Chapter 2

Introduction to Android Application Architecture

The first chapter covered the environment and tools necessary to develop Android applications. This chapter will be a broad introductory tour of Android's application architecture. We will do that by doing three things. First, we will present the architecture of an Android app by building one. We will then present the essential components of Android architecture, namely, activities, resources, intents, activity life cycle, and saving state. We will conclude the chapter with a learning roadmap on how to use the rest of the book to create simple to sophisticated mobile apps.

In the **first section** of this chapter, a one-page calculator app will give you a bird's eye view of writing applications using the Android SDK. Creating this app will demonstrate how to create the UI, write Java code to control the UI, and build and deploy the app.

In addition to demonstrating the UI, this calculator app will introduce you to activities, resources, and intents. These concepts go to the heart of Android application architecture. We will cover these topics in detail in the **second section** of the chapter in order to give you a strong footing for understanding the rest of the Android SDK. We will also cover the activity life cycle and a brief overview of the persistence options for your application.

In the **third section** we will give you a roadmap for the rest of the book that addresses basic and advanced aspects of building Android applications. This final section breaks the chapters into a set of learning tracks. This section is a broad introduction to the entire set of Android APIs.

Furthermore, in this chapter you will find answers to the following: How can I create UI with a rich set of controls? How can I store state persistently? How can I read static files that are inputs to the app? How can I reach out and read from or write to the web? What other APIs does Android provide to make my app functional and rich?

Without further ado, let's drop you into the simple calculator application to open up the world of Android.

Exploring a Simple Android Application

The calculator application we want to demonstrate for this chapter is shown in Figure 2-1.

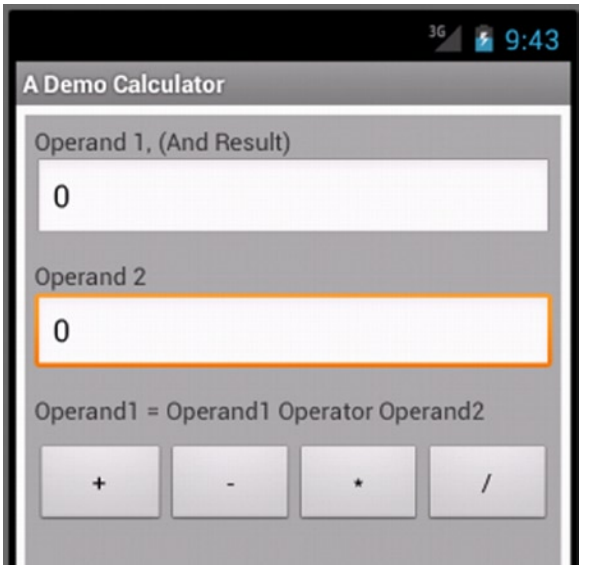

Figure 2-1. *A Calculator App*

Display in Figure 2-1 is called an activity in Android. This activity has two edit controls at the top representing two numbers. You can enter numbers in these edit boxes and use the operator buttons at the bottom of the figure to perform arithmetical operations. The result of an operation will be shown in the top edit control. These two edit boxes are labeled Operand 1 and Operand 2. To create this type of a calculator application using the Android SDK, you need to perform the following steps:

1. Create a User Interface (UI) definition in a text/xml file (called a layout or a layout file in Android).
2. Write programming logic in a Java file (usually in a class extending the base activity class).
3. Create a configuration file describing your application (this file is always called `AndroidManifest`.xml).
4. Create a project and a directory structure to place the files from steps 1, 2, and 3.
5. Build a deployable package using the project in step 4 (it is called an `.apk` file).

By going through the details of these steps you will get a feel for how Android applications are made. We will go through these steps now.

Defining UI through Layout Files

An Android application resembles a web application in lots of ways. In a web application the UI is your web page. UI of a web page is defined through HTML. An HTML web page is a series of controls like paragraphs, divisions, forms, buttons, etc. UI is constructed similarly in Android. A layout file in Android is like an HTML page, albeit the controls are drawn from the Android SDK instead of HTML. In Android this file is called a layout file. Listing 2-1 shows the layout file that produced the UI of Figure 2-1.

Listing 2-1. An Android Layout File that Defines UI for an Activity

```
<?xml version="1.0" encoding="utf-8"?>
<!--
*****************************************
* calculator_layout.xml
* corresponding activity: CalculatorMainActivity.java
* prefix: cl_ (Used for prefixing unique identifiers)
*
* Use:
*    Demonstrate a simple calculator
*    Demonstrate text views, edit text, buttons, business logic
*****************************************
-->
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:layout_margin="5dp" android:padding="5dp"
    android:background="@android:color/darker_gray"
    >
    <!--  Operand 1 -->
    <TextView android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Operand 1, (And Result)"
        />
    <EditText android:layout_width="match_parent" android:layout_height="wrap_content"
        android:id="@+id/editText1"  android:text="0"
        android:inputType="numberDecimal"/>
    <!--  Operand 2 -->
    <TextView android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Operand 2"
        android:layout_marginTop="10dp"
        />
    <EditText android:layout_width="match_parent"
       android:layout_height="wrap_content"
       android:text="0"
       android:id="@+id/editText2"
       android:inputType="numberDecimal">
    </EditText>
```

```
    <!--  Buttons for Various Operators -->
    <TextView android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Operand1 = Operand1 Operator Operand2"
        android:layout_marginTop="10dp"
        />
<LinearLayout
    android:orientation="horizontal"
    android:layout_marginTop="10dp"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <Button android:text="+" android:id="@+id/plusButton"
       android:layout_weight="1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content">
    </Button>
    <Button android:text="-" android:id="@+id/minusButton"
       android:layout_weight="1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content">
    </Button>
    <Button android:text="*" android:id="@+id/multiplyButton"
       android:layout_weight="1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content">
    </Button>
    <Button android:text="/" android:id="@+id/divideButton"
       android:layout_weight="1"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content">
    </Button>
</LinearLayout>
</LinearLayout>
```

Let's go through this calculator XML layout file of Listing 2-1 line by line. This file looks complicated compared to Figure 2-1. Yes, it is verbose, but you will see shortly it is simple in its architecture.

Specifying Comments in Layout Files

As a good practice the comments at the top of the layout XML file in Listing 2-1 indicate what this file name is, what UI activity will be used to display this file, what the purpose of this file is, and briefly what controls are in this layout file.

Adding Views and View Groups in Layout Files

Each XML node in a layout file represents a UI control. These controls can be either views or containers of other views. A container of other views is called a `ViewGroup`. For example, a button is a view. A `LinearLayout` in Listing 2-1 is a `ViewGroup` that places all its child views either vertically down or horizontally across. So, a `LinearLayout` is like an HTML `div` that lays out its children either across or down.

Specifying Control Properties in Layout Files

The UI controls in the calculator layout file are `LinearLayout`, `TextView`, `EditText`, and a `button`. Each of these controls represents a Java object when painted on the screen. Being an object, each of these controls has properties. If the controls belong to the core Android SDK, their properties are prefixed with `"android:"` as in `"android:orientation"` for the `LinearLayout` control. The majority, if not all, of the controls that you normally use in your apps are from the core Android SDK. When you write your own controls they are called custom controls. These custom controls allow you to define custom properties. See the "Roadmap" section of this chapter for more on custom controls.

Indicating ViewGroup Properties

Some of the control properties are labeled as `"android:layout_,"` such as `android:layout_width`. These properties, although mentioned in a given XML node, like a `button`, are read and used by parent node, like `LinearLayout`, to place the children. Parent nodes are view groups like the `LinearLayout`. You can see this difference in how `padding` and `margins` are defined for the first `LinearLayout` node in the layout file of listing 2-1. The property `padding` belongs to the topmost `LinearLayout` object in this example, whereas the property for margins of that same topmost `LinearLayout`, the `layout_margin` property, belongs to the parent of the `LinearLayout`, which is an implicit view group provided by the Android framework. So for padding you say `android:padding`, and for margins you say `android:layout_margin`. Notice the presence or lack of `"layout_"` prefix. If you want to know what properties an object (or control) supports, you can use `Ctrl-Space in eclipse` to see a set of suggestions for the properties for that object. Depending on your development environment you can easily find an equivalent set of key combinations to do the same.

Controlling Width and Height of a Control

Two often-used properties for a control are its layout width and layout height. The layout parent of a control manages these values. Values for these properties are typically `match_parent` and `wrap_content`. If you say your `TextView` is set to `match_parent` for its width, the width of the control matches up with the parent width. When a `TextView` is set for its height `wrap_content`, then its height will be just sufficient to contain all its text in the vertical direction. Of course, these two properties are available to all child controls of a layout, not just the text control. These two layout control properties, `match_parent` and `wrap_content`, also apply to the height of a control as well.

Introducing Resources and Backgrounds

Although we are in the middle of explaining the controls in the layout file, this is a good place to introduce resources. Layout files are, and are made up of, resources. In the calculator layout file, we have set the background of the entire view by setting the background on the root `LinearLayout` control. This instruction looked like the following:

```
android:background="@android:color/darker_gray"
```

Every view or control in Android supports the `background` property. Backgrounds are usually identified as resources. In this example, the background is pointing to a resource, coming from the Android package, which is of type `color` whose referenced value is darker gray.

A number of inputs to your application are represented as resources in Android. Some example resources are image files, entire layout files, colors, strings, XML files, menus, and many other things as listed in the Android SDK. For instance, the entire calculator layout file we are talking about is itself a resource.

As you can see from the calculator layout file, resources are of different types. In Android they are further broadly classified as "value based" or "file based." Examples of resources that are values are strings and colors. Examples of resources that are files are images or layout files. Listing 2-2 shows an example of creating value-based resources that are strings and colors.

