@dimen/activity_vertical_margin"
11     tools:context="com.jblearning.helloandroid.MainActivity">
12
13   <TextView
14       android:layout_width="wrap_content"
15       android:layout_height="wrap_content"
16       android:text="Hello World!"/>
17 </RelativeLayout>
```

EXAMPLE 1.1 The activity_main.xml file

The activity_horizonal_margin and the activity_vertical_margin values are defined within two dimen elements in the dimens.xml file (lines 3–4), shown in **EXAMPLE 1.2**. Their values are 16 px. The suffix dp stands for density pixels, which means that the value is device independent. If our app requires using a different value per screen size, we can define many values in the dimens.xml file, one per screen size.

Inside the dimens.xml file, the syntax for defining a `dimen` resource is:

```
<dimen name="dimenName">valueOfDimension</dimen>
```

```
1  <resources>
2    <!-- Default screen margins, per the Android Design guidelines. -->
3    <dimen name="activity_horizontal_margin">16dp</dimen>
4    <dimen name="activity_vertical_margin">16dp</dimen>
5  </resources>
```

EXAMPLE 1.2 The dimens.xml file

Try to modify dimens.xml as in **EXAMPLE 1.3**. The environment updates its preview as shown in **FIGURE 1.13**.

```
1  <resources>
2    <!-- Default screen margins, per the Android Design guidelines. -->
3    <dimen name="activity_horizontal_margin">50dp</dimen>
4    <dimen name="activity_vertical_margin">100dp</dimen>
5  </resources>
```

EXAMPLE 1.3 The modified dimens.xml file

Similarly, the `String` whose value is `HelloAndroid` (see the blue title of the app in Figure 1.13) is defined at line 2 of the strings.xml file (**EXAMPLE 1.4**), located in the values directory of the res directory. We can create `String` constants in our Java files, but it is recommended to store `String` constants in the strings.xml file as much as possible. If we want to modify one or several `String` values later, it is easier to edit that file than to edit our Java code.

Inside the strings.xml file, the syntax for defining a `String` resource is:

```
<string name="stringName">valueOfString</string>
```

```
1  <resources>
2      <string name="app_name">HelloAndroid</string>
3  </resources>
```

EXAMPLE 1.4 The strings.xml file

Try to modify strings.xml as in **EXAMPLE 1.5**. The environment updates its preview as shown in **FIGURE 1.14**. The `String` `app_name` is used in the AndroidManifest.xml file, which is explained in paragraph 1.7.

FIGURE 1.13 A preview of the app in the environment after updating the dimens.xml file

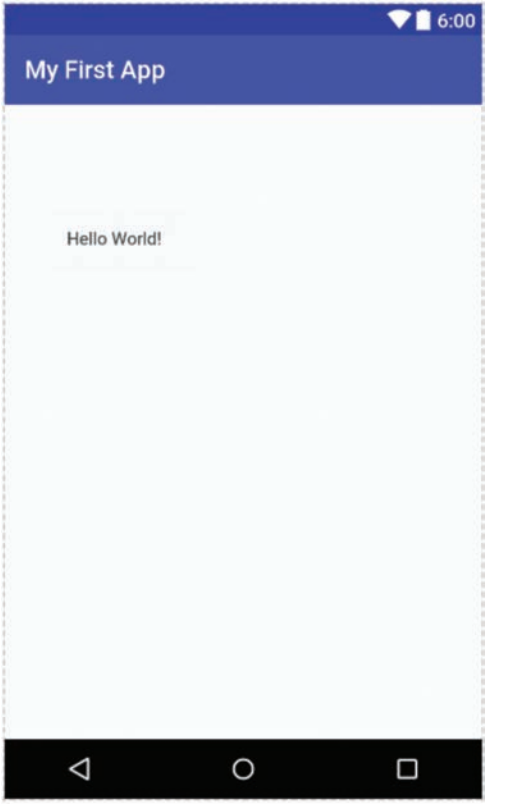

FIGURE 1.14 A preview of the app in the environment after updating the strings.xml file

```
1  <resources>
2      <string name="app_name">My First App</string>
3  </resources>
```

EXAMPLE 1.5 The modified strings.xml file

The file styles.xml, located in the values directory of the res directory, defines the various styles used in the app. EXAMPLE 1.6 shows its automatically generated contents.

Inside the styles.xml file, we can modify a style by adding an `item` element using this syntax:

```
<item name="styleAttribute">valueOfItem</item>
```

The name of the style attribute that specifies the text size inside a `TextView` is `android:textSize`. In EXAMPLE 11.7, we change the default text size to 40 at line 6. The environment updates its preview as shown in FIGURE 1.15.

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

EXAMPLE 1.6 The styles.xml file

```
<resources>

    <!-- Base application theme. -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Customize your theme here. -->
        <item name="android:textSize">40sp</item>
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>

</resources>
```

EXAMPLE 1.7 The modified styles.xml file

Note that the style defined in styles.xml consists of three color constants using the syntax `@color/color_name`. These constants are defined in the colors.xml file, shown in **EXAMPLE 1.8**, and also located in the values directory of the res directory. Its contents are automatically generated. Inside the colors.xml file, the syntax for defining a `color` resource is:

```
<color name="colorName">valueOfColor</color>
```

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

EXAMPLE 1.8 The colors.xml file

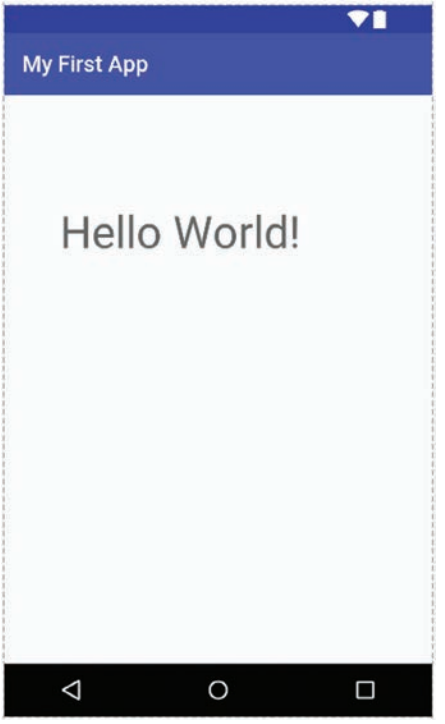

FIGURE 1.15 A preview of the app in the environment after updating the styles.xml file

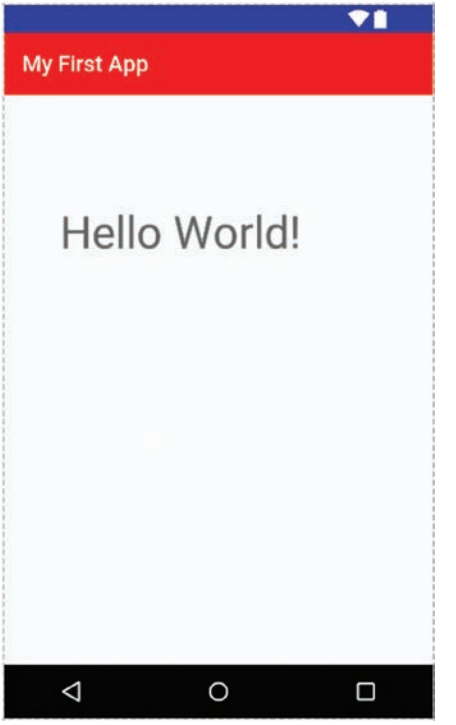

FIGURE 1.16 A preview of the app in the environment after updating the colors.xml file

The color values are defined using the syntax `#RRGGBB` where `R`, `G`, and `B` are hexadecimal digits representing the amount of red, blue, and green, respectively.

If we modify the value of the color named `colorPrimary` to `#FF0000` (full red), the preview is updated as shown in **FIGURE 1.16**.

1.3.4 The `MainActivity` Class

The entry point for our app is the `MainActivity` class, whose name is composed of the names Main and Activity. It is located in the `com.jblearning.helloandroid` package in the java directory (shown in **EXAMPLE 1.9**).

The source code for an Android app uses Java syntax. Line 1 is the package declaration. When this project was created, we told Android Studio to place the code in the package `com.jblearning.helloandroid`. Lines 3–4 import the classes used in this class. We need to import `AppCompatActivity` because `MainActivity` extends it (line 6) and thus inherits from it. `AppCompatActivity` inherits (indirectly) from the `Activity` class and includes an action bar (in red on Figure 1.16) above the activity screen, where the app title shows.

An activity is a component that provides and controls a screen with which users can interact. An app can consist of several activities, possibly passing data to each other. Activities are managed

