UNDERSTANDING THE COMPONENTS OF A SCREEN

As explained in Chapter 3, the basic unit of an Android application is an *activity*, which displays the UI of your application. The activity may contain widgets such as buttons, labels, textboxes, and so on. Typically, you define your UI using an XML file (for example, the `activity_main.xml` file located in the `res/layout` folder of your project), which looks similar to what is shown in here.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.design.widget.CoordinatorLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:context="com.jfdimarzio.helloworld.MainActivity">

    <android.support.design.widget.AppBarLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:theme="@style/AppTheme.AppBarOverlay">

        <android.support.v7.widget.Toolbar
            android:id="@+id/toolbar"
            android:layout_width="match_parent"
            android:layout_height="?attr/actionBarSize"
            android:background="?attr/colorPrimary"
            app:popupTheme="@style/AppTheme.PopupOverlay" />

    </android.support.design.widget.AppBarLayout>

    <include layout="@layout/content_main" />

    <android.support.design.widget.FloatingActionButton
        android:id="@+id/fab"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|end"
        android:layout_margin="@dimen/fab_margin"
        android:src="@android:drawable/ic_dialog_email" />

</android.support.design.widget.CoordinatorLayout>
```

During runtime, you load the XML UI in the `onCreate()` method handler in your `Activity` class, using the `setContentView()` method of the `Activity` class:

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

During compilation, each element in the XML file is compiled into its equivalent Android GUI (Graphical User Interface) class, with attributes represented by methods. The Android system then creates the activity's UI when the activity is loaded.

> **NOTE** *Although it is always easier to build your UI using an XML file, sometimes you need to build your UI dynamically during runtime (for example, when writing games). Hence, it is also possible to create your UI entirely using code. Later in this chapter is an example showing how to create a UI using code.*

Views and ViewGroups

An activity contains *views* and *ViewGroups*. A view is a widget that has an appearance on screen. Examples of views are buttons, labels, and text boxes. A view derives from the base class `android.view.View`.

> **NOTE** *Chapters 5 and 6 discuss the various common views in Android.*

One or more views can be grouped into a ViewGroup. A ViewGroup (which is itself a special type of view) provides the layout in which you can order the appearance and sequence of views. Examples of ViewGroups include `RadioGroup` and `ScrollView`. A ViewGroup derives from the base class `android.view.ViewGroup`.

Another type of ViewGroup is a `Layout`. A `Layout` is another container that derives from `android.view.ViewGroup` and is used as a container for other views. However, whereas the purpose of a ViewGroup is to group views logically—such as a group of buttons with a similar purpose—a `Layout` is used to group and arrange views visually on the screen. The `Layouts` available to you in Android are as follows:

- `FrameLayout`
- `LinearLayout(Horizontal)`
- `LinearLayout(Vertical)`
- `TableLayout`
- `TableRow`
- `GridLayout`
- `RelativeLayout`

The following sections describe each of these `Layouts` in more detail.

> ***NOTE*** *One of the great things about Android development, is that you can mix and match multiple Layouts to create a truly unique control layout for your application.*

FrameLayout

The `FrameLayout` is the most basic of the Android layouts. `FrameLayouts` are built to hold one view. As with all things related to development, there is no hard rule that `FrameLayouts` can't be used to hold multiple views. However, there is a reason why `FrameLayouts` were built the way they were.

Given that there are myriad screen sizes and resolutions, you have little control over the specifications of the devices that install your application. Therefore, when your application is resized and reformatted to fit any number of different devices you want to make sure it still looks as close to your initial design as possible.

The `FrameLayout` is used to help you control the stacking of single views as the screen is resized. In the following Try It Out, you add a `TextView` to a `FrameLayout` in your `HelloWorld` application.

TRY IT OUT Place a TextView Within a FrameLayout

Using the HelloWorld project you created in Chapters 1 and 2, create a new layout resource file for your application, add a `FrameLayout` to the new layout resource, and finally place a `TextView` within the `FrameLayout`.

1. Open the HelloWorld project in Android Studio.
2. Right-click the `res/layout` folder and add a new layout resource file. Name the file **framelayout_example.xml**.
4. Using the design panel, drag the `FrameLayout` and place it anywhere on the device screen.
5. Using the design panel, drag a `Plain TextView` and place it in the `FrameLayout`.
6. Type some text into the `Plain TextView`.

How It Works

The `FrameLayout` displays the `Plain TextView` as a free-floating control. The original purpose of the `FrameLayout` was to hold a single element. However, it can be used to hold more.

LinearLayout (Horizontal) and LinearLayout (Vertical)

The `LinearLayout` arranges views in a single column or a single row. Child views can be arranged either horizontally or vertically, which explains the need for two different layouts—one for horizontal rows of views and one for vertical columns of views.

> ***NOTE*** `LinearLayout(Horizontal)` *and* `LinearLayout(Vertical)` *are actually the same layout with one property changed. Later in this section, you find out how the* `android:orientation` *property of the* `LinearLayout` *controls if the application has a horizontal or vertical flow.*

To see how `LinearLayout` works, consider the following elements typically contained in the `activity_main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />
</LinearLayout>
```

In the `activity_main.xml` file, observe that the root element is `<LinearLayout>` and it has a `<TextView>` element contained within it. The `<LinearLayout>` element controls the order in which the views contained within it appear.

Each view and ViewGroup has a set of common attributes, some of which are described in Table 4-1.

TABLE 4-1: Common Attributes Used in Views and ViewGroups

ATTRIBUTE	DESCRIPTION
`layout_width`	Specifies the width of the view or ViewGroup
`layout_height`	Specifies the height of the view or ViewGroup
`layout_marginTop`	Specifies extra space on the top side of the view or ViewGroup
`layout_marginBottom`	Specifies extra space on the bottom side of the view or ViewGroup
`layout_marginLeft`	Specifies extra space on the left side of the view or ViewGroup
`layout_marginRight`	Specifies extra space on the right side of the view or ViewGroup
`layout_gravity`	Specifies how child views are positioned
`layout_weight`	Specifies how much of the extra space in the layout should be allocated to the view
`layout_x`	Specifies the x-coordinate of the view or ViewGroup
`layout_y`	Specifies the y-coordinate of the view or ViewGroup

> ***NOTE*** *Some of these attributes are applicable only when a view is in a specific ViewGroup. For example, the* `layout_weight` *and* `layout_gravity` *attributes are applicable only when a view is in either a* `LinearLayout` *or a* `TableLayout`*.*

For example, the width of the `<TextView>` element fills the entire width of its parent (which is the screen in this case) using the `fill_parent` constant. Its height is indicated by the `wrap_content` constant, which means that its height is the height of its content (in this case, the text contained within it). If you don't want the `<TextView>` view to occupy the entire row, you can set its `layout_width` attribute to `wrap_content`, like this:

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

The preceding code sets the width of the view to be equal to the width of the text contained within it. Consider the layout in the next code snippet, which shows two views with their width explicitly stated as a measurement, and their heights set to the height of their contents.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<TextView
    android:layout_width="100dp"
    android:layout_height="wrap_content"
    android:text="@string/hello" />
<Button
    android:layout_width="160dp"
    android:layout_height="wrap_content"
    android:text="Button"
    android:onClick="onClick" />
</LinearLayout>
```

UNITS OF MEASUREMENT

When specifying the size of an element on an Android UI, you should be aware of the following units of measurement:

- `dp`—Density-independent pixel. 1 `dp` is equivalent to one pixel on a 160 dpi screen. This is the recommended unit of measurement when you're specifying the dimension of views in your layout. The 160 dpi screen is the baseline density assumed by Android. You can specify either `dp` or `dip` when referring to a density-independent pixel.
- `sp`—Scale-independent pixel. This is similar to `dp` and is recommended for specifying font sizes.

- `pt`—Point. A point is defined to be 1/72 of an inch, based on the physical screen size.
- `px`—Pixel. Corresponds to actual pixels on the screen. Using this unit is not recommended, as your UI might not render correctly on devices with a different screen resolution.

Here, you set the width of both the `TextView` and `Button` views to an absolute value. In this case, the width for the `TextView` is set to 100 density-independent pixels wide, and the `Button` to 160 density-independent pixels wide. Before you see how the views look on different screens with different pixel densities, it is important to understand how Android recognizes screens of varying sizes and densities.

Figure 4-1 shows the screen of the Nexus 5 (from the emulator). It has a 5-inch screen (diagonally), with a screen width of 2.72 inches. Its resolution is 1080 (width) × 1920 (height) pixels. The pixel density of a screen varies according to screen size and resolution.

FIGURE 4-1

To test how the views defined in the XML file look when displayed on screens of different densities, create two Android Virtual Devices (AVDs) with different screen resolutions and abstracted LCD densities. Figure 4-2 shows an AVD with 1080 × 1920 resolution and LCD density of 480.

Figure 4-3 shows another AVD with 768 × 1280 resolution and LCD density of 320.

Using the `dp` unit ensures that your views are always displayed in the right proportion regardless of the screen density. Android automatically scales the size of the view depending on the density of the screen.

FIGURE 4-2

FIGURE 4-3

HOW TO CONVERT DP TO PX

The formula for converting `dp` to `px` (pixels) is as follows:

Actual pixels = `dp` * (dpi / 160), where dpi is either 120, 160, 240, or 320.

Therefore, in the case of the `Button` on a 235 dpi screen, its actual width is 160 * (240/160) = 240 `px`. When run on the 180 dpi emulator (regarded as a 160 dpi device), its actual pixel width is now 160 * (160/160) = 160 `px`. In this case, one `dp` is equivalent to one `px`.

To prove that this is indeed correct, you can use the `getWidth()` method of a `View` object to get its width in pixels:

```
public void onClick(View view) {
    Toast.makeText(this,
        String.valueOf(view.getWidth()),
        Toast.LENGTH_LONG).show();
}
```

What if instead of using `dp` you now specify the size using pixels (`px`)?

```
<TextView
    android:layout_width="100px"
    android:layout_height="wrap_content"
    android:text="@string/hello" />
<Button
    android:layout_width="160px"
    android:layout_height="wrap_content"
    android:text="Click Me"
    android:onClick="onClick"/>
```

Figure 4-4 shows how the `Label` and `Button` appear on a 480 dpi screen. Figure 4-5 shows the same views on a 320 dpi screen. In this case, Android does not perform any conversion because all the sizes are specified in pixels. If you use pixels for view sizes, the views appear smaller on a device with a high dpi screen than a screen with a lower dpi (assuming screen sizes are the same).

FIGURE 4-4

FIGURE 4-5

The preceding example also specifies that the orientation of the layout is vertical:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
```

The default orientation layout is horizontal, so if you omit the `android:orientation` attribute, the views appear as shown in Figure 4-6.

FIGURE 4-6

In `LinearLayout`, you can apply the `layout_weight` and `layout_gravity` attributes to views contained within it, as the modifications to `activity_main.xml` in the following code snippet show:

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

```
<Button
android:layout_width="160dp"
    android:layout_height="0dp"
    android:text="Button"
    android:layout_gravity="left"
    android:layout_weight="1" />
<Button
android:layout_width="160dp"
    android:layout_height="0dp"
    android:text="Button"
    android:layout_gravity="center"
    android:layout_weight="2" />
<Button
android:layout_width="160dp"
    android:layout_height="0dp"
    android:text="Button"
    android:layout_gravity="right"
    android:layout_weight="3" />
    </LinearLayout>
```

Figure 4-7 shows the positioning of the views as well as their heights. The `layout_gravity` attribute indicates the positions the views should gravitate toward, whereas the `layout_weight` attribute specifies the distribution of available space. In the preceding example, the three buttons occupy about 16.6 percent (1/(1+2+3) * 100), 33.3 percent (2/(1+2+3) * 100), and 50 percent (3/(1+2+3) * 100) of the available height, respectively.

> ***NOTE*** *The height of each button is set to 0* `dp` *because the layout orientation is vertical.*

If you change the orientation of the `LinearLayout` to horizontal (as shown in the following code snippet), you need to change the width of each view to 0 `dp`. The views display as shown in Figure 4-8:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="horizontal" >
<Button
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Button"
    android:layout_gravity="left"
    android:layout_weight="1" />
<Button
    android:layout_width="0dp"
    android:layout_height="wrap_content"
```

```
    android:text="Button"
    android:layout_gravity="center_horizontal"
    android:layout_weight="2" />
<Button
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:text="Button"
    android:layout_gravity="right"
    android:layout_weight="3" />
</LinearLayout>
```

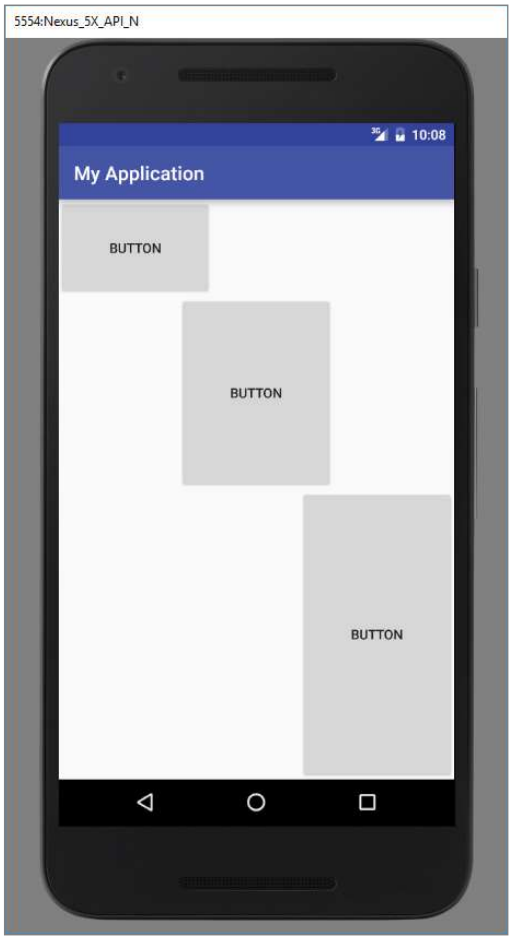

FIGURE 4-7

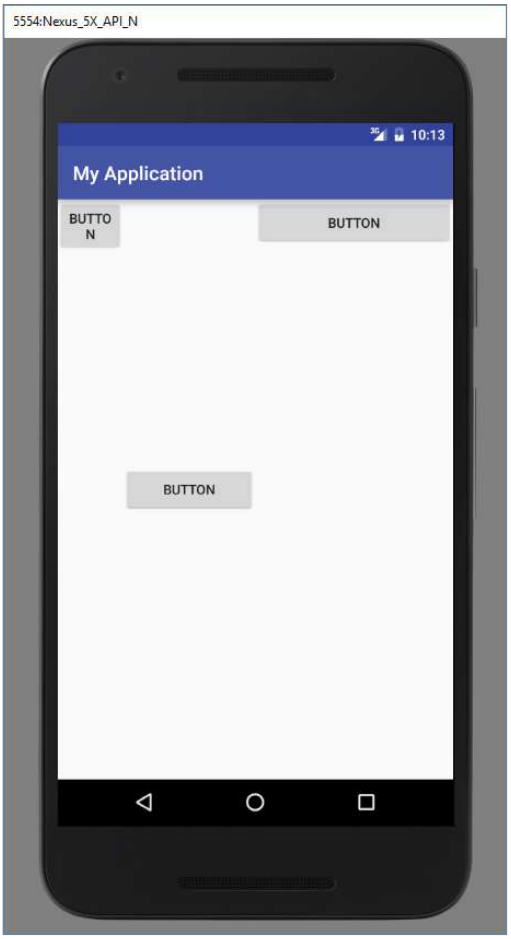

FIGURE 4-8

In the following Try It Out, you combine multiple `LinearLayouts` to create an L-shaped configuration of views.

TRY IT OUT Combine Two LinearLayouts with Different Orientations

Use two `LinearLayouts`, one with a vertical orientation and one with a horizontal orientation to place three `TextViews` and three `Buttons` in the configuration shown in Figure 4-9.

FIGURE 4-9

1. Create a new layout resource file named `linearlayouts_example.xml`. By default, this new file will be created with a `LinearLayout` (Vertical).
2. Place three stacked `TextViews` in a single column within the `LinearLayout`.
3. Place a `LinearLayout` (Horizontal) under the third `TextView`.
4. Place three buttons in a row within the `LinearLayout` (Horizontal).

How It Works

Layouts can be combined in many different configurations to produce almost any look imaginable. In this example, you use a vertical and a horizontal `LinearLayout` to create an L-shaped layout. Your finished `.xml` file should look like this:

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

```
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Text 1"
        android:id="@+id/textView" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Text 2"
        android:id="@+id/textView2" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Text 3"
        android:id="@+id/textView3" />

    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Button 1"
            android:id="@+id/button" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Button 2"
            android:id="@+id/button2" />

        <Button
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Button 3"
            android:id="@+id/button3" />
    </LinearLayout>
</LinearLayout>
```

TableLayout

The `TableLayout` Layout groups views into rows and columns. You use the `<TableRow>` element to designate a row in the table. Each row can contain one or more views. Each view you place

within a row forms a cell. The width of each column is determined by the largest width of each cell in that column.

Consider the content of `activity_main.xml` shown here:

```
<TableLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent" >
    <TableRow>
        <TextView
            android:text="User Name:"
            android:width ="120dp"
            />
        <EditText
            android:id="@+id/txtUserName"
            android:width="200dp" />
    </TableRow>
    <TableRow>
        <TextView
            android:text="Password:"
            />
        <EditText
            android:id="@+id/txtPassword"
            android:inputType="textPassword"
            />
    </TableRow>
    <TableRow>
        <TextView />
        <CheckBox android:id="@+id/chkRememberPassword"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Remember Password"
            />
    </TableRow>
    <TableRow>
        <Button
            android:id="@+id/buttonSignIn"
            android:text="Log In" />
    </TableRow>
</TableLayout>
```

Figure 4-10 shows how the preceding code appears when rendered on the Android emulator.

> ***NOTE*** *In the preceding example, there are two columns and four rows in the* `TableLayout`*. The cell directly under the Password* `TextView` *is populated with a* `<TextView/>` *empty element. If you don't do this, the Remember Password check box will appear under the Password* `TextView`*, as shown in Figure 4-11.*

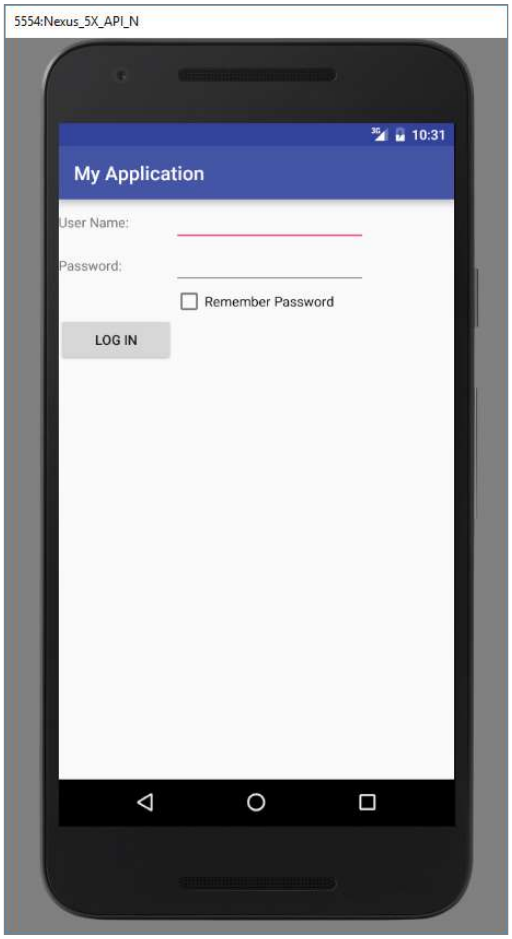

FIGURE 4-10

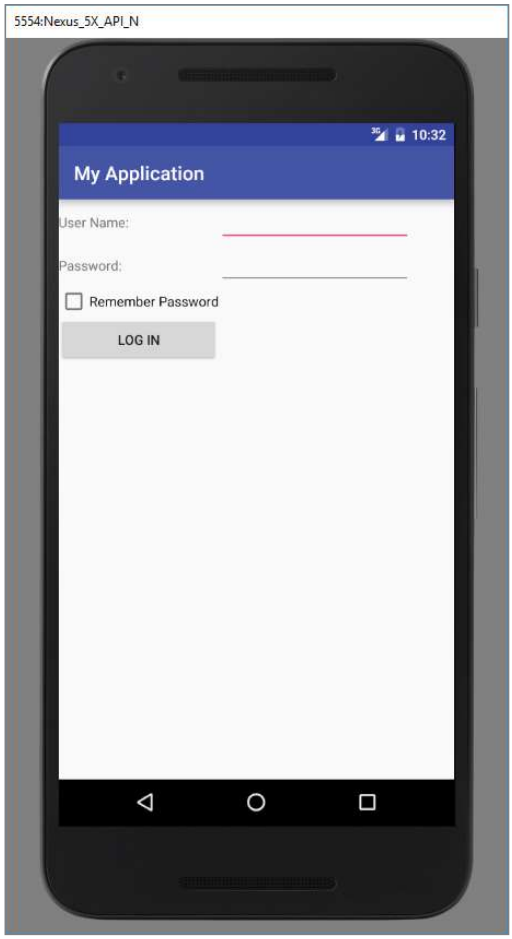

FIGURE 4-11

RelativeLayout

The `RelativeLayout` layout enables you to specify how child views are positioned relative to each other. Consider the following `activity_main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    android:id="@+id/RLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android" >

    <TextView
        android:id="@+id/lblComments"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Comments"
        android:layout_alignParentTop="true"
        android:layout_alignParentStart="true" />
```

```
    <EditText
        android:id="@+id/txtComments"
        android:layout_width="fill_parent"
        android:layout_height="170dp"
        android:textSize="18sp"
        android:layout_alignStart="@+id/lblComments"
        android:layout_below="@+id/lblComments"
        android:layout_centerHorizontal="true" />

    <Button
        android:id="@+id/btnSave"
        android:layout_width="125dp"
        android:layout_height="wrap_content"
        android:text="Save"
        android:layout_below="@+id/txtComments"
        android:layout_alignEnd="@+id/txtComments" />

    <Button
        android:id="@+id/btnCancel"
        android:layout_width="124dp"
        android:layout_height="wrap_content"
        android:text="Cancel"
        android:layout_below="@+id/txtComments"
        android:layout_alignStart="@+id/txtComments" />
</RelativeLayout>
```

Notice that each view embedded within the `RelativeLayout` has attributes that enable it to align with another view. These attributes are as follows:

- `layout_alignParentTop`
- `layout_alignParentStart`
- `layout_alignStart`
- `layout_alignEnd`
- `layout_below`
- `layout_centerHorizontal`

The value for each of these attributes is the ID for the view that you are referencing. The preceding XML UI creates the screen shown in Figure 4-12.

FrameLayout

The `FrameLayout` layout is a placeholder on screen that you can use to display a single view. Views that you add to a `FrameLayout` are always anchored to the top left of the layout. Consider the following content in `main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    android:id="@+id/RLayout"
    android:layout_width="fill_parent"
```

```
        android:layout_height="fill_parent"
        xmlns:android="http://schemas.android.com/apk/res/android" >
        <TextView
            android:id="@+id/lblComments"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Hello, Android!"
            android:layout_alignParentTop="true"
            android:layout_alignParentStart="true" />
        <FrameLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignStart="@+id/lblComments"
            android:layout_below="@+id/lblComments"
            android:layout_centerHorizontal="true" >
            <ImageView
                android:src="@mipmap/butterfly"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content" />
        </FrameLayout>
    </RelativeLayout>
```

FIGURE 4-12

Here, you have a `FrameLayout` within a `RelativeLayout`. Within the `FrameLayout`, you embed an `ImageView`. The UI is shown in Figure 4-13.

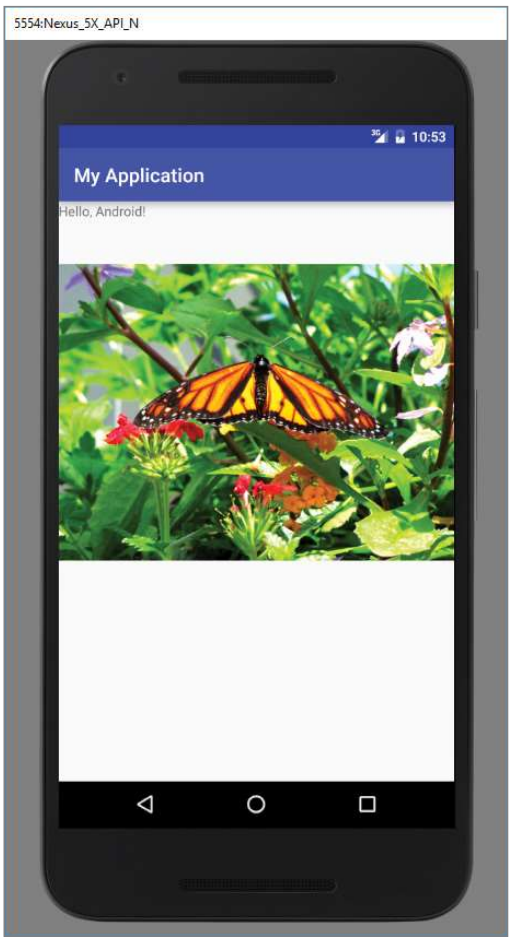

FIGURE 4-13

> ***NOTE*** *This example assumes that the* `res/mipmap-hdpi` *folder has an image named* `butterfly.png`.

If you add another view (such as a `Button` view) within the `FrameLayout`, the view overlaps the previous view (see Figure 4-14):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    android:id="@+id/RLayout"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android" >
    <TextView
```

```
        android:id="@+id/lblComments"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, Android!"
        android:layout_alignParentTop="true"
        android:layout_alignParentStart="true" />
    <FrameLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignStart="@+id/lblComments"
        android:layout_below="@+id/lblComments"
        android:layout_centerHorizontal="true" >
        <ImageView
            android:src="@mipmap/butterfly"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
        <Button
            android:layout_width="124dp"
            android:layout_height="wrap_content"
            android:text="Print Picture" />
    </FrameLayout>
```

FIGURE 4-14

> ***NOTE*** *You can add multiple views to a* `FrameLayout`*, but each is stacked on top of the previous one. This is when you want to animate a series of images, with only one visible at a time.*

ScrollView

A `ScrollView` is a special type of `FrameLayout` in that it enables users to scroll through a list of views that occupy more space than the physical display. The `ScrollView` can contain only one child view or ViewGroup, which normally is a `LinearLayout`.