Listing 2-2. Example of Value-Based Resources

```
<?xml version="1.0" encoding="utf-8"?>
<!-- this file will be in /res/values subdirectory  -->
<resources>
    <string name="hello">Hello World, CalculatorMainActivity!</string>
    <string name="app_name">A Demo Calculator</string>
    <color name="red">#FF0000</color>
    <color name="blue">#0000FF</color>
</resources>
```

You can have any number of value-based files as long as they are all under the `/res/values` subdirectory. Each file will start with the `resources` root node. You can use `Ctrl-Space` to discover what other possible value-based resources are available.

Turning to file-based resources, Listing 2-3 shows an example of placing a number of file-based resources under their respective resource subdirectories.

Listing 2-3. Example of File-Based Resources

```
/res/layout/page1_layout.xml (A layout file for say page 1)
/res/drawable/page1_background.jpg (An example image file)
/res/drawable-hdpi/page1_background.jpg (Same image file for a different density)
/res/xml/some_preferences.xml (example of an input file for your app)
```

Any of these resources, be it file based or value based, can be referenced in the layout files using the "@" resource reference syntax. For example, in the calculator layout file in Listing 2-1 the background can be set literally and explicitly as a color value between the quotes, such as `"#FFFFFF,"` or point to a resource reference (indicated by a starting `@color/red`) that is already defined as a color resource (as in Listing 2-2). In this syntax led by "@," the type of referenced resource is "color." Some of the key words for other types of resources are `string` (for strings), `drawable` (for images), etc.

In Listing 2-1, the way to read the value of the `background` property of the `LinearLayout`, namely, `@android:color/darker_gray`, is as follows: Use the value of the resource identified as `darker_gray` in the Android core framework and whose resource type is `color`. With this knowledge of resource reference syntax, take a look at the calculator layout file listing

one more time and you will be able to read it where each control has properties and each property has a value that is either directly specified or references a resource that is elsewhere defined in a resource file.

The indirection of a control property value defined as a resource reference has an advantage. A resource can be customized for languages, device density variation, and a variety of factors without altering the compiled Java source code. For example when you supply background images you can place a number of these images in different directories and name them using the convention specified by Android. Then Android knows how to locate the right image, given its name, depending on the device your app is running on.

Working with Text Controls in the Layout File

In the calculator layout example we have used two text-based controls. One is a `TextView` control, which is used as a label; the other is an `EditText` control, which is used for taking input text. We have already shown you how to set the width and height of any view by using the attributes that start with "`layout_`". Every text-based control also has an attribute called `text`. In our examples we have directly specified the literal text as a value for this property. The recommendation is to use instead a resource reference. For example:

```
android:text="Literal text"  //what we did for clarity
or
android:text="@string/LiteralTextId" //doing it properly
```

The latter resource ID, `LiteralTextId`, then can be defined in a file in the `/res/values` subdirectory much like in Listing 2-2.

`EditText` control in the calculator layout has an attribute `inputType` to provide the necessary constraints and validations that need to take place when data is typed into the editable field. Refer to the documentation to see a large number of constraints that are available for editable fields. Alternatively, you can use eclipse ADT to discover the available input types on the fly during coding.

Working with Autogenerated IDs for Controls

To manipulate the controls that are in the calculator layout of Listing 2-1, we need a way to turn them into Java objects. This is done by locating these controls using a unique ID in the currently loaded layout file of an activity. Let's look at one example in the layout file where an `EditText` control is given an ID of `editText2` as follows:

```
android:id="@+id/editText2"
```

This format tells Android that `ID` of this `EditText` control is a resource of type `ID` and its integer value should be known in Java as `editText2`. The + is a convenience to allocate a new unique integer for `editText2`. If you don't have the + sign, then Android looks for an integer-valued resource defined with an ID that is called `editText2`. With the convenience of + we can avoid separately defining a resource first and then use it. In some cases, you may have a need for a well-known ID that is shared by multiple pieces of code, in which case you

will remove the + and take the multiple steps of defining the ID first and then using its name in multiple places. You will see in the programming logic section (soon to follow) how these control IDs are used to locate the controls and manipulate them.

Implementing Programming Logic

To see the calculator layout on the screen of your device, you need a Java class derived from the Android SDKs class `activity`. Such an `activity` represents a window in your mobile application. So you need to craft a calculator activity by extending the Android base `activity` class as shown in Listing 2-4.

Listing 2-4. Programming Logic: Implementing an Activity Class

```
/**
 * Activity name: CalculatorMainActivity
 * Layout file: calculator_layout.xml
 * Layout shortcut prefix for ids: cl_
 * Menu file: none
 * Purpose and Logic
 * ******************
 * 1. Demonstrate business logic for a simple calculator
 * 2. Load the calculator_layout.xml as layout
 * 3. Setup button callbacks
 * 4. Respond to button clicks
 * 5. Read values from edit text controls
 * 6. Perform operation and update result edit control
 */
public class CalculatorMainActivity extends Activity
implements OnClickListener
{
    private EditText number1EditText;
    private EditText number2EditText;

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.calculator_layout);
        gatherControls();
        setupButtons();
    }
    private void gatherControls()   {
        number1EditText = (EditText)this.findViewById(R.id.editText1);
        number2EditText = (EditText)this.findViewById(R.id.editText2);
        number2EditText.requestFocus();
    }
    private void setupButtons()    {
        Button b = (Button)this.findViewById(R.id.plusButton);
        b.setOnClickListener(this);

        b = (Button)this.findViewById(R.id.minusButton);
        b.setOnClickListener(this);
```

```
    b = (Button)this.findViewById(R.id.multiplyButton);
    b.setOnClickListener(this);

    b = (Button)this.findViewById(R.id.divideButton);
    b.setOnClickListener(this);
}
@Override
public void onClick(View v)     {
    String sNum1 = number1EditText.getText().toString();
    String sNum2 = number2EditText.getText().toString();
    double num1 = getDouble(sNum1);
    double num2 = getDouble(sNum2);
    Button b = (Button)v;

    double value = 0;
    if (b.getId() == R.id.plusButton)   {
        value = plus(num1, num2);
    }
    else if (b.getId() == R.id.minusButton)   {
        value = minus(num1, num2);
    }
    else if (b.getId() == R.id.multiplyButton)   {
        value = multiply(num1, num2);
    }
    else if (b.getId() == R.id.divideButton)   {
        value = divide(num1, num2);
    }
    number1EditText.setText(Double.toString(value));
}

private double plus(double n1, double n2)    {
    return n1 + n2;
}
private double minus(double n1, double n2)    {
    return n1 - n2;
}
private double multiply(double n1, double n2)    {
    return n1 * n2;
}
private double divide(double n1, double n2)    {
    if (n2 == 0)    {
        return 0;
    }
    return n1 / n2;
}
private double getDouble(String s)    {
    if (validString(s))        {
        return Double.parseDouble(s);
    }
    return 0;
}
```

```
    private boolean invalidString(String s)    {
        return !validString(s);
    }
    private boolean validString(String s)    {
        if (s == null)     {
            return false;
        }
        if (s.trim().equalsIgnoreCase(""))    {
            return false;
        }
        return true;
    }
}
```

In this listing, the calculator activity is called `CalculatorMainActivity`. Once you have this activity, you can load the calculator layout into it in order to see the calculator screen of Figure 2-1.

Let's learn a bit about an activity in Android. A programmer does not need to instantiate an activity directly. An activity can be instantiated by the Android framework based on user's actions. In that sense, an activity is a "*managed component*" managed by Android.

An activity can get partially hidden or completely hidden when another UI with higher priority sits on top of it (for example, due to a phone call). Or, an activity that is in the background can be temporarily removed due to memory constraint. In these circumstances, the activity can be automatically brought back when a user revisits the application.

Loading the Layout File into an Activity

As the activity is event driven, an activity relies on callbacks. The first callback of importance is the `onCreate()` callback. In the calculator activity given in Listing 2-4, you can easily locate this method. This is where we will load the calculator layout into the calculator activity. This is done through the method `setContentView()`. The input to this method is an identifier for the calculator layout file.

A nice feature of Android is what it does with the various resources including the layout files. It autogenerates a java class called `R.java` where it defines integer IDs for all the resources be they value based or file based. In the activity given in Listing 2-4, the variable `R.layout.calculator_layout` points to the calculator layout file (which itself is in Listing 2-1).

When you are dipping your toes into the Android framework, the other mysterious thing in `onCreate()` is the `savedInstanceBundle`. As the Android framework may stop and restart (even re-create) activities, it needs a way to pass the last state of the activity to the `onCreate()` method. That is what the `savedInstanceBundle` is. It is a collection of key value pairs holding the previous state of the activity. You will learn about this aspect of state management in more detail later in the chapter, and also in Chapter 9, where we cover what happens when a device is rotated. For the implementation of the calculator example we simply call the super class's method to pass on that state bundle.

Gathering Controls

The next two methods, `gatherControls()` and `setupButtons()`, set up the interaction model for the calculator. In the `gatherControls()` method you obtain java references for the edit controls that you need to manipulate (read or write to) and save them locally in the calculator activity class. You do this by using the `findViewById()` method on the base activity class. The `findViewById()` method takes as input the ID of the control that is in the layout file. Here also Android autogenerates these IDs and places them into the `R.java` class. In your eclipse project you can see this file in the `/gen` subdirectory. Listing 2-5 shows the generated `R.java` file for this calculator project. (If you were to try this project yourself, these IDs may differ. So use this listing primarily to understand concepts.)