```
1  package com.jblearning.helloandroid;
2
3  import android.support.v7.app.AppCompatActivity;
4  import android.os.Bundle;
5
6  public class MainActivity extends AppCompatActivity {
7
8    @Override
9    protected void onCreate( Bundle savedInstanceState ) {
10     super.onCreate( savedInstanceState );
11     setContentView( R.layout.activity_main );
12   }
13 }
```

EXAMPLE 1.9 The `MainActivity` class

on a stack (a last-in, first-out data structure). To create an activity, we must subclass `Activity` or one of its subclasses as done in the skeleton code of `MainActivity`, which was automatically generated. Some methods of the `Activity` class are automatically called by the system when an activity starts, stops, resumes, or is destroyed.

The `onCreate` method (lines 8–12) is automatically called when the activity starts. We should override this method and if needed, create the components of the activity inside that method. At line 10, we call the `onCreate` method inherited from `Activity` via `AppCompatActivity`. At line 11, we define the layout of the app by calling the `setContentView` method, shown in **TABLE 1.1**. The layout is defined in the activity_main.xml file. The argument of the `setContentView` method is an integer that represents the id of a layout resource defining a screen layout. Ids of resources can be found in the R.java file inside the app/build/generated/source/r/debug/com/jblearning/helloandroid directory. The `R` class contains many inner classes: `anim`, `attr`, `bool`, `color`, `dimen`, `drawable`, `id`, `integer`, `layout`, `menu`, `string`, `style`, `styleable`. Each inner class is `public` and `static` and contains one or many `int` constants. Each `int` constant is accessible via the syntax:

```
R.className.constantName
```

The `activity_main` constant, the id for the activity_main.xml file, a layout resource, is defined in the `layout` class, inside the `R` class. Thus, it can be referenced using `R.layout.activity_main`.

TABLE 1.1 Selected `setContentView` methods of the `Activity` class

Method	Description
void setContentView(View view)	Sets the content of this Activity to view
void setContentView(int layoutResID)	Sets the content of this Activity using the resource with id layoutResID, which will be inflated

Calling `setContentView` with an integer argument that represents a layout XML file, in this case `R.layout.activity_main`, is called **inflating the XML**.

1.4 Running the App Inside the Emulator

Android Studio includes an emulator so that we can run and test an app inside the environment before testing it on a device. We first run the app in the emulator. In order for the emulator to work, we may need to do three things:

- If needed, enable the virtualization extension in the BIOS menu (if it is disabled; note that it may be enabled by default). When starting our computer, we need to interrupt the boot process, access the system's BIOS menu, and enable virtualization. If the computer does not support virtualization, it is unlikely that the emulator will work.
- If needed, download, install, and run the Intel Hardware Accelerated Execution Manager (HAXM). It may already be on our computer after we downloaded Android Studio. It is also possible that it needs to be updated. The file name should be intelhaxm-android.exe.
- Close and restart Android Studio.

Most, if not all, recent computers support virtualization.

The emulator runs inside an **Android Virtual Device (AVD)**. We can create AVDs with the **AVD Manager**. To open the AVD Manager, choose Window, AVD Manager, or click on the AVD Manager icon, as shown in FIGURE 1.17. This opens the AVD Manager, as shown in FIGURE 1.18. To create a new AVD, we click on the + Create Virtual Device. . . button. That opens a new panel (FIGURE 1.19) that allows us to select among a list of premade AVDs. After selecting one from the list and clicking on Next, it is displayed in a new panel (FIGURE 1.20). We select the version we want and

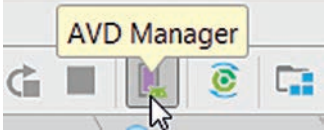

FIGURE 1.17 The AVD Manager icon

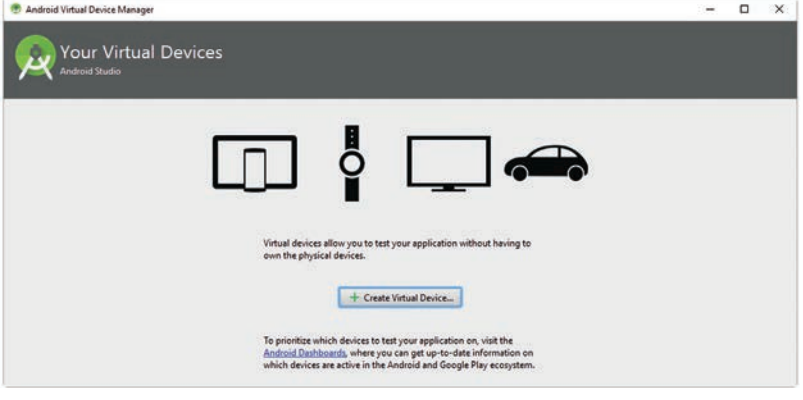

FIGURE 1.18 Opening the AVD Manager for the first time

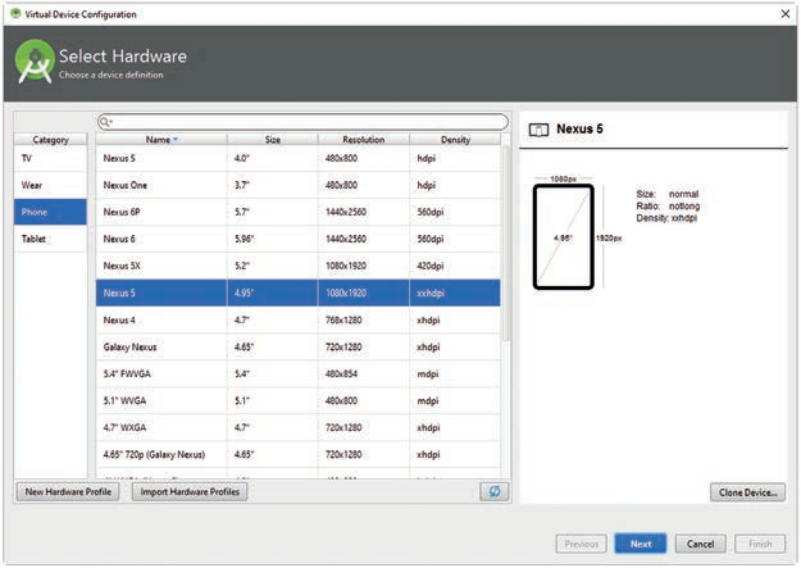

FIGURE 1.19 **Selecting a new AVD**

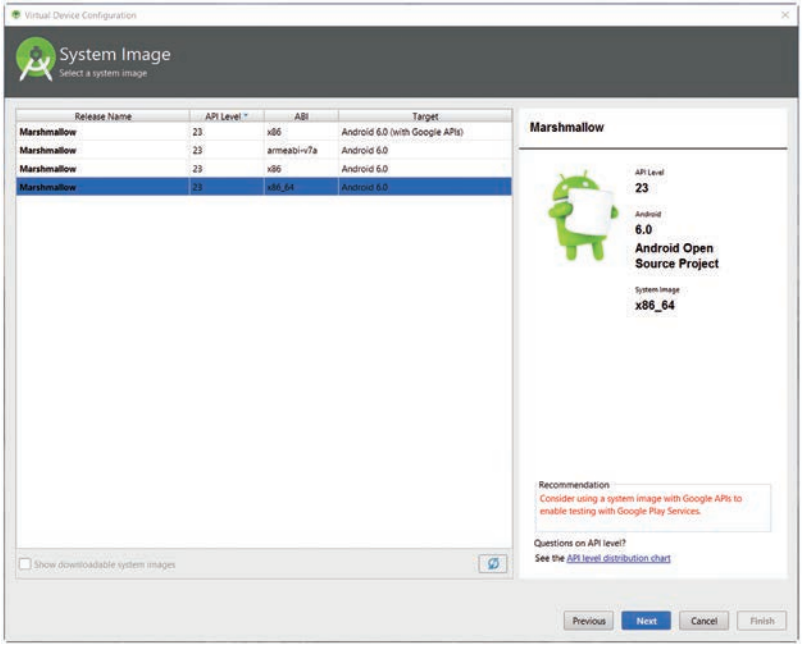

FIGURE 1.20 **After selecting an AVD**

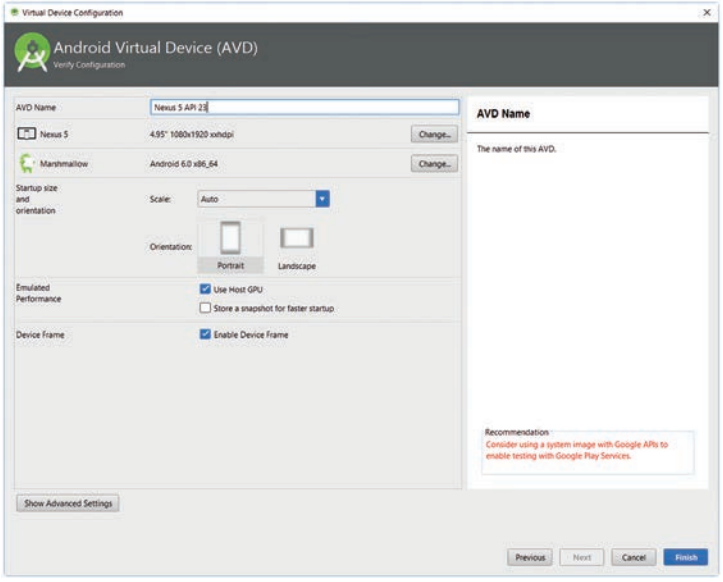

FIGURE 1.21 Characteristics of the selected AVD

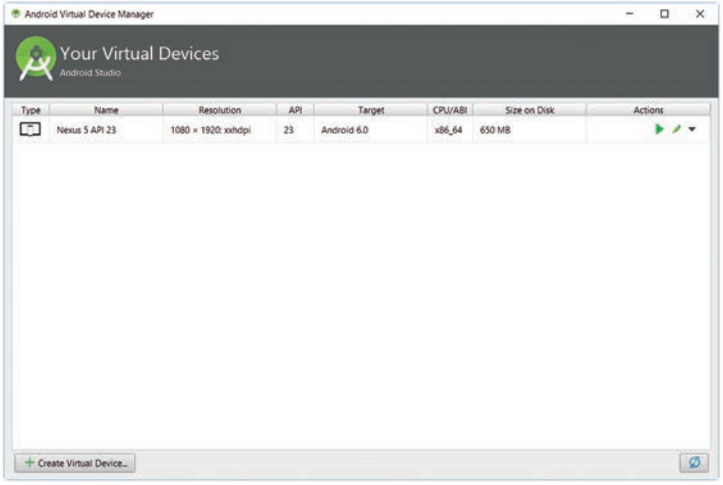

FIGURE 1.22 The AVD Manager showing the updated list of AVDs

click on next. The characteristics of the AVD are displayed, as shown in **FIGURE 1.21**. We can edit them if we want; in particular, we can edit the Scale attribute if we want to change the resolution of the AVD (and the size of the emulator as a result) when we run. After we click on Finish, the new AVD has been added, as shown in **FIGURE 1.22**.

In order to run the app, click on the Run icon in the icon bar, as shown in FIGURE 1.23.

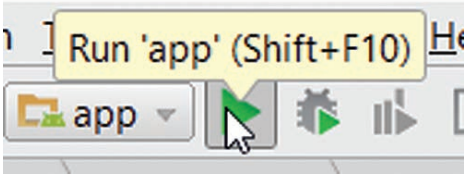

FIGURE 1.23 **Run configurations dialog box**

A dialog box opens (FIGURE 1.24) and we choose the AVD we want to use by selecting one from the list; then, we click on OK. We can also access the AVD Manager by clicking on the Create New Emulator button. Note that if the emulator is already running or an actual device is connected to our computer, it will show in the panel under Connected Devices. After a couple of minutes, the app should be running in the emulator. In order to save time, we recommend that you leave the emulator open as long as Android Studio is open.

SOFTWARE ENGINEERING: Leave the emulator open for the length of your development session in order to save time.

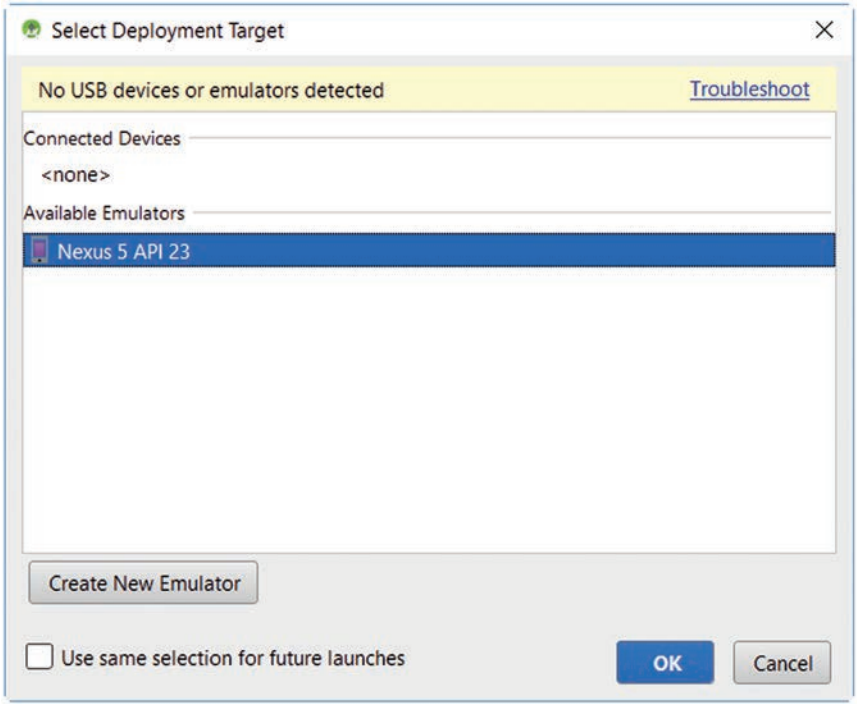

FIGURE 1.24 **Deployment target dialog box, after clicking on the Run icon**

FIGURE 1.25 **Emulator running the HelloAndroid app**

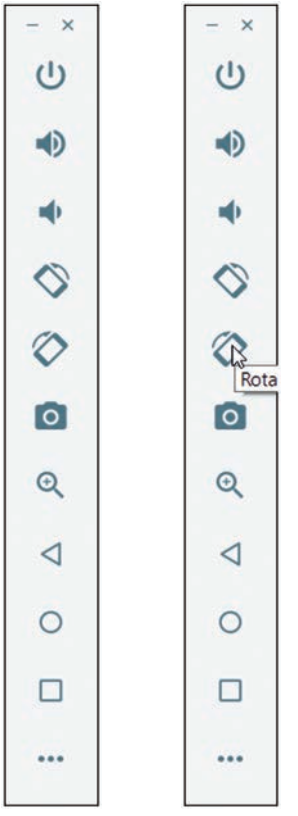

FIGURE 1.26 **The toolbar of the emulator, showing the Rotate tool**

FIGURE 1.25 shows the app running in the emulator. By default, the emulator runs in vertical, also called **portrait**, orientation. It includes a toolbar on its right that enables us to control some of its features. In particular, we can rotate the emulator using the tool shown in FIGURE 1.26. We can also click on Ctrl+F11 to rotate the emulator to the horizontal, or **landscape**, orientation (see FIGURE 1.27; note that we did not include the toolbar in this screenshot). Ctrl+F12 brings the emulator back to the vertical orientation. We can move the emulator by clicking on the top, white part of its frame and dragging it. TABLE 1.2 summarizes useful information about the emulator.