> ***NOTE*** *Do not use a* `ListView` *(discussed in Chapter 5) with the* `ScrollView`*. The* `ListView` *is designed for showing a list of related information and is optimized for dealing with large lists.*

The following `main.xml` content shows a `ScrollView` containing a `LinearLayout`, which in turn contains some `Button` and `EditText` views:

```
<ScrollView
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    xmlns:android="http://schemas.android.com/apk/res/android" >

    <LinearLayout
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical" >
        <Button
            android:id="@+id/button1"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Button 1" />
        <Button
            android:id="@+id/button2"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Button 2" />
        <Button
            android:id="@+id/button3"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Button 3" />
        <EditText
            android:id="@+id/txt"
            android:layout_width="fill_parent"
            android:layout_height="600dp" />
```

```
        <Button
            android:id="@+id/button4"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Button 4" />
        <Button
            android:id="@+id/button5"
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:text="Button 5" />
    </LinearLayout>
</ScrollView>
```

If you load the preceding code on the Android emulator, you see something like what's shown in Figure 4-15.

FIGURE 4-15

Because the `EditText` automatically gets the focus, it fills up the entire activity (as the height was set to 600dp). To prevent it from getting the focus, add the following two bolded attributes to the `<LinearLayout>` element:

```
        <LinearLayout
            android:layout_width="fill_parent"
```

```
android:layout_height="wrap_content"
android:orientation="vertical"
 android:focusable="true"
 android:focusableInTouchMode="true" >
```

Now you are able to view the buttons and scroll through the list of views (see Figure 4-16).

FIGURE 4-16

Sometimes you might want `EditText` to automatically get the focus, but you do not want the soft input panel (keyboard) to appear automatically (which happens on a real device). To prevent the keyboard from appearing, add the following bolded attribute to the `<activity>` element in the `AndroidManifest.xml` file:

```
<activity
    android:label="@string/app_name"
    android:name=".LayoutsActivity"
    android:windowSoftInputMode="stateHidden" >
    <intent-filter >
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

ADAPTING TO DISPLAY ORIENTATION

One of the key features of modern smartphones is their ability to switch screen orientation, and Android is no exception. Android supports two screen orientations: *portrait* and *landscape*. By default, when you change the display orientation of your Android device, the current activity automatically redraws its content in the new orientation. This is because the `onCreate()` method of the activity is fired whenever there is a change in display orientation.

> ***NOTE*** *When you change the orientation of your Android device, your current activity is actually destroyed and then re-created.*

However, when the views are redrawn, they may be drawn in their original locations (depending on the layout selected). Figure 4-17 shows the previous example displayed in landscape mode.

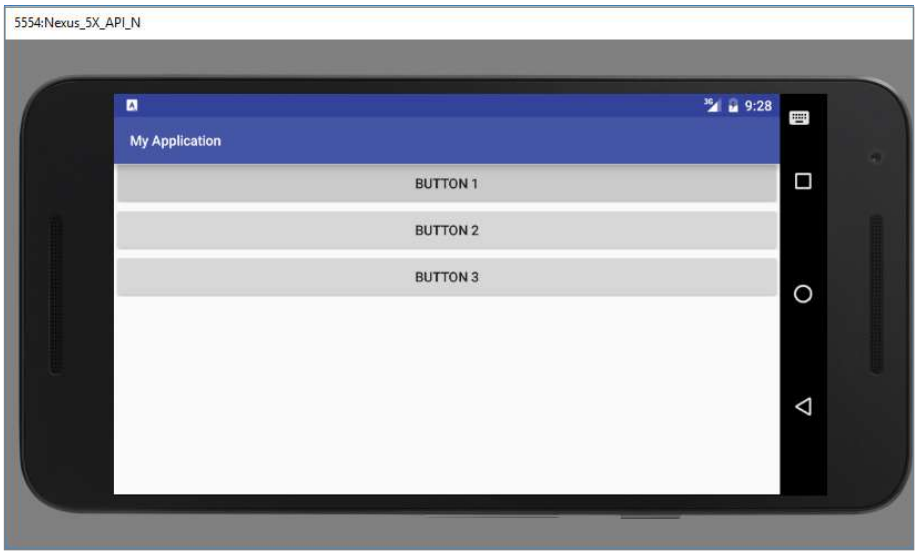

FIGURE 4-17

In general, you can employ two techniques to handle changes in screen orientation:

- **Anchoring**—The easiest way is to "anchor" your views to the four edges of the screen. When the screen orientation changes, the views can anchor neatly to the edges.
- **Resizing and repositioning**—Whereas anchoring and centralizing are simple techniques to ensure that views can handle changes in screen orientation, the ultimate technique is resizing each and every view according to the current screen orientation.

5

Designing Your User Interface with Views

WHAT YOU WILL LEARN IN THIS CHAPTER

- How to use the basic views in Android to design your user interface
- How to use the picker views to display lists of items
- How to use the list views to display lists of items
- How to use specialized fragments

> ***CODE DOWNLOAD*** *The wrox.com code downloads for this chapter are found at* `www.wrox.com/go/beginningandroidprog` *on the Download Code tab. The code is in the chapter 05 download and individually named according to the names throughout the chapter.*

In the previous chapter, you learned about the various layouts that you can use to position your views in an activity. You also learned about the techniques you can use to adapt to different screen resolutions and sizes. This chapter gives you a look at the various views that you can use to design the user interface (UI) for your applications.

In particular, the chapter covers the following ViewGroups:

- **Basic views**—Commonly used views, such as the `TextView`, `EditText`, and `Button` views
- **Picker views**—Views that enable users to select from a list, such as the `TimePicker` and `DatePicker` views

- **List views**—Views that display a long list of items, such as the `ListView` and the `SpinnerView` views
- **Specialized fragments**—Special fragments that perform specific functions

Subsequent chapters cover the other views not covered in this chapter, such as the analog and digital clock views and other views for displaying graphics, and so on.

USING BASIC VIEWS

To get started, this section explores some of the basic views that you can use to design the UI of your Android applications:

- `TextView`
- `EditText`
- `Button`
- `ImageButton`
- `CheckBox`
- `ToggleButton`
- `RadioButton`
- `RadioGroup`

These basic views enable you to display text information, as well as perform some basic selection. The following sections explore all these views in more detail.

TextView View

When you create a new Android project, Android Studio always creates the `activity_main.xml` file (located in the `res/layout` folder), which contains a `<TextView>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```

You use the `TextView` view to display text to the user. This is the most basic view and one that you will frequently use when you develop Android applications. If you need to allow users to edit the

text displayed, you should use the subclass of `TextView`—`EditText`—which is discussed in the next section.

> ***NOTE*** *In some other platforms,* `TextView` *is commonly known as the* Label View. *Its sole purpose is to display text on the screen, and it is read only—the user cannot enter any text into a label.*

Button, ImageButton, EditText, CheckBox, ToggleButton, RadioButton, and RadioGroup Views

Besides the `TextView` view, which you will likely use the most often, there are some other basic views that you will find yourself frequently using:

- `Button`—Represents a push-button widget.
- `ImageButton`—Similar to the `Button` view, except that it also displays an image.
- `EditText`—A subclass of the `TextView` view, which allows users to edit its text content.
- `CheckBox`—A special type of button that has two states: checked or unchecked.
- `RadioGroup` and `RadioButton`—The `RadioButton` has two states: either checked or unchecked. A `RadioGroup` is used to group one or more `RadioButton` views, thereby allowing only one `RadioButton` to be checked within the `RadioGroup`.
- `ToggleButton`—Displays checked/unchecked states using a light indicator.

The following Try It Out provides details about how these views work.

TRY IT OUT Using the Basic Views (BasicViews1.zip)

1. Using Android Studio, create an Android project and name it **BasicViews1**.
2. Modify the `activity_main.xml` file located in the `res/layout` folder by adding the following elements shown in bold:

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

<Button android:id="@+id/btnSave"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="save" />
<Button android:id="@+id/btnOpen"
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
    android:text="Open" />
<ImageButton android:id="@+id/btnImg1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:src="@mipmap/ic_launcher" />
<EditText android:id="@+id/txtName"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
<CheckBox android:id="@+id/chkAutosave"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Autosave" />
<CheckBox android:id="@+id/star"
    style="?android:attr/starStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
<RadioGroup android:id="@+id/rdbGp1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical" >

    <RadioButton android:id="@+id/rdb1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Option 1" />
    <RadioButton android:id="@+id/rdb2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Option 2" />
</RadioGroup>
<ToggleButton android:id="@+id/toggle1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
</LinearLayout>
```

3. To see the views in action, debug the project in Android Studio by pressing Shift+F9. Figure 5-1 shows the various views displayed in the Android emulator.