Listing 2-5. Autogenerated Resource IDs: R.java

```
public final class R {
    public static final class attr {
    }
    public static final class drawable {
        public static final int background=0x7f020000;
        public static final int icon=0x7f020001;
    }
    public static final class id {
        public static final int divideButton=0x7f050005;
        public static final int editText1=0x7f050000;
        public static final int editText2=0x7f050001;
        public static final int minusButton=0x7f050003;
        public static final int multiplyButton=0x7f050004;
        public static final int plusButton=0x7f050002;
    }
    public static final class layout {
        public static final int calculator_layout=0x7f030000;
    }
    public static final class string {
        public static final int app_name=0x7f040001;
        public static final int hello=0x7f040000;
    }
}
```

Notice how `R.java` uses a different class prefix for each resource type. This allows the programmer in eclipse to quickly separate the IDs by what their type is. So, for example, all IDs for layout files are prefixed with `R.layout`, and all image IDs are prefixed with `R.drawable`, and all strings with `R.string`, etc. However, there is a caution while working these IDs. Even if you have ten layout files the IDs for all the controls are generated into a single namespace such as R.id.* (where “id” is an example of a resource type). So you may want to get into the habit of naming controls in the layout files with some prefix indicating which layout files they belong to.

Setting Up Buttons

Some of the controls in the calculator layout of Listing 2-1 are the calculator buttons. They are the buttons representing operators: +, -, x, and /. We need code to be invoked when these buttons are pressed. The way to do that is by registering a callback object on the button controls. These callback objects must implement the `View.OnClickListener` interface. The calculator activity in addition to extending the `activity` class also implements the `View.OnClickListener` interface, allowing us to register our activity as the one that needs to be called back when each button is pressed. As you can see in the code of the activity (Listing 2-4), this is done by calling the `setOnClickListener` on each button.

Responding to Button Clicks: Tying It All Together

When any of the operator buttons is clicked, the `onClick()` method in the calculator activity given in Listing 2-4 gets called. In this method we will investigate the ID of the `view` that called back. This calling view should be one of the buttons. In this method we will read the values from both the `EditText` controls (the operand values) and then invoke a method that is specific to each operator. The operator method will calculate the result and update the `EditText`, which is labeled Result.

Updating the AndroidManifest.XML

So far we have the UI (in terms of the layout file) and we have the business logic in terms of the calculator activity. Every Android app must have its configuration file. This file is called the `AndroidManifest.xml`. This is available in the root directory of the project. Listing 2-6 shows the `AndroidManifest.xml` for this project.

Listing 2-6. Application Configuration File: AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.androidbook.calculator"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="14" />
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".CalculatorMainActivity"
            android:theme="@android:style/Theme.Light"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>
</manifest>
```

The package attribute of this manifest file follows a naming structure similar to the java namespaces. In the calculator app, the package is set to `com.androidbook.calculator`. This is like giving a name and a unique identifier to your app. Once you sign this app and install it on an app publisher like the Google Play Store, only you will be able to update it or release subsequent versions of it. The `uses-sdk` directive indicates the API for which this app is backward compatible. The application node has a number of properties including its label and an icon that will show up in the Android device apps menu. Inside an application node we need to define all of the activities that make up this application. Each activity is identified by its respective java classname. If the activity classname is not fully qualified, then the java package is assumed to be the same as the application package identified. Theme for the activity indicates a set of properties that the views belonging to that activity will inherit. It is like setting a CSS style on the HTML UI. Android comes with a few default styles. Choosing a light theme is good for contrast while taking screen shots (as shown in Figure 2-1). Chapter 7 is dedicated to using styles and themes in your apps.

In the Android application manifest file an activity can specify a series of intent filters. `Intent` is a programming concept that is unique to Android. Android relies heavily on these intents. Android uses intent objects to invoke application components including activities. An intent object can contain an explicit activity classname so that when you invoke that intent you end up invoking the activity. Or instead of having an explicit classname, an intent can indicate a generic action like VIEW to view a web page. When you invoke such an intent with a generic action Android will present all possible activities that can satisfy that action. Activities register with Android through the manifest file that they can respond to some actions through an intent filter. Listing 2-7 shows how you can invoke an activity through an intent object.

Listing 2-7. Using an Intent Object to Invoke an Activity

```
//currentActivity refers to the activity in which this code runs
Intent i = new Intent(currentActivity,SomeTargetActivity.class);
currentActivity.startActivity(i); //start the target activity
```

Although we used `currentActivity` as the value of the first argument to create an intent, all it needs is a base class reference called Context. A `context` reference represents the application context in which a component like an activity runs. Coming back to the intent object, it has a number of flags and extra data elements that you can use to control the behavior of the target activity that the intent is invoking. Listing 2-8 shows an example.

Listing 2-8. Using Extras on an Intent Object

```
//currentActivity refers to the activity in which this code runs
Intent intent = new Intent(currentActivity,SomeTargetActivity.class);
intent.setFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP
                | Intent.FLAG_ACTIVITY_SINGLE_TOP);
intent.putExtra("some-key", "some-value");
currentActivity.startActivity(intent);
```

In this example, we want the target activity to be brought to the top of the window or activity stack and close any other activities that were on top of it before. As one invokes activities from other activities they sit on top of each other. This stack allows the back button to navigate back to the previous activity in the stack. When you go back, the current top activity is finished and the previous activity is shown in the foreground. The code in Listing 2-8 is like going back to the last position of the target activity, making that the top instance, and removing/finishing all recent activities above that. `Extras` on intent are a set of key value pairs that you can pass to the target activity from the source activity. Activities are pretty isolated from each other. They don't share their local variables between each other. Instead, they should pass their data through objects that can be serialized and deserialized. Android uses an interface similar to `Serializable` called `Parcelable` that allows greater flexibility and efficiency.

Ultimately, every activity is almost always started by an intent object. You can get the intent object anywhere in your target activity by calling `getIntent()`. Once you get the intent object, you can get its extras and see if there is any pertinent data that you need.

Complete study of intents and their variations is a large topic. We will return to talk more about intents later in this chapter after concluding our discussion on the calculator app. We have also included a URL for the free dedicated chapter on intents from our previous editions at the end of this chapter.

Placing the Files in the Android Project

Let's return to our main line of thought, the calculator app. By now you have the three files you need to create the calculator application. Use what you have learned in the first chapter to create an empty Android project and adjust that project to place these three files. These files are given in Listing 2-9 along with their parent directories.

Listing 2-9. Placement of Files for the Calculator App