An interesting recent feature of Android Studio is **Instant Run**. With Instant Run, we can modify some selected components of an app, for example the strings.xml file, click on the Run icon, and the emulator automatically updates the app on the fly. Instant Run is enabled by default. If we want to disable it, we can choose File, Settings, and within the Build, Execution, Deployment section, select Instant Run, and then edit it, as shown in FIGURE 1.28. If we do not want to use Instant Run, we can uncheck it and check Restart activity on code changes.

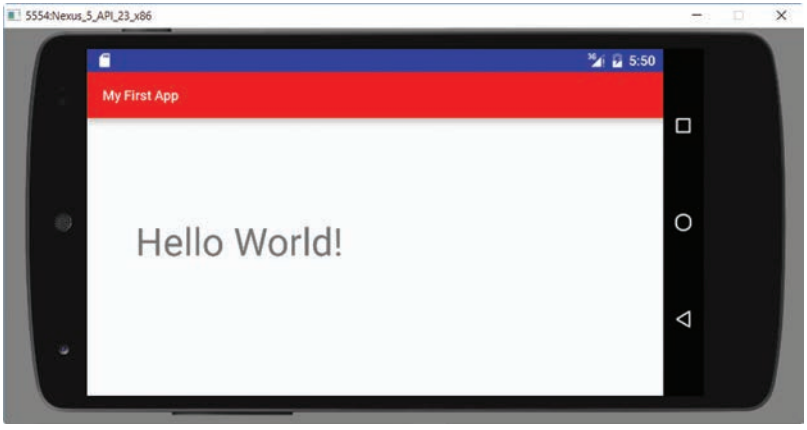

FIGURE 1.27 **Emulator running the HelloAndroid app in horizontal orientation**

TABLE 1.2 Useful information about the emulator

Turn on virtualization in the BIOS (if necessary)	Enable the emulator to run
Install and run the HAXM executable (if necessary)	Enable the emulator to run
Ctrl+F11 and Ctrl+F12	Rotate the emulator
Toolbar	Control features of the emulator

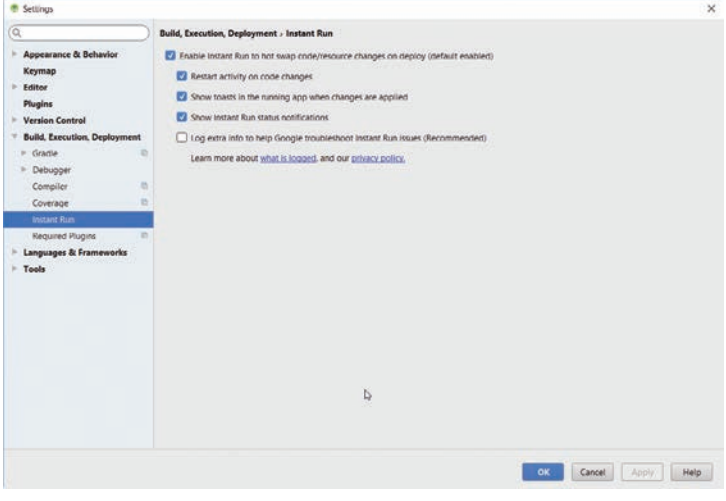

FIGURE 1.28 **The Settings panel showing the Instant Run capability**

1.5 Debugging the App with Logcat

Just like with a regular Java program, we can send output to the console, which can be very useful for debugging purposes. In order to do that, we can use one of the `static` methods of the `Log` class shown in TABLE 1.3. The `Log` class is part of the `android.util` package. This can be very useful at development time to provide us with feedback on various objects and variables as we test our app. Output from logging statements will show in the lower panel (click on the Android tab located at the bottom of the screen in order to open that panel) if we click on the **Logcat** tab, as shown in FIGURE 1.29. If the Logcat tab does not show, we should run the app in **Debug** mode first to force it to show. To run the app in Debug mode, click on the Debug button (the one with the bug icon, to the immediate right of the Run button) as shown in Figure 1.23. Logging statements should be commented out in the final version of the app.

The main difference in these methods is how verbose their output is. From the most to the least verbose, the order is: `v`, `d`, `i`, `w`, `e`.

There can be a lot of output inside Logcat. We can filter the output inside Logcat by using a filter so it only shows the output that we select from the app. To set up a filter, click on the drop-down list located on the top right corner inside the lower left panel; select Edit Filter Configuration (shown on the right side of FIGURE 1.30). Inside the dialog box shown on FIGURE 1.31, give the filter a tag. We choose `MainActivity`: in this way, it identifies that this output was generated

TABLE 1.3 Selected methods of the *Log* class

Method	Description
static int d(String tag, String msg)	This method sends a debug message; tag identifies the source of the log message; it can be used to filter messages in Logcat.
static int e(String tag, String msg)	This method sends an error message.
static int i(String tag, String msg)	This method sends an info message.
static int v(String tag, String msg)	This method sends a verbose message.
static int w(String tag, String msg)	This method sends a warning message.

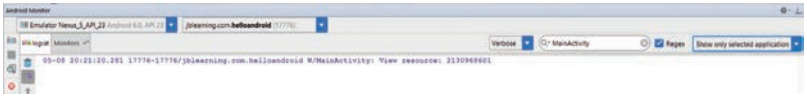

FIGURE 1.29 The lower panel—the Logcat tab and the MainActivity filter are selected

by some code inside the `MainActivity` class. When using the methods of the `Log` class to output data, we use `MainActivity` as the first argument (the tag) of those methods. In turn, when we run the app, if we select the filter named `MainActivity`, as shown in FIGURE 1.32, we only see the output that was tagged with `MainActivity`.

EXAMPLE 1.10 shows the updated `MainActivity` class, including a feedback output statement. The `Log` class is imported at line 5. We declare a constant named

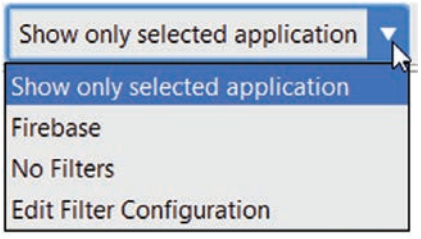

FIGURE 1.30 Opening the Logcat filter dialog box

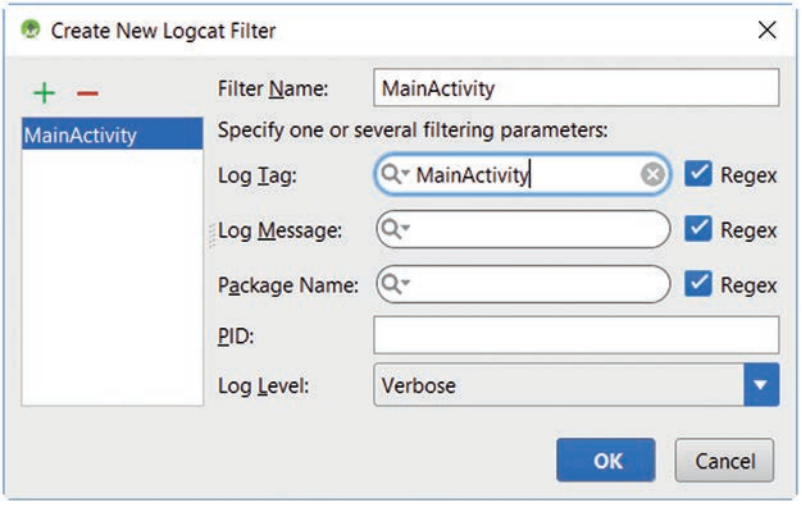

FIGURE 1.31 Setting up a Logcat filter

`MA` at line 8 to store the `String MainActivity`. The output statement at line 14 uses `MA` as its tag, and outputs the value of the resource `R.layout.activity_main`.

As we run the app, we see the result of our output statement in Figure 1.29. As the figure shows, the Logcat tab is selected (top left), and the `MainActivity` filter is selected (top right). We can verify that the id of the resource `R.layout.activity_main` (2130968601) matches the value found in the R.java file (hexadecimal number 0x7f040019).

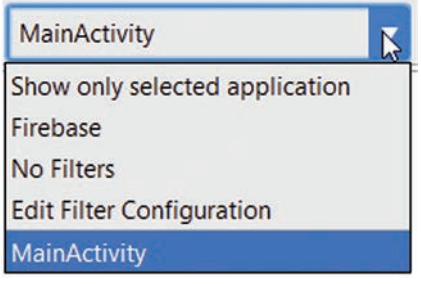

FIGURE 1.32 Selecting the MainActivity filter