4. Click each of the views and note how they vary in look and feel. Figure 5-2 shows the following changes to the view:

   - The first `CheckBox` view (Autosave) is checked.
   - The second `CheckBox` View (star) is selected.
   - The second `RadioButton` (Option 2) is selected.
   - The `ToggleButton` is turned on.

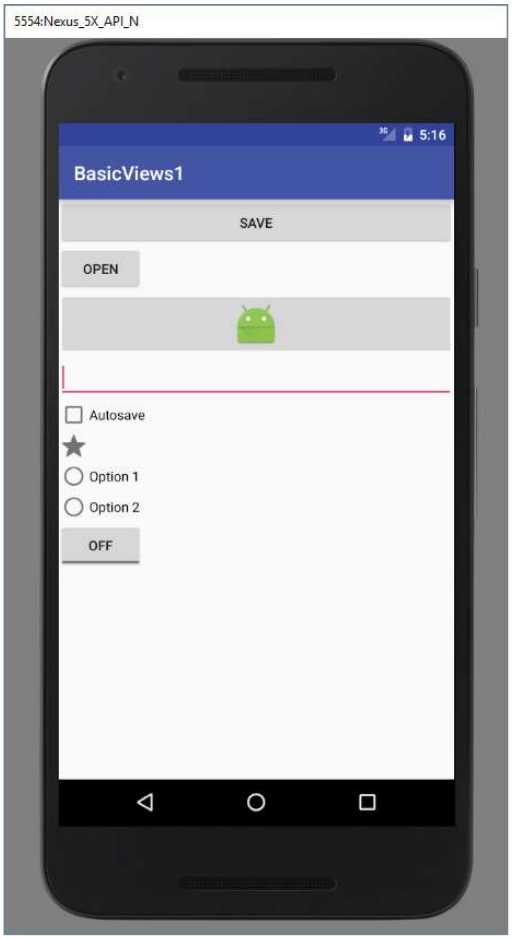

FIGURE 5-1

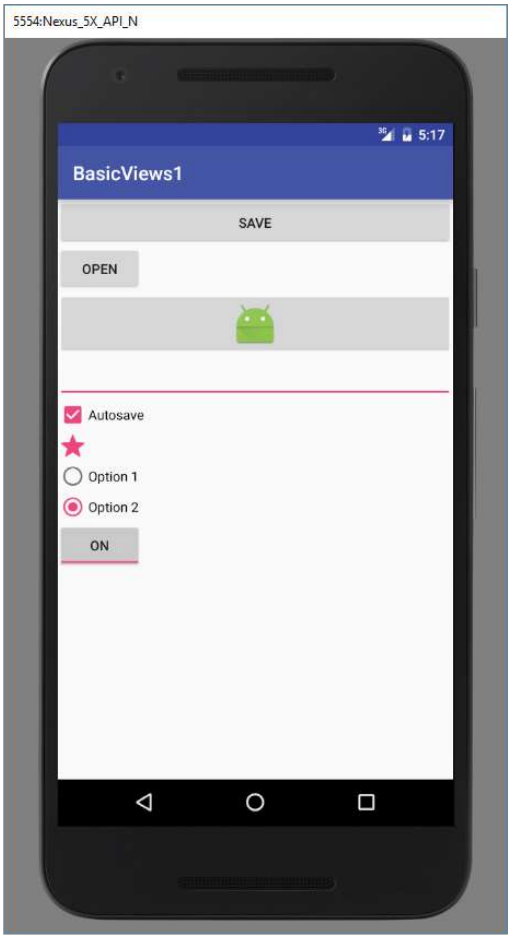

FIGURE 5-2

How It Works

So far, all the views are relatively straightforward. The views are listed using the `<LinearLayout>` element, so they are stacked on top of each other when they are displayed in the activity.

For the first `Button`, the `layout_width` attribute is set to `fill_parent`, which makes its width occupy the entire width of the screen:

```
<Button android:id="@+id/btnSave"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="save" />
```

For the second Button, the `layout_width` attribute is set to `wrap_content` so that its width will be the width of its content—specifically, the text that is displayed (for example, Open):

```
<Button android:id="@+id/btnOpen"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Open" />
```

The `ImageButton` displays a button with an image. You set the image through the `src` attribute. In this code, note that an image has been used for the application icon:

```
<ImageButton android:id="@+id/btnImg1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:src="@drawable/ic_launcher" />
```

The `EditText` view displays a rectangular region in which the user can enter text. In this example, `layout_height` has been set to `wrap_content` so that the text entry location automatically adjusts to fit the amount of text entered by the user (see Figure 5-3).

```
<EditText android:id="@+id/txtName"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
```

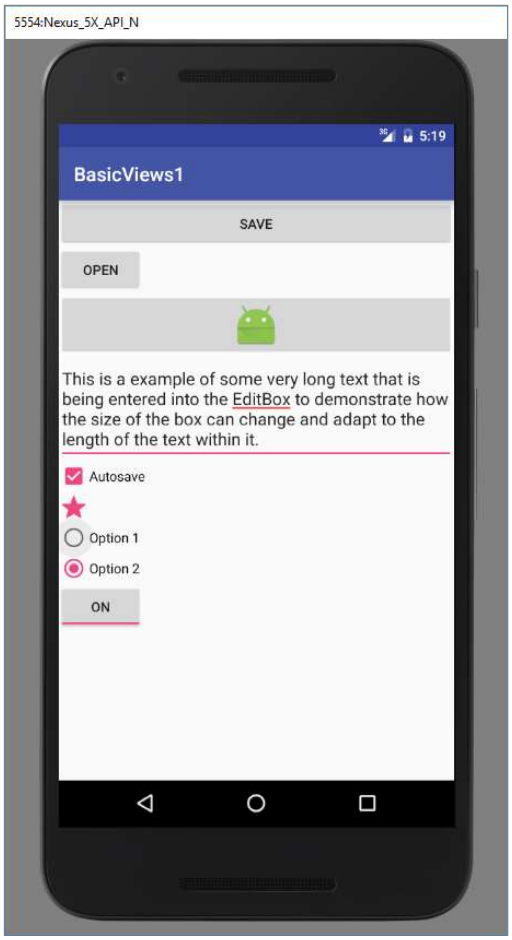

FIGURE 5-3

The `CheckBox` displays a check box that users can tap to check or uncheck:

```
<CheckBox android:id="@+id/chkAutosave"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Autosave" />
```

If you do not like the default look of the `CheckBox`, you can apply a style attribute so that the check mark is replaced by another image, such as a star:

```
<CheckBox android:id="@+id/star"
    style="?android:attr/starStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

The format for the value of the `style` attribute is as follows:

```
?[package:][type:]name
```

The `RadioGroup` encloses two `RadioButtons`. This is important because radio buttons are usually used to present multiple options to the user for selection. When a `RadioButton` in a `RadioGroup` is selected, all other `RadioButtons` are automatically unselected:

```
<RadioGroup android:id="@+id/rdbGp1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical" >

    <RadioButton android:id="@+id/rdb1"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Option 1" />
    <RadioButton android:id="@+id/rdb2"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Option 2" />
</RadioGroup>
```

Notice that the `RadioButtons` are listed vertically, one on top of another. If you want to list them horizontally, you need to change the `orientation` attribute to `horizontal`. You would also need to ensure that the `layout_width` attribute of the `RadioButton` views are set to `wrap_content`:

```
<RadioGroup android:id="@+id/rdbGp1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:orientation="horizontal" >
    <RadioButton android:id="@+id/rdb1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 1" />
    <RadioButton android:id="@+id/rdb2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 2" />
</RadioGroup>
```

Figure 5-4 shows the `RadioButton` views displayed horizontally.

FIGURE 5-4

The `ToggleButton` displays a rectangular button that users can toggle on and off by clicking:

```
<ToggleButton android:id="@+id/toggle1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
```

One thing that has been consistent throughout this example is that each view has the `id` attribute set to a particular value, such as in the case of the `Button`:

```
<Button android:id="@+id/btnSave"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/save" />
```

The `id` attribute is an identifier for a view, which allows it to be retrieved using the `View.findViewById()` or `Activity.findViewById()` methods.

Now that you have seen what the various views for an activity look like, the following Try It Out demonstrates how you can programmatically control them.

TRY IT OUT Handling View Events

1. Using the BasicViews1 project you created in the previous Try It Out, modify the `MainActivity.java` file by adding the following bolded statements:

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.CheckBox;
import android.widget.RadioButton;
import android.widget.RadioGroup;
import android.widget.Toast;
import android.widget.ToggleButton;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        //---Button view---
        Button btnOpen = (Button) findViewById(R.id.btnOpen);
        btnOpen.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                DisplayToast("You have clicked the Open button");
            }
        });
        //---Button view---
        Button btnSave = (Button) findViewById(R.id.btnSave);
        btnSave.setOnClickListener(new View.OnClickListener()
        {
            public void onClick(View v) {
                DisplayToast("You have clicked the Save button");
            }
        });
        //---CheckBox---
        CheckBox checkBox = (CheckBox) findViewById(R.id.chkAutosave);
        checkBox.setOnClickListener(new View.OnClickListener()
        {
            public void onClick(View v) {
                if (((CheckBox)v).isChecked())
                    DisplayToast("CheckBox is checked");
                else
                    DisplayToast("CheckBox is unchecked");
            }
        });
        //---RadioButton---
        RadioGroup radioGroup = (RadioGroup) findViewById(R.id.rdbGp1);
        radioGroup.setOnCheckedChangeListener(
```

```
        new RadioGroup.OnCheckedChangeListener()
            {
                public void onCheckedChanged(RadioGroup group, int checkedId) {
                    RadioButton rb1 = (RadioButton) findViewById(R.id.rdb1);
                    if (rb1.isChecked()) {
                        DisplayToast("Option 1 checked!");
                    } else {
                        DisplayToast("Option 2 checked!");
                    }
                }
            });
            //---ToggleButton---
            ToggleButton toggleButton =
                    (ToggleButton) findViewById(R.id.toggle1);
            toggleButton.setOnClickListener(new View.OnClickListener()
            {
                public void onClick(View v) {
                    if (((ToggleButton)v).isChecked())
                        DisplayToast("Toggle button is On");
                    else
                        DisplayToast("Toggle button is Off");
                }
            });
        }
        private void DisplayToast(String msg)
        {
            Toast.makeText(getBaseContext(), msg,
                    Toast.LENGTH_SHORT).show();
        }
    }
```

2. Press Shift+F9 to debug the project on the Android emulator.
3. Click each of the views and observe the message displayed in the Toast window.

How It Works

To handle the events fired by each view, you first must programmatically locate the view that you created during the `onCreate()` event. You do so using the `findViewById()` method (belonging to the `Activity` base class). You supply the `findViewById()` method with the ID of the view:

```
//---Button view---
Button btnOpen = (Button) findViewById(R.id.btnOpen);
```

The `setOnClickListener()` method registers a callback to be invoked later when the view is clicked:

```
btnOpen.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        displayToast("You have clicked the Open button");
    }
});
```

The `onClick()` method is called when the view is clicked.

To determine the state of the `CheckBox`, you must typecast the argument of the `onClick()` method to a `CheckBox` and then verify its `isChecked()` method to see if it is checked:

```
CheckBox checkBox = (CheckBox) findViewById(R.id.chkAutosave);
checkBox.setOnClickListener(new View.OnClickListener()
{
    public void onClick(View v) {
        if (((CheckBox)v).isChecked())
            DisplayToast("CheckBox is checked");
        else
            DisplayToast("CheckBox is unchecked");
    }
});
```

For the `RadioButton`, you need to use the `setOnCheckedChangeListener()` method on the `RadioGroup` to register a callback to be invoked when the checked `RadioButton` changes in this group. The following code will work for two radio buttons, as in a yes/no selection on a form:

```
//---RadioButton---
RadioGroup radioGroup = (RadioGroup) findViewById(R.id.rdbGp1);
radioGroup.setOnCheckedChangeListener(new OnCheckedChangeListener()
{
    public void onCheckedChanged(RadioGroup group, int checkedId) {
        RadioButton rb1 = (RadioButton) findViewById(R.id.rdb1);
        if (rb1.isChecked()) {
            DisplayToast("Option 1 checked!");
        } else {
            DisplayToast("Option 2 checked!");
        }
    }
});
```

When a `RadioButton` is selected, the `onCheckedChanged()` method is fired. Within it, locate individual `RadioButton` views and then call each `isChecked()` method to determine which `RadioButton` is selected. Alternatively, the `onCheckedChanged()` method contains a second argument that contains a unique identifier of the selected `RadioButton`.

The `ToggleButton` works just like the `CheckBox`.

So far, to handle the events on the views, you first had to get a reference to the view and then register a callback to handle the event. However, there is another way to handle view events. Using the `Button` as an example, you can add an attribute called `onClick`:

```
<Button android:id="@+id/btnSave"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/save"
    android:onClick="btnSaved_clicked"/>
```

The `onClick` attribute specifies the click event of the button. The value of this attribute is the name of the event handler. Therefore, to handle the button's click event, you simply create a method called

`btnSaved_clicked`, as shown in the following example (note that the method must have a single parameter of type `View`):

```
public class BasicViews1Activity extends Activity {

    public void btnSaved_clicked (View view) {
        DisplayToast("You have clicked the Save button1");
    }

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        //...
    }
    private void DisplayToast(String msg)
    {
        Toast.makeText(getBaseContext(), msg,
                Toast.LENGTH_SHORT).show();
    }
}
```

If you compare the `onClick` approach to the callback method discussed previously, you'll see this is much simpler. Which method you use is really up to you, but this book mostly uses the `onClick` approach.

ProgressBar View

The `ProgressBar` view provides visual feedback about some ongoing tasks, such as when you are performing a task in the background. For example, you might be downloading some data from the web and need to update the user about the status of the download. In this case, the `ProgressBar` view is a good choice. The following activity demonstrates how to use the `ProgressBar` view.

TRY IT OUT Using the ProgressBar View (BasicViews2.zip)

1. Using Android Studio, create an Android project and name it **BasicViews2**.
2. Modify the `activity_main.xml` file located in the `res/layout` folder by adding the following code in bold:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<ProgressBar android:id="@+id/progressbar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
</LinearLayout>
```

3. In the `MainActivity.java` file, add the following bolded statements:

```
import android.os.Handler;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ProgressBar;

public class MainActivity extends AppCompatActivity {
    static int progress;
    ProgressBar progressBar;
    int progressStatus = 0;
    Handler handler = new Handler();
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        progress = 0;
        progressBar = (ProgressBar) findViewById(R.id.progressbar);
        //---do some work in background thread---
        new Thread(new Runnable()
        {
            public void run()
            {
                //---do some work here---
                while (progressStatus < 10)
                {
                    progressStatus = doSomeWork();
                }
                //---hides the progress bar---
                handler.post(new Runnable()
                {
                    public void run()
                    {
                        //---0 - VISIBLE; 4 - INVISIBLE; 8 - GONE---
                        progressBar.setVisibility(View.GONE);
                    }
                });
            }
            //---do some long running work here---
            private int doSomeWork()
            {
                try {
                    //---simulate doing some work---
                    Thread.sleep(500);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
                return ++progress;
            }
        }).start();
    }
}
```

4. Press Shift+F9 to debug the project on the Android emulator. Figure 5-5 shows the `ProgressBar` animating. After about five seconds, it disappears.

FIGURE 5-5

How It Works

The default mode of the `ProgressBar` view is indeterminate—that is, it shows a cyclic animation. This mode is useful for tasks that do not have specific completion times, such as when you are sending some data to a web service and waiting for the server to respond. If you simply put the `<ProgressBar>` element in your `main.xml` file, it continuously displays a spinning icon. It is your responsibility to stop it when your background task has completed.

The code added to the `MainActivity.java` file shows how you can spin off a background thread to simulate performing some long-running tasks. To do so, use the `Thread` class with a `Runnable` object. The `run()` method starts the execution of the thread, which in this case calls the `doSomeWork()` method to simulate doing some work. When the simulated work is done (after about five seconds), use a `Handler` object to send a message to the thread to dismiss the `ProgressBar`:

```
//---do some work in background thread---
new Thread(new Runnable()
{
```

```
        public void run()
        {
            //---do some work here---
            while (progressStatus < 10)
            {
                progressStatus = doSomeWork();
            }
            //---hides the progress bar---
            handler.post(new Runnable()
            {
                public void run()
                {
                    //---0 - VISIBLE; 4 - INVISIBLE; 8 - GONE---
                    progressBar.setVisibility(View.GONE);
                }
            });
        }
        //---do some long running work here---
        private int doSomeWork()
        {
            try {
                //---simulate doing some work---
                Thread.sleep(500);
            } catch (InterruptedException e)
            {
                e.printStackTrace();
            }
            return ++progress;
        }
    }).start();
```

When the task is completed, hide the `ProgressBar` by setting its `Visibility` property to `View.GONE` (value `8`). There are two differences between the `INVISIBLE` constant and the `GONE` constant:

- `INVISIBLE` constant simply hides the `ProgressBar` (the region occupied by the `ProgressBar` is still taking up space in the activity).
- `GONE` constant removes the `ProgressBar` view from the activity and does not take up any space.

The next Try It Out shows how you can change the look of the `ProgressBar`.

TRY IT OUT Customizing the ProgressBar View

1. Using the BasicViews2 project created in the previous Try It Out, modify the `activity_main.xml` file as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<ProgressBar android:id="@+id/progressbar"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@android:style/Widget.ProgressBar.Horizontal" />
</LinearLayout>
```

2. Modify the `MainActivity.java` file by adding the following bolded statements:

```
import android.os.Handler;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.ProgressBar;

public class MainActivity extends AppCompatActivity {
    static int progress;
    ProgressBar progressBar;
    int progressStatus = 0;
    Handler handler = new Handler();
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        progress = 0;
        progressBar = (ProgressBar) findViewById(R.id.progressbar);
        progressBar.setMax(200);

        //---do some work in background thread---
        new Thread(new Runnable()
        {
            public void run()
            {
                //---do some work here---
                while (progressStatus < 100)
                 {
                     progressStatus = doSomeWork();
                     //---Update the progress bar---
                     handler.post(new Runnable()
                     {
                         public void run() {
                             progressBar.setProgress(progressStatus);
                         }
                     });
                 }
                 //---hides the progress bar---
                 handler.post(new Runnable()
                 {
                     public void run()
                     {
                         //---0 - VISIBLE; 4 - INVISIBLE; 8 - GONE---
                         progressBar.setVisibility(View.GONE);
                     }
                 });
```

```
            }
            //---do some long running work here---
            private int doSomeWork()
            {
                try {
                    //---simulate doing some work---
                    Thread.sleep(500);
                } catch (InterruptedException e)
                {
                    e.printStackTrace();
                }
                return ++progress;
            }
        }).start();
    }
}
```

3. Press Shift+F9 to debug the project on the Android emulator.
4. Figure 5-6 shows the `ProgressBar` displaying the progress. The `ProgressBar` disappears when the progress reaches 50 percent.

FIGURE 5-6

How It Works

To make the `ProgressBar` display horizontally, simply set its `style` attribute to `@android:style/Widget.ProgressBar.Horizontal`:

```
<ProgressBar android:id="@+id/progressbar"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@android:style/Widget.ProgressBar.Horizontal" />
```

To display the progress, call its `setProgress()` method, passing in an integer indicating its progress:

```
//---Update the progress bar---
handler.post(new Runnable()
{
    public void run() {
        progressBar.setProgress(progressStatus);
    }
});
```

In this example, set the range of the `ProgressBar` from 0 to 200 (via the `setMax()` method). This causes the `ProgressBar` to stop and then disappear when it is halfway through (because you only continue to call the `doSomeWork()` method as long as the `progressStatus` is less than 100). To ensure that the `ProgressBar` disappears only when the progress reaches 100%, either set the maximum value to 100 or modify the `while` loop to stop when the `progressStatus` reaches 200, like this:

```
//---do some work here---
while (progressStatus < 200)
```

Besides the horizontal style for the `ProgressBar` that you have used for this example, you can also use the following constants:

- `Widget.ProgressBar.Horizontal`
- `Widget.ProgressBar.Small`
- `Widget.ProgressBar.Large`
- `Widget.ProgressBar.Inverse`
- `Widget.ProgressBar.Small.Inverse`
- `Widget.ProgressBar.Large.Inverse`

AutoCompleteTextView View

The `AutoCompleteTextView` is a view that is similar to `EditText` (in fact it is a subclass of `EditText`), except that it automatically shows a list of completion suggestions while the user is typing. The following Try It Out shows how to use the `AutoCompleteTextView` to automatically help users complete the text entry.

TRY IT OUT Using the AutoCompleteTextView (BasicViews3.zip)

1. Using Android Studio, create an Android project and name it **BasicViews3**.
2. Modify the `activity_main.xml` file located in the `res/layout` folder as shown here in bold:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Name of President" />
<AutoCompleteTextView android:id="@+id/txtCountries"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content" />
</LinearLayout>
```

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

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

public class MainActivity extends AppCompatActivity {
    String[] presidents = {
            "Dwight D. Eisenhower",
            "John F. Kennedy",
            "Lyndon B. Johnson",
            "Richard Nixon",
            "Gerald Ford",
            "Jimmy Carter",
            "Ronald Reagan",
            "George H. W. Bush",
            "Bill Clinton",
            "George W. Bush",
            "Barack Obama"
    };
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
                android.R.layout.simple_dropdown_item_1line, presidents);
        AutoCompleteTextView textView = (AutoCompleteTextView)
                findViewById(R.id.txtCountries);
        textView.setThreshold(3);
        textView.setAdapter(adapter);
    }
}
```

4. Press Shift+F9 to debug the application on the Android emulator. As shown in Figure 5-7, a list of matching names appears as you type into the `AutoCompleteTextView` field.

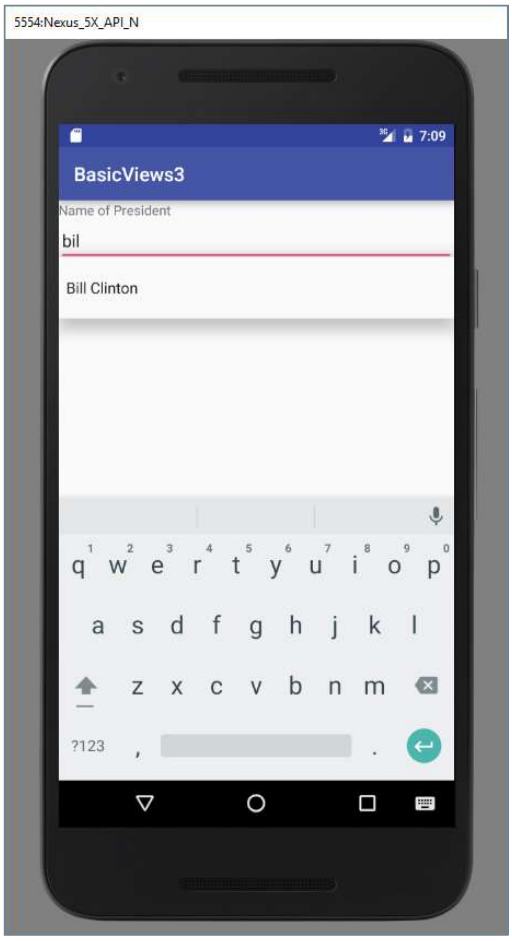

FIGURE 5-7

How It Works

In the `BasicViews3Activity` class, you first create a `String` array containing a list of presidents' names:

```
String[] presidents = {
        "Dwight D. Eisenhower",
        "John F. Kennedy",
        "Lyndon B. Johnson",
        "Richard Nixon",
        "Gerald Ford",
        "Jimmy Carter",
        "Ronald Reagan",
        "George H. W. Bush",
        "Bill Clinton",
```

```
        "George W. Bush",
        "Barack Obama"
    };
```

The `ArrayAdapter` object manages the array of strings that are displayed by the `AutoCompleteTextView`. In the preceding example, you set the `AutoCompleteTextView` to display in the `simple_dropdown_item_1line` mode:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
    android.R.layout.simple_dropdown_item_1line, presidents);
```

The `setThreshold()` method sets the minimum number of characters the user must type before the suggestions appear as a drop-down menu:

```
textView.setThreshold(3);
```

The list of suggestions to display for the `AutoCompleteTextView` is obtained from the `ArrayAdapter` object:

```
textView.setAdapter(adapter);
```

USING PICKER VIEWS

Selecting a date and time is one of the common tasks you need to perform in a mobile application. Android supports this functionality through the `TimePicker` and `DatePicker` views. The following sections demonstrate how to use these views in your activity.

TimePicker View

The `TimePicker` view enables users to select a time of the day, in either 24-hour mode or AM/PM mode. The following Try It Out shows you how to use the `TimePicker` in the latest version of the Android SDK. When you are creating the project for this sample, be sure that you choose an SDK that is level 23 or greater.

TRY IT OUT Using the TimePicker View (BasicViews4.zip)

1. Using Android Studio, create an Android project and name it **BasicViews4**.
2. Modify the `activity_main.xml` file located in the `res/layout` folder by adding the following bolded lines:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >
<TimePicker android:id="@+id/timePicker"
    android:layout_width="wrap_content"
```

web for display. In this scenario, saving the images to internal storage is a good solution. You might also need to persist data created by the user, such as when you have an application that enables users to take notes and save them for later use. In both of these scenarios, using the internal storage is a good choice.