```
/res/layout/calculator_layout.xml
/src/com/androidbook/calculator/CalculatorActivity.java
/AndroidManifest.xml
```

Figure 2-2 shows the structure of your Android project in eclipse. You can see the relative locations of the files of Listing 2-9.

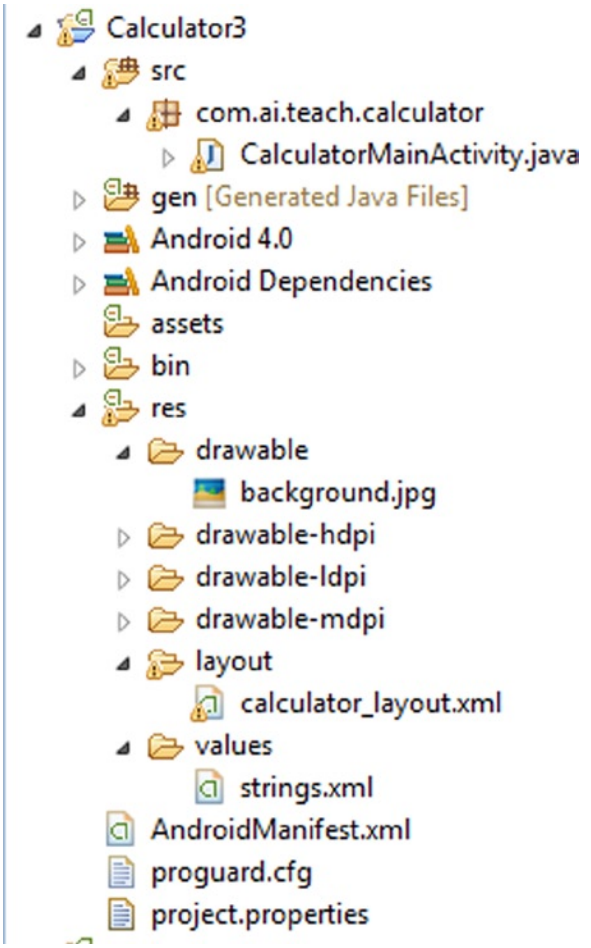

Figure 2-2. *A calculator app directory structure*

The directory structure in Figure 2-1 also shows you where other resources like images and strings are placed. You can also see the directory structure for the device-dependent image files in Figure 2-2. This is how Android solves the localization multilingual support as well by using different resource subdirectory suffixes. You will learn about the other subdirectories of an Android project as you go through this book.

Testing the Calculator App on a Real Device

All that is left now is to build the APK file, sign it, and be ready to deploy. The simplest way to test your project is to have eclipse deploy the APK to the emulator and test it. The simplest way to test this file (once it is signed) on a device is to e-mail it to yourself and open the e-mail on your device. There is a security setting on the device to allow APKs from unverified sources. As long as this is allowed, you will be able to install the APK file and run it on your device. Or you can also connect the device to the USB port and have eclipse deploy the APK directly to the device. You can even debug it on the device through eclipse. You can also copy the APK file from your PC or Mac to the device SD card and install it from there.

This concludes our section on the calculator app, which illustrated the nature of Android apps. We will move to the second section of the chapter now where we will talk about activities in a lot more depth and also revisit resources, intents, and saving state. Let's start with activities.

Android Activity Life Cycle

An Android activity is a self-standing component of an Android application that can be started, stopped, paused, restarted, or reclaimed depending on various events including user-initiated and system-initiated ones. So it is really important to review the architecture of the life cycle of an activity by looking at all of its callbacks. Figure 2-3 shows the life cycle of an activity by documenting the order of its callbacks and the circumstances under which those callbacks are executed. Let's consider these callback methods one by one.

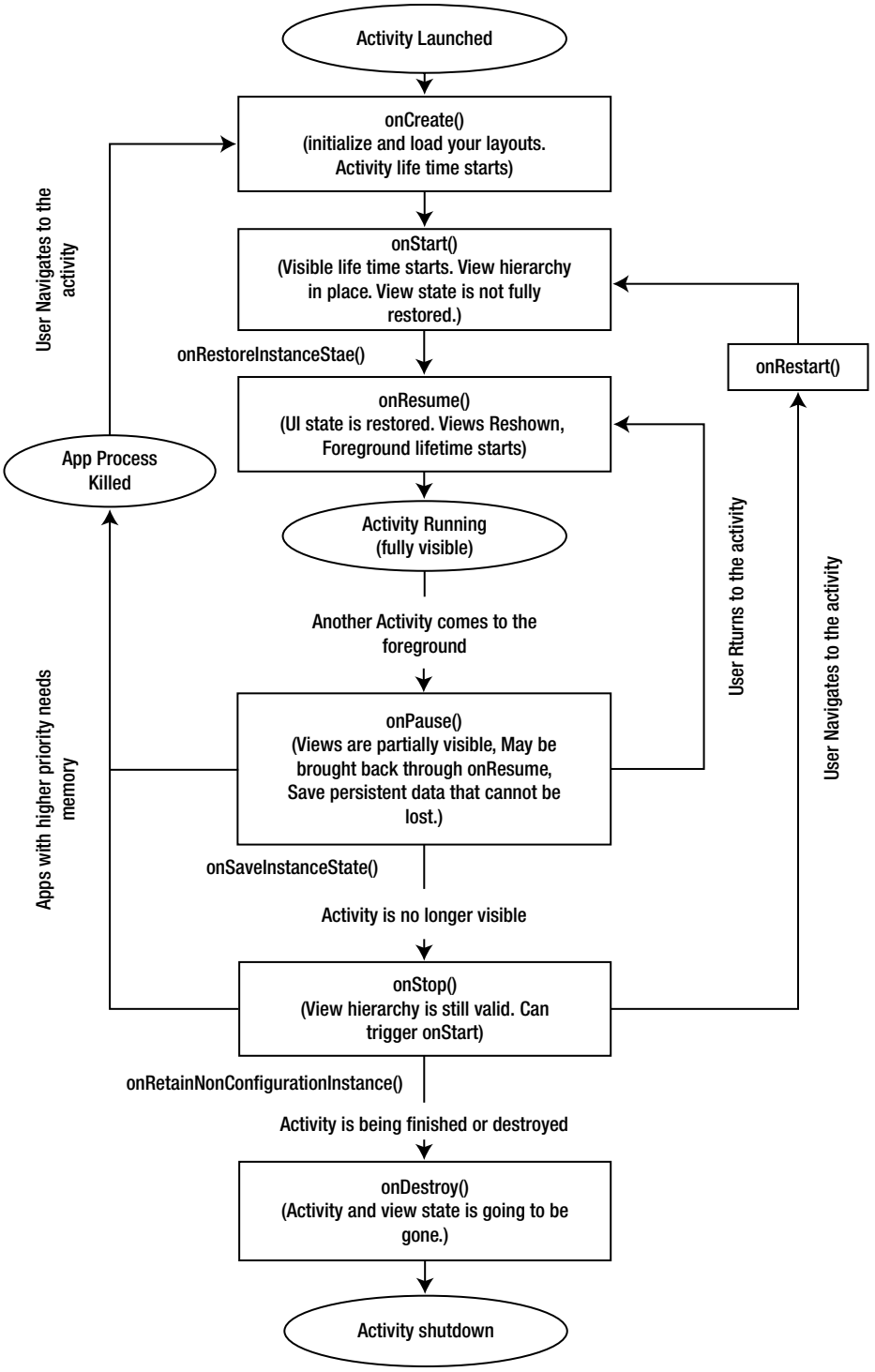

Figure 2-3. Annotated Android activity life cycle

void onCreate(Bundle savedInstanceState)

The activity's life cycle starts with this method. In this method you should load your view hierarchies by loading the layouts into the content view of the activity. You also initialize any activity level variables that may be used during the lifetime of the activity. Like many of the callbacks you also call the parent's `onCreate()` method first.

When `onCreate` is called, an activity may be in one of three states. The activity may have been a brand-new activity starting out its life for the first time. Or it may be an activity that is automatically restarted because of a configuration change such a device rotating from one orientation to another. Or it is an activity that is restarted following a previous process shutdown due to low-memory conditions and being in the background. In the `onCreate` callback, you should take these scenarios into account if what you need to do in each scenario is different.

Now we can understand the argument to this method involving the `savedInstanceBundle`. You can use this bundle to look into the previous state of the activity. This bundle may have been originally used to save the state of the activity during a configuration change or when the activity and its process shut down due to low-memory conditions. The state that is saved into this bundle argument is usually called the *instance state* of the activity. The instance state is somewhat temporary in nature; specifically, it is tied to this instance of the application during this invocation. This type of state is not expected to be written to permanent storage like files. The user will not be too disconcerted if this state is to revert to an initial state when the application is revived. In the callback we will explain soon called `onPause()` you can save the state that must be persisted to long-term storage. If that happens, you can use the `onCreate()` method to load that state as well as part of the start-up.

There is another consideration that this method can take into account. When an activity is restarted or re-created because of an orientation change, the old activity is destroyed and a new activity is created in its place. This means the new activity has a new reference in memory. The old activity reference is no longer valid. It would be wrong to have an external thread or a global object that is holding onto the old activity. So there needs to be a mechanism when the activity is re-created to tell the external object that there is a new activity reference. To do that, the re-created activity needs to know the reference of that external object. This external object reference is called "non-configuration instance reference." There is a callback method called `onRetainNonConfigurationInstance()` that can return a reference to this external object; we shall cover this shortly. Android SDK then keeps this reference and makes it available to the re-created activity through a method called `getLastNonConfigurationInstance()`. Note that in Chapter 8, we will show you how to do this better through what are called headless retained fragments. We will return to this topic also in Chapter 15 on AsyncTask.

There is another nuance to the `onCreate` method. You may want to ensure that in the layouts you have right views and fragments (which you will learn in Chapter 8) to match when the state was saved. Because a subsequent `onRestoreInstanceState()` (which is called after `onStart()`) assumes that all the view and fragment hierarchies are present to restore their respective states, the mere presence of the previous state will not re-create the views. So it is up to this method to load the right layouts to be shown. This is usually not an issue if you don't delete or add views during the interaction with the activity.

void onStart()

After being created, this method pushes the activity into a visible state. In other words this method starts the "visible life cycle" of the activity. This method is called right after `onCreate()`. This method assumes that the view hierarchies are loaded and available from the `onCreate()`. You normally don't need to do override this method and if you do, make sure you call the parent's `onStart()` first. In Figure 2-2 note that this method can also be called from another callback called `onRestart`.

You must be aware that the `onRestoreInstanceState` method is called after this method. So you shouldn't make assumptions about the state of the views in this method. So try not to manipulate the state of the views in this method. Do that refinement in the subsequent `onRestoreInstanceState` or the `onResume` method. Because this is a counterpart of the `onStop()`, do the reverse should you have stopped something in `onStop()` or in `onPause()`. If you see something is being done in this method, look at it with caution and make sure it is what you want. Also know that the start-and-stop cycle can happen multiple times during the overall current cycle of the activity.

This method can also be called when the activity is shown after being hidden first, because another activity has come to the top of the visibility stack. In those cases this method is called after the `onRestart()`, which itself is triggered after the `onStop()`. So there are two paths into this method: either `onCreate()` or `onRestart()`. In both cases the view hierarchies are expected to be established and available prior to this callback.

void onRestoreInstanceState(Bundle savedInstanceState)

If an activity is to be legitimately closed by a user, then the state that user is willing to discard is the instance state. For example, when the user chooses a back button, then he/she is informing Android that he/she no longer is interested in this activity and that the activity can be closed, discarding all its state that is not yet saved. So this state, which is transitory and only valid for the life of the activity while it is in memory, is the instance state.

If the system choses to close the activity, because there is a change in orientation, then the user will expect that transitory (instance) state right back when the activity is restarted. To facilitate this, Android calls this `onRestoreInstanceState` method with a bundle that contains the saved instance state. (See the `onSavedInstanceState` method explanation.)

In contrast to instance state, the persistent state of an activity is something the user expects to see even after the activity finishes and is no longer in play. This persistence state may have been created during the activity or may even exist before the activity is created. This type of state, especially when it is created with the help of the activity, must be explicitly saved to an external persistent store like a file. If the activity doesn't use an explicit "save" button for such needs, then the "`onPause`" method needs to be used to save such implicit persistent states. This is because no method after `onPause` is guaranteed to be called in case of low-memory conditions. You shouldn't rely on the instance state if the information is too important to lose.

void onResume()

The callback method `onResume` is the precursor to having the activity fully visible. This is also the start of the foreground cycle for the activity. In this foreground cycle the activity can move between `onResume()` and `onPause()` multiple times as other activities, notifications, or dialogs with more urgency come on top and go.

By the time this method is called, we can expect the views and their state fully restored. You can take this opportunity to tweak final state changes. As this method doesn't have a bundle, you need to rely on the information from `onCreate` or `onRestoreInstanceState` methods to fine-tune state if further needed.

If you had stopped any counters or animations during `onPause` you can restart them here. You can also keep track of the case if the views are really destroyed or not by following the previous callback methods (whether `onResume` is a result of `onCreate`, `onRestart`, or `onPause`) and do the minimum possible to adjust the view state. Typically you will not do state management here but only those tasks that need to be turned on or off based on visibility.

void onPause()

This callback indicates that the activity is about to go into background. You should stop any counters or animations that were running when the activity was fully visible. The activity may go to `onResume` or proceed to `onStop`. Going to `onResume` will bring the activity to the foreground. Going to `onStop` will take the activity into a background state.

As per the SDK, it is also the last method that is guaranteed to be called before the activity and the process is completely reclaimed. So it is the last opportunity that a developer has to save any non-instance and persistent data to a file.

Android SDK also waits for this method to return before making the foreground activity fully active. So you want to be brief in this method. Also notice that this method has no bundle that is passed. This is an indication that this method is for storing persistent data and also in an external storage medium such as a file or a network.

You can also use this method to stop any counters, animations, or status displays of a background task. You can resume them in `onResume`.

void onStop()

The callback method `onStop()` moves the activity from partially visible to the background state while keeping all of the view hierarchies intact. This is the counterpart of `onStart`. The activity can be taken back to the visible cycle by calling `onStart`. This state transition of going from `onStop` to `onStart` during the same activity life cycle goes through the `onRestart()` method.

After this call, the activity is no longer visible. But keep in mind that this may not be called after `onPause` under low-memory conditions. Because of this uncertainty do not use this method to start or stop services that are outside of this process. Do that in `onPause` instead and resume them in `onResume`. However, you can use this method to control services or work

that is inside your process. This is because, as long as the process is active, this method is going to get called. If the whole process is taken down then those dependent tasks or global variables will go away anyway.

void onSaveInstanceState(Bundle saveStateBundle)

The control goes to `onDestroy()` coming out of `onStop` if the process is still in memory. However, if Android realizes that activity is being closed without the user's expectation then it would call the `onSaveInstanceState()` before calling `onDestroy()`. Orientation change is a very concrete example of this. The SDK warns that the timing of `onSaveInstanceState()` is not predictable whether before or after `onStop()`.

The default implementation of this method already saves the state of views. However if there is some explicit state that is not known to the views you need to save it in the bundle object and retrieve it back in the `onRestoreInstanceState` method. You do need to call the parent's `onSaveInstanceState()` method first so that views have an opportunity to save their state themselves. There are some restrictions and rules for the views to be able to save their state. The chapters on UI controls (Chapters 3, 4, and 5) and configuration change (Chapter 9) go into more detail on this subject.

void onRestart()

This method is called when the activity transitions from background state to partially visible state, i.e., going from `onStop` to `onStart`. You can use this knowledge in `onStart` if you want to optimize code there based on whether it is a fresh start or a restart. When it is a restart the view and their state are fairly intact.

You can do things in this method that would have been done in `onStart`, but optimized when the activity is not visible, but too expensive to be done multiple times in `onResume`.

Object onRetainNonConfigurationInstance()

This callback method is in place to deal with activity re-creation due to configuration changes. This method returns an object reference in your process memory that needs to be retied to the activity once it is re-created. We explained this in more detail previously when we described the `onCreate` method.

When the activity is re-created, the object that is returned from this method is made available through the method `getLastNonConfigurationInstance()`. Now in `onCreate()` the new activity can use the previously established resources and object references. Importantly, if those previous resources are holding to the old activity reference, then the resources can be told to use the new one.

This dilemma exists because during an orientation change Android doesn't kill the process, but just discards the old activity, re-creates the activity in the new orientation, and expects the programmer to supply new layouts, etc., to suit the new configuration. So the working objects are still there holding onto an old activity. This is the method in association with its "get" counterpart to overcome this obstacle.

When you read Chapter 8, you will learn that this method is deprecated and you will use in its place what are called headless retained fragments. These headless retained fragments have the additional benefit of being able to track the activity life cycle and not just the reference to the activity.

void onDestroy()

`onDestroy()` is the counterpart of `onCreate()`. The activity is going to finish after onDestroy. An activity can finish for two primary reasons.

One is an explicit close. This can happen when the user has explicitly caused the activity to finish either by clicking a button that is provided to indicate that the user is done or by using a back button leaving the activity to go to the previous activity. Under such circumstances the activity will not be brought back by the system unless the user chooses the activity again. In this scenario the activity life cycle ends with the onDestroy method.

The second reason an activity can close is involuntary. When the orientation of a device changes, the Android SDK will forcefully close the activity and call the `onDestroy` method followed by re-creating the activity and calling the `onCreate` again.

When an activity is in the background and if the system needs memory, Android may shut down the process and may not have an opportunity to call the `onDestroy` method. Due to this uncertainty, much like onStop, don't use this method to control tasks or services that are outside the process in which the activity had been running. However, if the process is still in memory the onDestroy will be called as part of the life cycle and you can place cleanup code in onDestroy as long as that code belongs to this process.

General Notes on Activity Callbacks

Use Figure 2-3 to guide you to see the order of these callbacks and how best to use them. If you were to override a callback you need to call back the parent method. The SDK documentation explicitly indicates which derived methods are required to call back their parent equivalents. Also refer to the SDK documentation to learn during which callbacks the system will not kill the process due to low-memory conditions. Also notice that only a handful of callbacks carry instance state bundle.

More on Resources

We want to tell you little more about how resources are used in Android applications. In the calculator layout file, you have seen some of the resources used like strings, images, IDs, etc.

Other resources that are not so obvious include dimensions, drawables, string arrays, language terms for plurals, xml files, and all types of input files. In Android, something is treated as a resource a) if it is an input to your program and is part of the apk file and b) if the value or content of the input can have different values based on language, locale, or orientation of the device, generally called a configuration change.

Directory Structure of Resources

All resources in Android are placed under the `/res` subdirectory of the root of your application package. Listing 2-10 shows an example of what a `/res` may look like:

Listing 2-10. Android Resource and Assets Directory Structure