```
1  package com.jblearning.helloandroid;
2
3  import android.support.v7.app.AppCompatActivity;
4  import android.os.Bundle;
5  import android.util.Log;
6
7  public class MainActivity extends AppCompatActivity {
8    public static String MA = "MainActivity";
9
10   @Override
11   protected void onCreate( Bundle savedInstanceState ) {
12     super.onCreate( savedInstanceState );
13     setContentView( R.layout.activity_main );
14     Log.w( MA, "View resource: " + R.layout.activity_main );
15   }
16 }
```

EXAMPLE 1.10 The modified `MainActivity` class, HelloAndroid, Version 1

As always with programming, we could have compiler errors, runtime errors, and logic errors. By default, Android Studio compiles our code as we type. It flags compiler errors with a small red line and warnings with a small orange line on the right margin of our code; it also shows a red tilde where the error is. For example, if we forget to include the semicolon at the end of line 14, Android Studio will signal the error as shown in FIGURE 1.33. If we move the mouse over the red line on the right margin or the red tilde, we see one or more suggestions to help us correct the error. In this case, we see:

```
';' expected
```

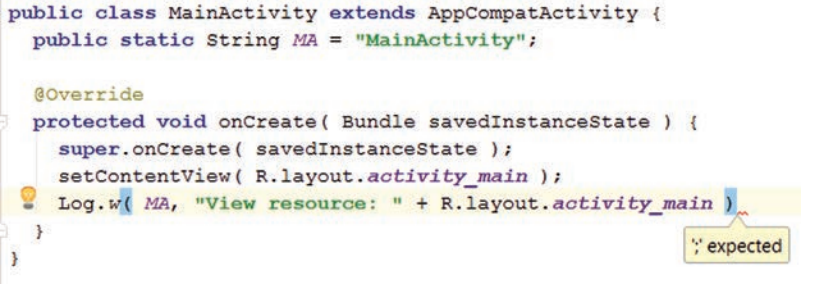

FIGURE 1.33 Compiler error flagged by Android Studio

The Android Studio includes many features that improve the programmer's experience, saves time by automatically generating code and prevents errors. Here are a few of these features:

- When we type a double quote, it automatically closes it by adding another one; furthermore, if we close it ourselves, it automatically deletes the extra double quote.
- If we use a class that has not been imported yet, it will underline it and suggests to import it by typing Alt+Enter, saving us some time.
- If we type an object reference and then a dot, it suggests methods of the object's class that we can call, and updates and restricts the list as we type.

SOFTWARE ENGINEERING: Use a filter to minimize the output inside Logcat.

1.6 Using the Debugger

In addition to Logcat, Android Studio includes traditional debugging tools, such as setting up breakpoints, checking the values of variables or expressions, stepping over code line by line, checking object memory allocation, taking screenshots and videos. In this section, we learn how to set up breakpoints and check the values of variables.

To set up a breakpoint, we click on the left of a statement—an orange filled circle appears. FIGURE 1.34 shows that we have set up two breakpoints.

To run the app in debug mode, we click on the debug icon on the toolbar, shown in Figure 1.23. The app runs and stops at the first breakpoint. The debugger tab is selected in the panel at the bottom of the screen and we can see some debugging information and tools. Under Frames, we can see where in the code we are currently executing. Under Variables, we can check the values of the various variables, for example `MA`, which has value `MainActivity`. To resume the app, we click on the green Resume icon at the top left of the panel, as shown in FIGURE 1.35.