```
/res/values/strings.xml
           /colors.xml
           /dimens.xml
           /attrs.xml
           /styles.xml
    /drawable/*.png
              /*.jpg
              /*.gif
              /*.9.png
              /*.xml
    /anim/*.xml`
    /layout/*.xml
    /raw/*.*
    /xml/*.xml
/assets/*.*/*.*
```

We will cover `attrs.xml` and `styles.xml` in Chapter 7. The xml files in the `anim` subdirectory define animations that can be applied to various views. We will cover these animation-related resources in the animations chapter (Chapter 18). The xml files in the xml subdirectory get compiled away to binary and their resource IDs can be used to read them. We will show an example of this shortly. The `/raw` subdirectory holds files that get placed, as they are, without getting converted to any binary format.

The `/assets` directory, which is a sibling of `/res`, is not part of the resource hierarchy. This means that the files in this subdirectory do not change based on language or a locale. Android does not generate any IDs for these files. This directory is more like a static local storage for any files that are used as inputs, such as configuration files for your application.

Except for the `assets` directory, every other artifact in the `/res` subdirectory will end up generating an ID in `R.*` namespace, as you have seen before. Each distinct resource type will have its own namespace under `R.*`, as in `R.id`, `R.string`, or `R.drawable`, etc.

Reading Resources from Java Code

In layout files, as you have seen the in calculator layout, one resource can refer to other resources. For example, the calculator layout resource file referenced the string and color references. This approach is common. Alternatively, you can also use Java API to retrieve the resource values using the method `Activity.getResources()`. This method returns a reference to the Android SDK java class `resources`. You can use methods on this class to get to the values of each resource identified in your local `R.*` namespace. Listing 2-11 shows an illustration of this approach:

Listing 2-11. Reading Resource Values in Java Code

```
Resources res = activity.getResources();
//Retrieving a color resource
int somecolor  = res.getColor(R.color.main_back_ground_color);
// Using a drawable resource
ColorDrawable redDrawable=(ColorDrawable)res.getDrawable(R.drawable.red_rectangle);
```

Runtime Behavior of Drawable Resources

The drawable directory is an interesting case worth covering to demonstrate the fluency of Android's architecture. As shown earlier, this directory can contain images that can be set as backgrounds. This directory also allows XML files that know how to get converted to `drawable` java objects which can then be used as backgrounds that are rendered at runtime. Listing 2-12 shows an example of this:

Listing 2-12. Example of a Shape Drawable XML Resource File

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape="rectangle">
    <solid android:color="#f0600000"/>
    <stroke android:width="3dp" android:color="#ffff8080"/>
    <corners android:radius="13dp" />
    <padding android:left="10dp" android:top="10dp"
        android:right="10dp" android:bottom="10dp" />
</shape>
```

If you place a file like this in the drawable subdirectory and call it `background1.xml`, it will result in an ID called `R.drawable.background1`. You can then use that ID as if it were a background image for any view that is drawn with a rectangular border. Other possible shapes are ovals, lines, and rings.

Similar to the shape xml file, each allowed XML file in the drawable directory defines a drawable that defines a particular way to draw. Examples of these drawables include bitmaps that can be decorated with certain behavior, or images that can transition from one image to another, layered drawables that are collections of other drawables, drawables that can be selected based on input parameters, drawables that can respond to progress by showing multiple images, drawables that can clip other drawables, etc... See the following URL for a number of sophisticated things you can do using these runtime drawable objects:

http://androidbook.com/item/4236

Using Arbitrary XML Files as Resources

Android also allows arbitrary XML files to be used as resources which can then be localized or tuned for each device. Listing 2-13 is an example of reading and processing an XML-based resource file from the `/res/xml` subdirectory.

Listing 2-13. Reading an XML Resource File

```
private String readAnXMLFile(Activity activity) throws XmlPullParserException, IOException {
   StringBuffer sb = new StringBuffer();
   Resources res = activity.getResources();
   XmlResourceParser xpp = res.getXml(R.xml.test);

   xpp.next();
   int eventType = xpp.getEventType();
    while (eventType != XmlPullParser.END_DOCUMENT) {
        if(eventType == XmlPullParser.START_DOCUMENT) {
           sb.append("******Start document");
        }
        else if(eventType == XmlPullParser.START_TAG)  {
           sb.append("\nStart tag "+xpp.getName());
        }
        else if(eventType == XmlPullParser.END_TAG) {
           sb.append("\nEnd tag "+xpp.getName());
        }
        else if(eventType == XmlPullParser.TEXT) {
           sb.append("\nText "+xpp.getText());
        }
        eventType = xpp.next();
    }//eof-while
    sb.append("\n******End document");
    return sb.toString();
}//eof-function
```

Working with Raw Resource Files

Android also allows any type of non-compiled files as resources. Listing 2-14 is an example of reading a file that is placed in the `/res/raw` subdirectory. Being a resource even the raw files that are in this directory can be customized for language or a device configuration. Android generates IDs automatically for these files as well, as they are resources like any other resource.

Listing 2-14. Reading a Raw Resource File

```
String getStringFromRawFile(Activity activity) throws IOException {
      Resources r = activity.getResources();
      InputStream is = r.openRawResource(R.raw.test);
      //assuming you have a function to convert a stream to a string
      String myText = convertStreamToString(is);
      is.close(); //take care of exceptions etc.
      return myText;
}
```

Reading Files from the Assets Directory

Although usually clubbed with resources, the `/assets` directory is a bit different. This directory does not sit under the `/res` path, so the files in this directory do not behave like resource files. Android does not generate resource IDs for these files in the `R.*` namespace. These files are not customizable based on locale or device configuration. Listing 2-15 shows an example of reading a file that is placed in the `/assets` subdirectory.

Listing 2-15. Reading a File from Assets Directory

```
String getStringFromAssetFile(Activity activity) {
    AssetManager am = activity.getAssets();
    InputStream is = am.open("test.txt");
    String s = convertStreamToString(is);
    is.close();
    return s;
}
```

Thus far, we have used an activity reference to get hold of the resources or an `AssetManager` object as in Listing 2-15. In reality, all we need is the base class of the activity, the `context` object.

Reading Resources and Assets Without an Activity Reference

Sometimes you may need to read an XML resource file or an asset file from the bowels of source code, where it is intrusive to pass the activity reference. For these cases, you can use the following approach to obtain the application context and then use that reference instead to get to the assets and resources.

When Android loads your application (in order to invoke any of its components), it instantiates and calls an application object to inform that the application could initialize itself. This application classname is specified in the Android manifest file. If `MyApplication.java` is your application java class, then it can be specified in the Android manifest file as shown in Listing 2-16.

Listing 2-16. Specifying an Application Class in the Manifest File

```
<application android:name=".MyApplication"
        android:icon="@drawable/icon" .../>
```

Listing 2-17 shows how we can code the `MyApplication` and also shows how we can capture the application context in a global variable.

Listing 2-17. Sample Code for an Application that Captures Application Context

```
public class MyApplication extends Application {
   //Make sure to check for null for this variable
   public static volatile Context s_appContext = null;

   @Override
   public void onConfigurationChanged(Configuration newConfig) {
      super.onConfigurationChanged(newConfig);
   }
   @Override
   public void onCreate() {
      super.onCreate();
      MyApplication.s_appContext = this.getApplicationContext();
   }
   @Override
   public void onLowMemory() {
      super.onLowMemory();
   }
   @Override
   public void onTerminate() {
      super.onTerminate();
   }
}
```

With the application context captured in a global variable, we now can get to the asset manager to read our assets as in Listing 2-18.

Listing 2-18. Using Application Object to Get to Application Asset Files

```
AssetManager am = MyApplication.s_appContext.getAssets();
InputStream is = am.open(filename);
```

Understanding Resource Directories, Language, and Locale

Let's wrap up the idea of Android resources by pointing out how resource directories are used to load resources based on language, locale, or a configuration change of the device like say orientation. See how in Listing 2-19 a layout file with the same name is located in multiple layout directories starting with same prefix of layout but with different qualifiers such as "`port`" for portrait and "`land`" for landscape. There are a large number of these qualifiers available in the SDK documentation. We also cover some of these aspects in Chapter 9 (Configuration Changes). Listing 2-19 shows an example of how layout files are arranged by portrait or landscape configuration:

Listing 2-19. Demonstrating Resource Qualifiers

```
\res\layout\main_layout.xml
\res\layout-port\main_layout.xml
\res\layout-land\main_layout.xml
```

More on Intents

We have talked about how intents are used to invoke activities. We want cover few more essential aspects of intents now. Listing 2-20 shows how intents are used to invoke a number of prebuilt Google applications.

Listing 2-20. Sample Code Using Intents

```
public class IntentsUtils {
    public static void invokeWebBrowser(Activity activity)    {
       Intent intent = new Intent(Intent.ACTION_VIEW);
       intent.setData(Uri.parse("http://www.google.com"));
       activity.startActivity(intent);
    }
    public static void invokeWebSearch(Activity activity)    {
       Intent intent = new Intent(Intent.ACTION_WEB_SEARCH);
       intent.setData(Uri.parse("http://www.google.com"));
       activity.startActivity(intent);
    }
    public static void dial(Activity activity)    {
       Intent intent = new Intent(Intent.ACTION_DIAL);
       activity.startActivity(intent);
    }
    public static void call(Activity activity)    {
       Intent intent = new Intent(Intent.ACTION_CALL);
       intent.setData(Uri.parse("tel:555-555-5555"));
       activity.startActivity(intent);
    }
    public static void showMapAtLatLong(Activity activity)     {
       Intent intent = new Intent(Intent.ACTION_VIEW);
       //geo:lat,long?z=zoomlevel&q=question-string
       intent.setData(Uri.parse("geo:0,0?z=4&q=business+near+city"));
       activity.startActivity(intent);
    }
}
```

Notice how these intents do not invoke a specific activity by its classname but rather use the target qualities of suitable activities. For example to invoke a browser to view a web page, the intent simply says the action is `ACTION_VIEW` and the data portion of the intent is set to the web address. Android then looks around to see all the activities that know how to show the type of data requested in the data attribute. It will then give the user an option which of the activities that the user wants to choose to open the URL. These types of intents that don't specify the classname of the component to invoke are called implicit intents. We will cover this in a little bit more detail shortly.

Starting Activities for Results

Listing 2-21 shows an example of an activity where one of its methods is invoking a target activity in order to obtain a result when that target activity is completed. This is done through the method `invokePick()` in as shown in Listing 2-21.

Listing 2-21. Using Intents to Get Results from Activities

```
public class SomeActivity extends Activity {
.....
//Call this method to start a target activity that knows how to pick a note
//Use a data URI that tells the target activity which list of notes to show
public static void invokePick(Activity activity) {
  Intent pickIntent = new Intent(Intent.ACTION_PICK);
  int requestCode = 1;
  pickIntent.setData(Uri.parse(
     "content://com.google.provider.NotePad/notes"));
  activity.startActivityForResult(pickIntent, requestCode);
}

//the following method will be called when the target activity finishes
//Notice the outputIntent object that is passed back which could
//contain additional information

@Override
protected void
onActivityResult(int requestCode,int resultCode, Intent outputIntent) {
   super.onActivityResult(requestCode, resultCode, outputIntent);
   parseResult(this, requestCode, resultCode, outputIntent);
}
public static void parseResult(Activity activity
    , int requestCode, int resultCode , Intent outputIntent)
{
    if (requestCode != 1)  {
     Log.d("Test", "Someone else called this. not us");
     return;
    }
    if (resultCode != Activity.RESULT_OK)  {
      Log.d("Test", "Result code is not ok:" + resultCode);
               return;
    }
    Log.d("Test", "Result code is ok:" + resultCode);
    Uri selectedUri = outputIntent.getData();
    Log.d("Test", "The output uri:" + selectedUri.toString());

    //Proceed to display the note
    outputIntent.setAction(Intent.ACTION_VIEW);
    startActivity(outputIntent);
}
```

The constants RESULT_OK, RESULT_CANCELED, and RESULT_FIRST_USER are all defined in the activity class. The constant RESULT_FIRST_USER is used as a starting number for user-defined activity results. The numerical values of these constants are shown in Listing 2-22:

Listing 2-22. Result Values from Returned Activities

```
RESULT_OK = -1;
RESULT_CANCELED = 0;
RESULT_FIRST_USER = 1;
```

To make the PICK functionality work, the implementing or the target activity that is responding should have code that explicitly addresses the needs of an ACTION_PICK. Let's look at an example of how this is done in the Google sample NotePad application. (See the references section, where you can find this application.) When the item is selected in the list of items, the intent that invoked the target activity is checked to see whether it's an ACTION_PICK intent. If it is, the data URI of the selected note item is set in a new intent and returned through setResult() as shown in Listing 2-23. The calling activity then can investigate the returned intent to see what data it has in it. See the method parseResult() in Listing 2-21.

Listing 2-23. Target Activity Returning a Result through a Data URI

```
@Override
protected void onListItemClick(ListView l, View v, int position, long id) {
    Uri uri = ContentUris.withAppendedId(getIntent().getData(), id);

    String action = getIntent().getAction();
    if (Intent.ACTION_PICK.equals(action) ||
              Intent.ACTION_GET_CONTENT.equals(action))    {
        // The caller is waiting for us to return a note selected by
        // the user.  They have clicked on one, so return it now.
        setResult(RESULT_OK, new Intent().setData(uri));
        finish();
    }
    ...other ways of how this activity may have been invoked
}
```

Exercising the GET_CONTENT Action

ACTION_GET_CONTENT is similar to ACTION_PICK. In the case of ACTION_PICK, you are specifying a data URI that points to a collection of items, like a list of notes from a NotePad-like application. You will expect the intent action to pick one of the notes and return it to the caller. In the case of ACTION_GET_CONTENT, you indicate to Android that you need an item of a particular MIME type. Android searches for activities that can either create one of those items or choose from an existing set of items that satisfy that MIME type.

Using ACTION_GET_CONTENT, you can pick a note from a collection of notes supported by the NotePad application using the code shown in Listing 2-24:

Listing 2-24. Invoking Activities for Creating Content

```
public static void invokeGetContent(Activity activity) {
      Intent pickIntent = new Intent(Intent.ACTION_GET_CONTENT);
      int requestCode = 2;
      pickIntent.setType("vnd.android.cursor.item/vnd.google.note");
      activity.startActivityForResult(pickIntent, requestCode);
}
```

Notice how the intent type is set to the MIME type of a single note. Contrast this with the ACTION_PICK code, where it explicitly indicated a URL that points to a collection of notes (like a web URL that can retrieve a page worth of data).