```
@Override
protected void onCreate( Bundle savedInstanceState ) {
  super.onCreate( savedInstanceState );
  setContentView( R.layout.activity_main );
  Log.w( MA, "View resource: " + R.layout.activity_main );

  int count = 0;
  for( int i = 0; i < 3; i++ )
    count++;
}
}
```

FIGURE 1.34 **Breakpoints in the code**

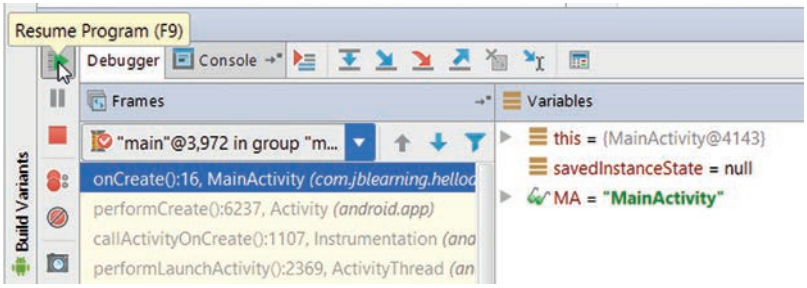

FIGURE 1.35 After stopping at the first breakpoint

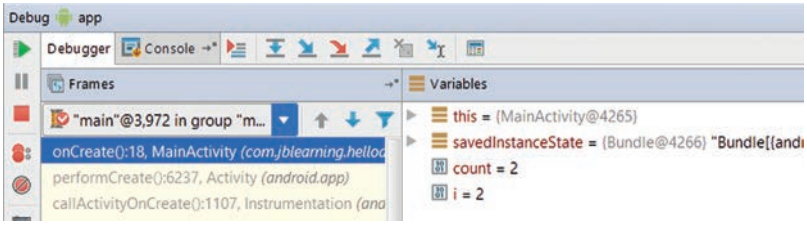

FIGURE 1.36 Breakpoints in the code

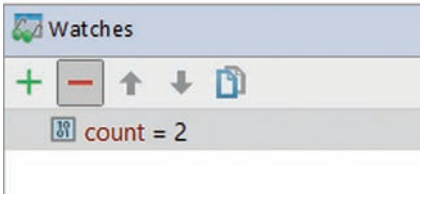

FIGURE 1.37 Managing a list of variables to watch

As we resume, stop at breakpoints, and resume the app a few times, the values of the various variables in our app are displayed under Variables. FIGURE 1.36 shows that `count` and `i` both have a value of `2` at that point.

If there are too many variables showing up under Variables and we only want to look at a few of them, we can make a list of variables in the right bottom panel, under Watches. We click on the + sign to add a variable to the list and click on the – sign to take a variable off the list. FIGURE 1.37 shows the value of count after we have added it to the list of watched variables.

1.7 Testing the App on an Actual Device

In order to test our app on an Android device, we need to:

- Download the driver for our Android device.
- Connect our Android device to our computer.

The driver for the Android device can be downloaded from the device manufacturer's website. For example, for Samsung's Android devices, the website is http://www.samsung.com/us/support/downloads#.

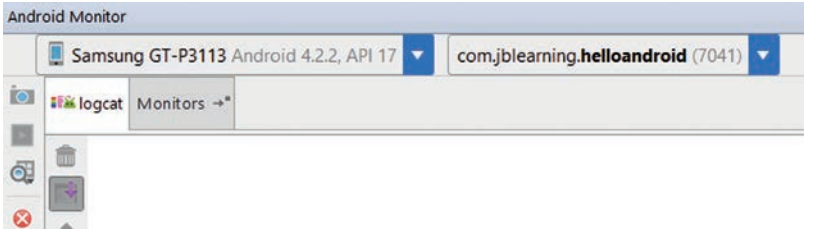

FIGURE 1.38 **The device's name and API level**

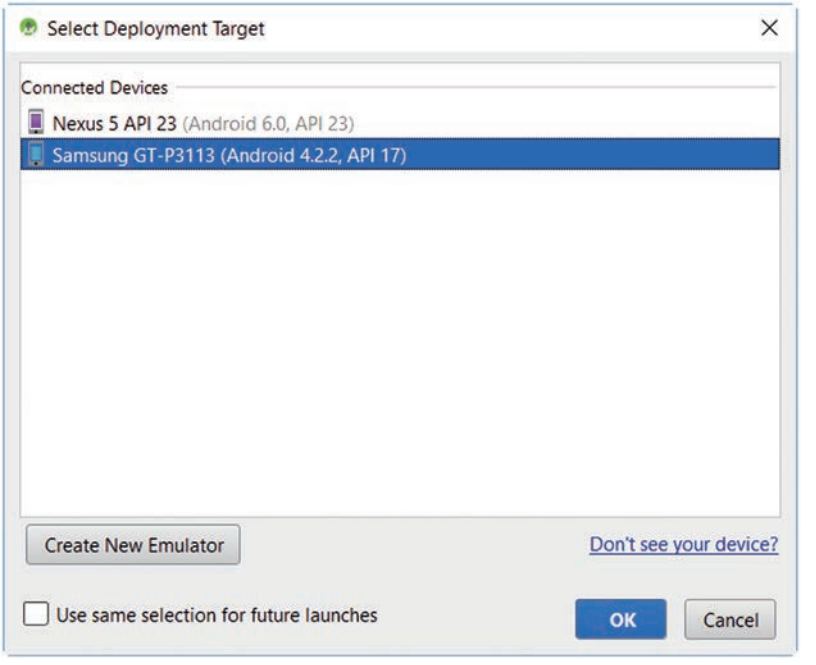

FIGURE 1.39 **Choose device dialog box**

Once we have done that, Android Studio will detect the connected device and display its name as well as its API level at the top left corner of the lower panel (assuming the lower panel is open, i.e., the Android tab is selected), as shown in **FIGURE 1.38**. We use a Samsung Galaxy Tab 2 7.0 tablet.

After we click on the Run or Debug button, a dialog box appears (**FIGURE 1.39**). The device appears in the list of devices or emulators we can run on. Select the device and click on OK. A few seconds later, the app runs (and is installed) on the device. Testing an app on an actual device is easier and faster than using the emulator. **FIGURE 1.40** shows the app running in the tablet. Note that we can still see the output in Logcat if we have output statements.

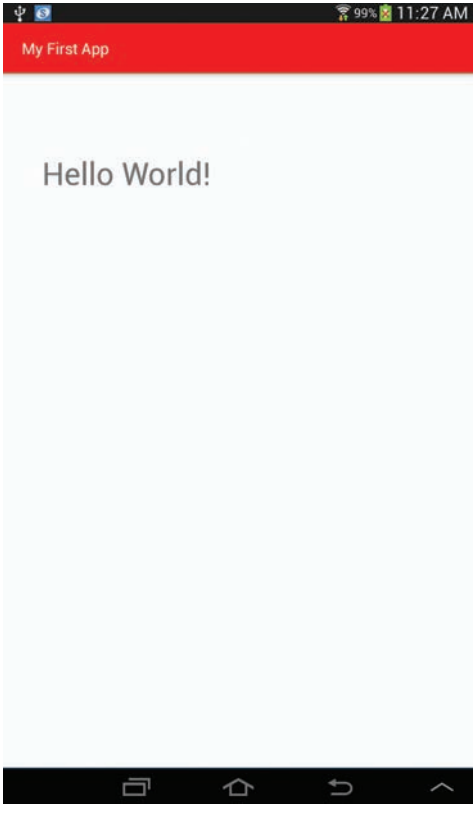

FIGURE 1.40 HelloAndroid app running inside the tablet

1.8 The App Manifest and the Gradle Build System

1.8.1 The AndroidManifest.xml File: App Icon and Device Orientation

The AndroidManifest.xml file, located in the manifests directory, specifies the resources that the app uses, such as activities, the file system, the Internet, and hardware resources. Before a user downloads an app on Google Play, the user is notified about these. **EXAMPLE 1.11** shows AndroidManifest.xml's automatically generated version. Among other things, it defines the icon (line 7) and the label or title for the app (line 8). The text inside the label is the `app_name String` defined in strings.xml.

We should supply a launcher icon for our app. This is the visual representation of our app on the Home screen or the apps screen. A launcher icon for a mobile device should be 48 × 48 dp. Various devices can have different screen densities, thus, we can supply several launcher icons,

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

  <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>
```