For an activity to respond to ACTION_GET_CONTENT, the activity has to register an intent filter indicating that the activity can provide an item of that MIME type. Listing 2-25 shows how the SDK's NotePad application accomplishes this:

Listing 2-25. Activity Filter for Get Content

```
<activity android:name="NotesList" android:label="@string/title_notes_list">
......
<intent-filter>
    <action android:name="android.intent.action.GET_CONTENT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note" />
      </intent-filter>
......
</activity>
```

The rest of the code for responding to onActivityResult() is identical to the previous ACTION_PICK example. If there are multiple activities that can return the same MIME type, Android will show you the chooser dialog to let you pick an activity.

Relating Intents and Activities

An intent is used to start not only activities but also other components like a service or a broadcast receiver. These components are covered in later chapters. You can see these components as having certain attributes. One attribute of a component may be the category to which this component belongs. Another attribute may be what type of data this component can view, edit, update, or delete. Another attribute may be what type of actions a component can respond to. If you were to look upon these components as entities in a database, their attributes can be seen as columns. Then an intent can be seen as a where clause that specifies all or some of those characteristics to choose a component like an activity to start. Listing 2-26 is an example of demonstrating how to query for all activities that are categorized as CATEGORY_LAUNCHER.

Listing 2-26. Querying Activities that Match an Intent

```
Intent mainIntent = new Intent(Intent.ACTION_MAIN, null);
mainIntent.addCategory(Intent.CATEGORY_LAUNCHER);
PackageManager pm = getPackageManager();
List<ResolveInfo> list = pm.queryIntentActivities(mainIntent, 0);
```

`PackageManager` is a key class that allows you to discover activities that match certain intents without invoking them. You can cycle through the received activities and invoke them as you see fit, based on the `ResolveInfo` API. Listing 2-27 is an extension to the preceding code that walks through the list of activities and invokes one of the activities if it matches a name. In the code, we have used an arbitrary name to test it:

Listing 2-27. Walking Through a Matched Activity List for an Intent

```
for(ResolveInfo ri: list) {
    //ri.activityInfo.
    Log.d("test",ri.toString());
    String packagename = ri.activityInfo.packageName;
    String classname = ri.activityInfo.name;
    Log.d("test", packagename + ":" + classname);
    if (classname.equals("com.ai.androidbook.resources.TestActivity")) {
        Intent ni = new Intent();
        ni.setClassName(packagename,classname);
        activity.startActivity(ni);
    }
}
```

Understanding Explicit and Implicit Intents

When you specify an explicit activity name (or a component name like a service or a broadcast receiver) in an intent, such an intent is called an explicit intent. When this intent is used to start an activity, that activity is invoked irrespective of what else is there in that intent such as its category or data.

As you have seen, an intent does not have to have an activity specified explicitly to invoke it. An intent can rely on an activity's action attribute, category attribute, or data attribute. These intents that omit the explicit activity or component class are called implicit intents. When you use an implicit intent to invoke an activity it is paramount that the activity must have as one of its categories CATEGORY_DEFAULT. If you expect your activity to be explicitly started by an intent, then you don't need to specify any category at all to that activity. Listing 2-28 shows an example of minimally registering an activity in an Android manifest file so that it can be invoked by an explicit intent.

Listing 2-28. Minimal Activity Definition

```
<activity android:name="com.androidbook.asynctask.TestProgressBarDriverActivity"
      android:label="Test Progress bars"/>
```

If you want to invoke this activity through an implicit intent without specifying its classname, like through an action say, then you need to add the following intent filters, one for the action and one for the needed mandatory category of default, as shown in Listing 2-29.

Listing 2-29. An Activity Definition with Filters

```
<activity android:name="com.androidbook.asynctask.TestProgressBarDriverActivity"
       android:label="Test Progress bars">
      <intent-filter>
           <action android:name="com.androidbook.intent.action.ME" />
           <category android:name="android.intent.category.DEFAULT" />
     </intent-filter>
</activity>
```

Saving State in Android

As you reviewed the calculator app, your next likely need is how to store the data of an Android app. Let's briefly cover the available options. There are five ways to store data in Android: 1) shared preferences, 2) internal files, 3) external files, 4) SQLlite, and 5) network storage in the cloud.

Shared preferences API is a sophisticated API in the Android SDK to save, display, and manipulate preferences for your applications. Although this feature is intended and tailored for preferences it can be used for saving arbitrary state of your application. Shared preferences are internal to the application and device. Android does not make this data available to other applications. A user is not expected to directly manipulate this data by mounting onto a USB port. This data is removed automatically when the application is removed. These shared preferences are covered in detail in Chapter 11.

While shared preferences data is structured key/value pair data and follows a few other semantics imposed, internal files are stand-alone files that you can write to without a predefined structure. We haven't found a compelling advantage of using internal files over shared preferences, or the other way, especially for small- to medium-sized state. So for most apps you can choose one or the other.

Unlike internal files, which are stored on the internal storage of the device, external files are stored on the SD card. These become public files that other apps including the user could see outside the context of your application. The external files can be used to store data that makes sense even outside of your app such as image files or video files. For strictly the internal state of the app, internal files are a better option.

The external files may also be an option if the state is very large running into tens of megabytes. Usually when that happens you don't want to save the state as a monolithic file anyway and opt for more granular storage as a relational database like the SQLlite.

We will give a quick overview and brief code samples in Chapter 25 on how to use preferences, internal files, and external files to store your app state. One of the tricks is to persist java object tree directly using JSON and GSOn while giving consideration to see if this level of granularity is appropriate. If you are not familiar with JSON, it is an object transport and storage format for JavaScript-based objects. It is also generally applicable any object structure as well including java objects and often used that way lately. The GSON is a Google library that converts Java objects to and from JSON strings.

SQLlite is a really good option that is recommended to store the state of an app. The short drawback is your logic to save and read data become verbose and cumbersome. You can probably use O/R mapping libraries to overcome this mismatch between java objects and its relational representation. SQLlite is also often used to store data that needs to be shared by multiple applications through a concept called content providers. This is the central topic of Chapter 25.

Finally, cloud-based network storage is coming into its own. For example a number of MBAAS (Mobile Backend as a Service) platforms such as parse.com support storing the mobile data directly in the cloud for both online and offline usage. This model is going to be increasingly relevant as you start making your app available on multiple devices for the same user or being able to collaborate with other users. This topic is covered in great detail in our companion book *Expert Android* from Apress.

Many time for your apps the GSON option to store app state in an internal file is really the quickest and most practical way to go. Of course you do want to analyze the granularity of the solution and see if this simpler approach won't become a burden on computing power or battery life. If your app gains lot of popularity you may want to use a second release with SQLlite by optimizing storage speed or use cloud storage if that is more appropriate for that release.

Roadmap for Learning Android and the Rest of the Book

Let's quickly review what we have covered so far. In the one-pager application you have seen how the UI is put together, how the business logic is coded in Java, and then how the application is defined to the Android sdk using the Android manifest file. We explained what resources are, how they reference each other, how they are referenced in layout files, and even how to read your input files as resources. We have shown you what intents are, their intricacies, and how to use them to invoke or discover activities. We have covered the activity's life cycle, which is really important to understand Android architecture. We have also given a quick rundown of how you can save the state of your application. This is a pretty good foundation to plan and write simple applications.

We now want to follow up this bird's eye view of Android applications with a roadmap of becoming an expert app developer on the Android platform. This roadmap divides the chapters of this book into the following six key learning tracks:

- Track 1: UI essentials for your Android applications
- Track 2: Saving state
- Track 3: Preparing/taking your application to Google Play
- Track 4: Making your application robust
- Track 5: Bringing finesse to your apps
- Track 6: Integrating with other devices and the cloud

Among these six tracks, the first three are the basic tracks that you must know well to write Android apps that are useful to you and the larger community. Tracks 4, 5, and 6 are there to make your apps better and feature rich in subsequent releases. We will talk about what chapters make up each track and what you are expected to gain from that track.

Track 1: UI Essentials for Your Android Applications

Android has a number of UI controls and layouts out of the box to write very feature rich-applications. Some examples are buttons, various TextViews, EditText controls, checkboxes, RadioButtons, date and time controls, list controls, controls to show analog and digital clocks, controls to show images and videos, controls to pick numbers, etc. We will cover a number of these in **Chapter 3**. In that chapter, we will also cover the essential layouts that are needed to compose the UI from those controls.

Once you are able to use the basic controls to construct your UI, the one control that you absolutely need in your apps is the list control. We did not cover list control as a basic control because it is a bit involved. Also Android has a number of features and approaches to do list-based applications. So we have dedicated a separate chapter for list controls and the data adapters that are necessary to populate those list controls. These aspects are covered in **Chapter 4**.

Once you master the basic controls, basic layouts, and list controls, you will start looking around for more sophisticated layouts like the grid and table layouts. These are covered in **Chapter 5** under "Using Advanced Layouts."

Menus are covered in **Chapter 6**. Android's menu infrastructure includes context menus, pop-up menus, option icons in an action bar, etc.

Your mobile app is not really complete without refining it through styling, much like CSS. **Chapter 7** covers how styles and themes work in Android.

Dialogs are essential in any UI. Dialogs are a bit involved in Android. To understand dialogs in Android you have to first understand the concept of fragments. Architecture of dialogs is only one aspect of fragments. Fragments are now core to the Android UI. **Chapter 8** explains what fragments are and in **Chapter 10** we cover dialogs, building upon Chapter 8.

In mobile apps, you cannot write an app without understanding what happens to your application when the device orientation changes. Programming correctly for orientation change is not trivial in Android. How to program for orientation and other device configuration changes is covered in **Chapter 9**.

For any reasonably useful application you will likely need to know all these UI essentials. So Track 1 is an essential track.

Track 2: Saving State

Once you know how to construct the UI of your application, the next need you will run into is to save the state of your application. Refer to the earlier section on saving state to see what options are available and in which chapters those options are covered. Track 2 is also an essential track as you should know how to save state.

Track 3: Preparing/Taking Your Application to the Market

By completing Tracks 1 and 2 you can build a pretty reasonable application that you can deploy to the marketplace. **Chapter 30** shows you how you can take your app to the Google Play store.

Track 4: Making Your Application Robust

Track 4 is an advanced track getting into the internals of Android. You will need to go through the chapters in this track to solidify your understanding of how Android works. We start this track with **Chapter 12** on compatibility library. This chapter teaches how to make your app run well on older releases while using features that are available only on the newer platforms.

Android allows you to run code in your application even though you are not actively using the application in the foreground. It could be the music you are playing in the background, or it could be backing up your images to a cloud, etc. This type of code is called a Service in Android. Working with services is covered in **Chapter 13**. These services can be triggered by a direct user action or through alarms or broadcast events. Alarm manager is covered in **Chapter 17**.

When you use intents to invoke components such as activities or services you are targeting a single component. Android also supports a publish-and-subscribe protocol where an intent can be used to invoke multiple components that register for it at the same time. These components are called receivers or broadcast receivers. A broadcast receiver is a piece of code in your application that is executed in response to a broadcasted event even if your application had not been started or was just dormant at the time of the event. How to work with broadcast receivers is covered in **Chapter 16**.

As you start using more and more features of Android such as services, broadcast receivers, and content providers you will need to understand how Android uses a single main thread to run the code in these components. This threading model is covered in detail in **Chapter 14**. Knowing this will help you write code that is robust. In this track, you will also learn about the very useful `AsyncTask`, which is used to simplify offloading work from the main thread. This API is often used from UI to read messages from the web or check for e-mails, etc. `AsyncTask` is covered in **Chapter 15**.

Track 5: Bringing Finesse to Your Apps

To make your apps look appealing, one of the first things you can do is to add a little or a lot of animation. This is covered in **Chapter 18**. Touch-based interfaces are now the norm. Manipulating your environment with drag and drop is more natural. You want to employ sensors to write apps that integrate with the external world better. These touch screens, drag and drop, and sensors are respectively covered in **Chapters 22, 23, and 24**.

Home screen widgets are a wonderful way to extract pieces of your app and make it available on any home screen of your choosing. This personalization feature, when used innovatively with value in mind, makes the interaction with the device simple and joyful. Widgets are covered in **Chapter 21**.

Map- and location-based apps are made for mobile devices. This topic is covered in **Chapter 19**.

You can very easily integrate audio and video into your apps on Android. This API is covered in **Chapter 20**.

Track 6: Integrating with Other Devices and the Cloud

You can use Google cloud messaging to reach out to the users of your mobile applications. Google cloud messaging is covered in **Chapter 29**. With NFC and Bluetooth capabilities in Android you can start interacting with your physical environment in your apps. We hope to post some material on these topics to the online companion for the book.

Final Track: Getting a Helping Hand from Expert Android

Now, we are going to talk about a few topics that are not covered in this book. You may want to consider these topics, should you find them relevant to your needs. Most of these are based on our research for the *Expert Android* book that we published in early 2014 through Apress.

Android has a public API to write custom components that can work and behave differently than what come out of the box. You can write custom views where you can control what to draw and how to draw, which can then coexist with other controls that are out of the box like a button or a text control. You can also combine multiple existing controls into a compound control that can then behave like an independent control. You can also design new layouts that suit your display needs. There are a lot of tricky things to create these custom components well. You have to understand the core Android view architecture. This material is covered over three chapters and 100 pages in the *Expert Android* book from Apress.

If your apps are form based, you will need to write a lot of code to validate the form input. You really need a framework to handle this. *Expert Android* has a chapter on creating a small form processing framework that is really useful and will reduce errors and the amount of code you need to write.

MBAAS, Mobile Backend as a Service, is a needed technology for mobile apps and is now pretty widely available. The facilities an MBAAS offers are user logins, social logins, user management, savind data on behalf of users in the cloud, communication with the users, collaboration between the users themselves, etc. In *Expert Android* we have multiple chapters dedicated to an MBAAS platform called Parse.

OpenGL has come a long way on Android with now-substantial support for the new generation of programmable GPUs. Android has been supporting ES 2.0 for some time now. In *Expert Android* we have over 100 pages of coverage on OpenGL. We start at the beginning and explain all the concepts without needing to refer to external books, although we do give an extensive bibliography on OpenGL. We cover ES 2.0 really well and provide guidance to combine OpenGL and regular views to pave the way for 3D components.

Federated search protocol of Android is powerful as you can use it in quite a few imaginative ways. *Expert Android* fully explores its fundamentals and also some alternate ways of using it optimally.

Android provides an increasingly large set of features for debugging. These topics are covered in *Expert Android*. A cell phone is ultimately a talking device, although it is used less and less often for that. We have a chapter on utilizing the telephony API in *Expert Android* as well.

As We Leave You Now with the Rest of the Book

Finally, you may be wondering why you should even become a mobile developer. We can cite two strong arguments, one of which never existed before. The familiar one is to be part of an IT organization for their mobile programming efforts. The IT opportunities are on the rise but not fully realized yet unlike what happened with the Web programming paradigm when it came into being. We expect this need, however, to be a gradually increasing demand.

On the other hand, the immediate and exciting opportunity is for you to become an independent app publisher. The availability of a sales channel for the apps that you write is a unique one in the software industry. Not every one of us is going to be a rising star in an IT organization. The independent developer path gives an avenue for you to grow at your own pace and in a direction that satisfies you. Luck and patience might even make you rich. At least you can add value to the society while meeting your needs.

Should you decide to venture into the Android mobile programming space, you want to be prepared with the right hardware that makes this experience bearable. If you are buying a Windows laptop see if you can get one with at least 8G of memory, solid-state hard drive, and a reasonably fast processor. Expect to spend about $1,000 to $1,500. If you are buying a Mac laptop, a similar configuration may cost you about $2,500. A good fast configuration is important for Android development. If you are a seasoned Java programmer, given this investment, and this book in hand, if you follow the tracks laid out here you can become a competent mobile Android app developer in about six months.

References

Here are additional resources for the topics discussed in this chapter.

- http://androidbook.com/free-android-chapters: You can use this URL to download detailed chapters on resources and intents (made available free from previous editions due to space limitations).
- http://androidbook.com/working-with-avds: You will find at this URL notes on installing Android, working with AVDs, signing APK files, and more to get you started with Android.
- http://androidbook.com/item/3574: This URL shows how to run an Android application on a device from the eclipse ADT. This link also shows you how to hook up your device through a USB port to your development computer.
- http://androidbook.com/item/4629: This URL talks about key callback functions on an activity. Monitoring the activity callbacks is a good way to get a handle on the activity life cycle. You can copy the code from here to create a base activity that can monitor and log these callbacks for you.

- http://androidbook.com/item/4440: This URL talks about how you can use GSON and JSON for persistence needs of your application. This article suggests an easy way to persist data on the device for your apps.
- *Expert Android* from Apress talks about passing objects through Android bundles as parcelables in depth.
- http://developer.android.com/guide/topics/resources/index.html: Android SDK roadmap to the documentation on resources.
- http://developer.android.com/guide/topics/resources/available-resources.html: Android documentation of various types of resources available.
- http://developer.android.com/guide/topics/resources/providing-resources.html#AlternativeResources: A list of various configuration qualifiers provided by the latest Android SDK.
- http://developer.android.com/guide/practices/screens_support.html: Guidelines on how to design Android applications for multiple screen sizes.
- http://developer.android.com/reference/android/content/res/Resources.html: Various Java methods available to read resources.
- http://developer.android.com/reference/android/R.html: Resources as defined to the core Android platform.
- http://androidbook.com/item/3542: Our research on plurals, string arrays, resource qualifiers, and alternate resources, as well as links to other references.
- http://androidbook.com/item/4236: Using drawable resources to control backgrounds.
- http://developer.android.com/training/notepad/index.html: A beginner's guide, yet a comprehensive introduction to Android applications through a NotePad example.
- http://developer.android.com/reference/android/content/Intent.html: Overview of intents, including well-known actions, extras, and so on.
- http://developer.android.com/guide/appendix/g-app-intents.html: Lists the intents for a set of Google applications. Here, you will see here how to invoke Browser, Map, Dialer, and Google Street View.
- http://developer.android.com/reference/android/content/IntentFilter.html: Talks about intent filters and is useful when you are registering intent filters for activities and other components in the manifest file.
- http://developer.android.com/guide/topics/intents/intents-filters.html: Goes into the resolution rules of intent filters.

- http://developer.android.com/training/notepad/index.html: URL where you can download the sample code for a NotePad application. This is a good sample application that features a number of Android APIs. A good place to go after the calculator application.
- http://developer.android.com/samples/index.html: This is the primary link to browse through the various samples presented for the Android SDK by Google.
- http://developer.android.com/training/index.html: This is the primary learning site from Google that presents a series of lessons to learn Android.
- https://code.google.com/p/openintents/: A web effort to make various Android applications work together.
- http://androidbook.com/item/4623: A roadmap for learning Android. Although some of these points are covered here, see this URL for the latest guidance on learning and maximizing Android.
- http://androidbook.com/item/4764: This is a knowledge folder containing a series of articles and tidbits on programming with Android basic UI.
- http://www.androidbook.com/proandroid5/projects. Look here for a list of downloadable projects related to this book. For this chapter, look for a ZIP file called ProAndroid5_Ch02_Calculator.zip.

Summary

This chapter laid out everything you need to understand to create mobile applications with the Android SDK. You have seen how UI is constructed. You know what activities are. You know the intricacies of the activity life cycle. You understood resources and intents. You know how to save state. Finally, you got to see the breadth of the Android SDK by reading the learning tracks that summarized the rest of the book. We hope these first two chapters gave you a head start for your development efforts with the Android SDK.