EXAMPLE 1.11 The AndroidManifest.xml file

one for each density. When doing that, we should follow the 2/3/4/6/8 scaling ratios between the various densities from medium (2) to xxx-high (8). If we only supply one icon, Android Studio will use that icon and expand its density as necessary. **TABLE 1.4** shows an example of possible dimensions using that rule. If we intend to publish, we should provide a 512 × 512 launcher icon for display in Google Play. We can find more information at https://www.google.com/design/spec/style/icons.html. **FIGURE 1.41** shows the mipmap directory after we added a file named hi.png whose dimensions are 48 × 48.

To set the launch icon for the app to hi.png, we assign the `String` `@mipmap/hi` to the `android:icon` attribute of the `application` element in the AndroidManifest.xml file. The

TABLE 1.4 Launcher icon ratios and possible dimensions

Density	Medium	High	X-High	XX-High	XXX-High
Scaling ratio	2	3	4	6	8
Dots per inch	160 dpi	240 dpi	320 dpi	480 dpi	640 dpi
Dimensions	48 × 48 px	72 × 72 px	96 × 96 px	144 × 144 px	192 × 192 px

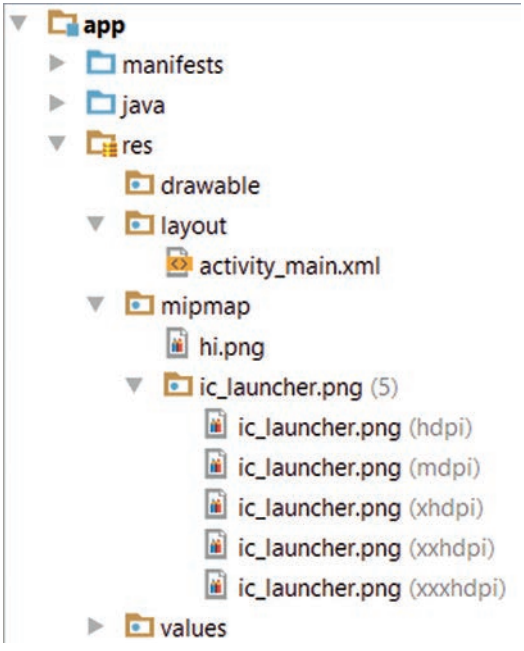

FIGURE 1.41 The hi.png file in the mipmap directory

`@mipmap/hi` expression defines the resource in the `mipmap` directory (of the `res` directory) whose name is `hi` (note that we do not include the extension). Line 7 of Example 1.11 becomes:

```
android:icon="@mipmap/hi"
```

One thing we have to worry about is that some users will use the app in vertical (portrait) orientation, and others will use the app in horizontal (landscape) orientation. The default behavior for an app is to rotate the screen as the user rotates the device, thus, our current app works in both orientations. **FIGURE 1.42** shows the app running inside the tablet in horizontal orientation. As an app gets more complex, this becomes an important issue. Later in the book, we discuss alternatives and strategies to manage orientations. Sometimes, we want the app to run in only one orientation, vertical for example, and therefore we do not want the app to rotate when the user rotates the device. Inside the `activity` element, we can add the `android:screenOrientation` attribute and specify either `portrait` or `landscape` as its value. For example, if we want our app to run in vertical orientation only, we add:

```
android:screenOrientation="portrait"
```

Note that we can control the behavior of the app on a per activity basis. In this app, there is only one activity, but there could be several. **EXAMPLE 1.12** shows the updated AndroidManifest.xml file. If we run the app on a device, the screen does not rotate as we rotate the device; it

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest package="com.jblearning.helloandroid"
3            xmlns:android="http://schemas.android.com/apk/res/android">
4
5    <application
6        android:allowBackup="true"
7        android:icon="@mipmap/hi"
8        android:label="@string/app_name"
9        android:supportsRtl="true"
10       android:theme="@style/AppTheme">
11     <activity android:name=".MainActivity"
12               android:screenOrientation="portrait">
13       <intent-filter>
14         <action android:name="android.intent.action.MAIN"/>
15
16         <category android:name="android.intent.category.LAUNCHER"/>
17       </intent-filter>
18     </activity>
19   </application>
20
21 </manifest>
```

EXAMPLE 1.12 **The modified AndroidManifest.xml file**

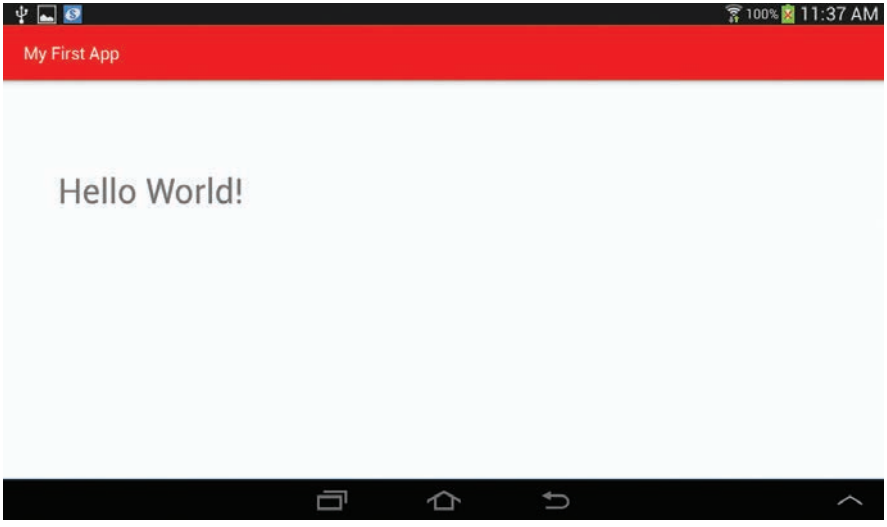

FIGURE 1.42 **HelloAndroid app, running inside the tablet in horizontal orientation**

FIGURE 1.43 The emulator screen showing the app icon

stays in portrait orientation. **FIGURE 1.43** shows the icon for the app in the emulator (among native apps).

1.8.2 The Gradle Build System

Android Application Package (APK), is the file format for distributing applications that run on the Android operating system. The file extension is .apk. To create an apk file, the project is compiled and its various parts are packaged into the apk file. At the time of this writing, the apk file can be found in the projectName/app/build/outputs/apk directory.

Apk files are built using the gradle build system, which is integrated in the Android Studio environment. When we start an app, the gradle build scripts are automatically created. They can be modified to build apps that require custom building, for example:

- Customize the build process.
- Create multiple apks for the app, each with different features, using the resources of the project. For example, different versions can be made for different audiences (paid versus free, consumer versus corporation).

Chapter Summary

- Android Studio is the official development environment for Android apps.
- In addition to the JDK, an Android SDK is available for development.
- Android Studio is free and available from Google.
- Android development uses XML files to define the Graphical User Interface (GUI), strings, styles, dimensions, etc.
- The res directory stores some of the app resources, such as activity_main.xml in the layout directory, and strings.xml, styles.xml, colors.xml, and dimens.xml in the values directory.
- The layout and GUI can be defined in XML files; the starting default file is activity_main.xml.
- `Strings` can be defined in the strings.xml file.
- Styles can be defined in the styles.xml file.
- Dimensions can be defined in the dimens.xml file.
- Colors can be defined in the colors.xml file.
- The R.java class is generated automatically and stores ids for various app resources.
- Android Studio includes a preview mode that can display the GUI for layout XML files.
- An activity is a component that provides a screen with which users can interact.
- An app can consist of several activities, possibly passing data to each other.
- Activities are managed on a stack (last-in, first-out data structure).
- To create an activity, we subclass the `Activity` class or an existing subclass of the `Activity` class.
- The `onCreate` method of the (entry point) `Activity` class is called automatically when the activity starts.
- We specify the View for an activity by calling the `setContentView` method.
- An app includes a manifest, the AndroidManifest.xml file, which defines the resources used by the app.
- The emulator enables us to simulate how an app would run on various devices.
- We can test an app in the emulator or on a device.
- The Instant Run feature enables us to modify various components of an app and run without restarting the app from scratch inside the emulator.
- When it is ready for installation, an app is packaged in a file with the .apk extension, known as the app's apk.
- Gradle is the build system for Android apps.

Exercises, Problems, and Projects

Multiple-Choice Exercises

1. AVD stands for
   - Android Validator
   - Android Virtual Device
   - Android Valid Device
   - Android Viral device
2. XML stands for
   - eXtended Mega Language
   - eXtended Multi Language
   - eXtensible Markup Language
   - eXtensible Mega Language
3. Mark the following XML snippets valid or invalid
   - `<a>hello</a>`
   - `</b>Hello<b>`
   - `<c digit="6"></c>`
   - `<d>He there</e>`
   - `<f letter='Z"/>`
   - `<1 digit="8">one</1>`
   - `<g digit1="1" digit2="2"></g>`
   - `<h><i name="Chris"></i></h>`
   - `<j><k name="Jane"></j>`
   - `<l><m name="Mary"></l></m>`
4. What is the name of the string that is defined by the following XML snippet inside strings.xml?

```
<string name="abc">Hello</string>
```

   - string
   - name
   - abc
   - Hello

5. What is the value of the string that is defined by the following XML snippet inside strings.xml?

```
<string name="abc">Hello</string>
```

- string
- name
- abc
- Hello

6. What will be the text displayed inside the TextView widget defined by the following XML snippet inside activity_main.xml?

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/hi" />
```

- @string/hi
- hi
- the value of the String named hi as defined in strings.xml
- android:text

7. The AppCompatActivity class is found in the package
   - android.app.Activity
   - android.app
   - android.Activity
   - android.support.v7.app
8. The AppCompatActivity class is a subclass of the Activity class
   - true
   - false

Fill in the Code

9. Inside the XML snippet next activity_main.xml, add a line of XML so that the text displayed in the text field is the value of the string book (assume that the string book has been defined in strings.xml)

```
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"

 />
```

10. Inside the onCreate method, fill in the code so that we set the layout and GUI defined in activity_main.xml

```
public void onCreate( Bundle savedInstanceState ){
       super.onCreate(savedInstanceState);
       // Your code goes here


}
```

Write an App

11. Write an app that displays "I like Android"
12. Write an app that only runs in horizontal orientation. It displays “This is fun!” on the top left corner of the screen with no margin either from the top or the left.