- There are times when you need to share your application data with other users. For example, you might create an Android application that logs the coordinates of the locations that a user has been to, and subsequently, you want to share all this data with other users. In this scenario, you can store your files on the SD card of the device so that users can easily transfer the data to other devices (and computers) for use later.

CREATING AND USING DATABASES

So far, all the techniques you have seen are useful for saving simple sets of data. For saving relational data, using a database is much more efficient. For example, if you want to store the test results of all the students in a school, it is much more efficient to use a database to represent them because you can use database querying to retrieve the results of specific students. Moreover, using databases enables you to enforce data integrity by specifying the relationships between different sets of data.

Android uses the SQLite database system. The database that you create for an application is only accessible to itself; other applications will not be able to access it.

In this section, you find out how to programmatically create a SQLite database in your Android application. For Android, the SQLite database that you create programmatically in an application is always stored in the `/data/data/<package_name>/databases` folder.

Creating the DBAdapter Helper Class

A good practice for dealing with databases is to create a helper class to encapsulate all the complexities of accessing the data so that it is transparent to the calling code. For this section, you create a helper class called `DBAdapter`, which creates, opens, closes, and uses a SQLite database.

In this example, you are going to create a database named `MyDB` containing one table named `contacts`. This table has three columns: `_id`, `name`, and `email`.

TRY IT OUT Creating the Database Helper Class (Databases.zip)

1. Using Android Studio, create an Android project and name it **Databases**.
2. Add a new Java Class file to the package and name it **DBAdapter** (see Figure 7-10).

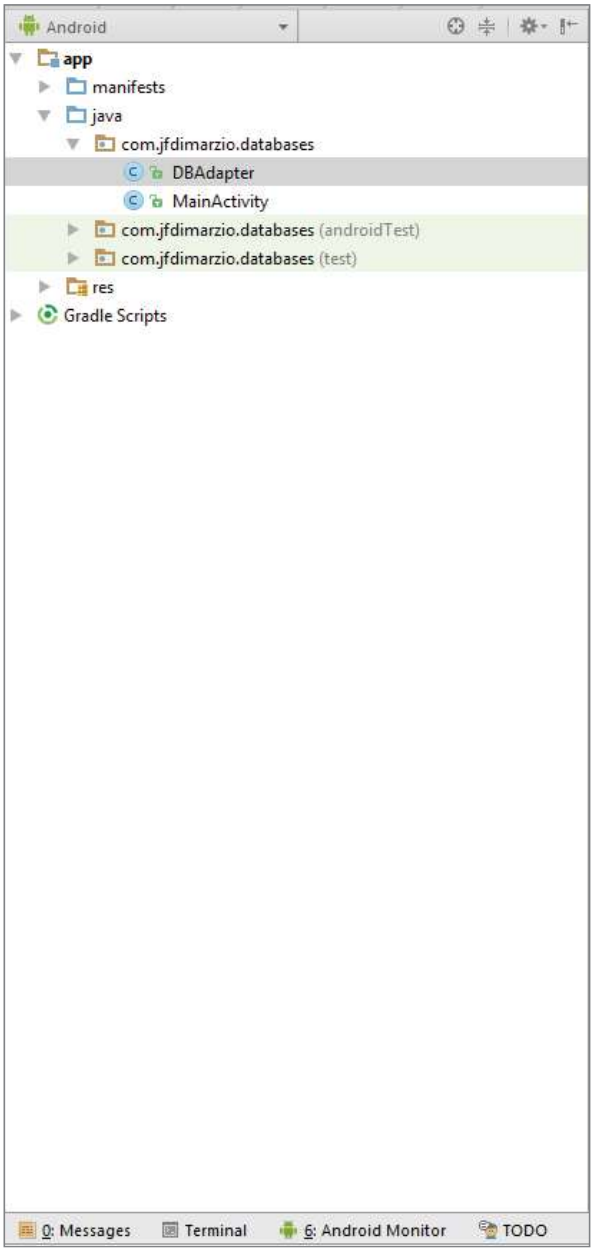

FIGURE 7-10

3. Add the following bolded statements to the `DBAdapter.java` file:

```
import android.content.ContentValues;
import android.content.Context;
```

```
import android.database.Cursor;
import android.database.SQLException;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
import android.util.Log;

public class DBAdapter {
    static final String KEY_ROWID = "_id";
    static final String KEY_NAME = "name";
    static final String KEY_EMAIL = "email";
    static final String TAG = "DBAdapter";
    static final String DATABASE_NAME = "MyDB";
    static final String DATABASE_TABLE = "contacts";
    static final int DATABASE_VERSION = 1;
    static final String DATABASE_CREATE =
            "create table contacts (_id integer primary key autoincrement, "
                    + "name text not null, email text not null);";
    final Context context;
    DatabaseHelper DBHelper;
    SQLiteDatabase db;

    public DBAdapter(Context ctx)
    {
        this.context = ctx;
        DBHelper = new DatabaseHelper(context);
    }
    private static class DatabaseHelper extends SQLiteOpenHelper
    {
        DatabaseHelper(Context context)
        {
            super(context, DATABASE_NAME, null, DATABASE_VERSION);
        }
        @Override
        public void onCreate(SQLiteDatabase db)
        {
            try {
                db.execSQL(DATABASE_CREATE);
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
        {
            Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
                    + newVersion + ", which will destroy all old data");
            db.execSQL("DROP TABLE IF EXISTS contacts");
            onCreate(db);
        }
    }
    //---opens the database---
    public DBAdapter open() throws SQLException
```

```
    {
        db = DBHelper.getWritableDatabase();
        return this;
    }
    //---closes the database---
    public void close()
    {
        DBHelper.close();
    }
    //---insert a contact into the database---
    public long insertContact(String name, String email)
    {
        ContentValues initialValues = new ContentValues();
        initialValues.put(KEY_NAME, name);
        initialValues.put(KEY_EMAIL, email);
        return db.insert(DATABASE_TABLE, null, initialValues);
    }
    //---deletes a particular contact---
    public boolean deleteContact(long rowId)
    {
        return db.delete(DATABASE_TABLE, KEY_ROWID + "=" + rowId, null) > 0;
    }
    //---retrieves all the contacts---
    public Cursor getAllContacts()
    {
        return db.query(DATABASE_TABLE, new String[] {KEY_ROWID, KEY_NAME,
                KEY_EMAIL}, null, null, null, null, null);
    }
    //---retrieves a particular contact---
    public Cursor getContact(long rowId) throws SQLException
    {
        Cursor mCursor =
                db.query(true, DATABASE_TABLE, new String[] {KEY_ROWID,
                                KEY_NAME, KEY_EMAIL},
     KEY_ROWID + "=" + rowId, null,
                        null, null, null, null);
        if (mCursor != null) {
            mCursor.moveToFirst();
        }
        return mCursor;
    }
    //---updates a contact---
    public boolean updateContact(long rowId, String name, String email)
    {
        ContentValues args = new ContentValues();
        args.put(KEY_NAME, name);
        args.put(KEY_EMAIL, email);
        return db.update(DATABASE_TABLE, args, KEY_ROWID + "=" + rowId, null) >
0;
    }
}
```

How It Works

You first define several constants to contain the various fields for the table that you are going to create in your database:

```
static final String KEY_ROWID = "_id";
static final String KEY_NAME = "name";
static final String KEY_EMAIL = "email";
static final String TAG = "DBAdapter";
static final String DATABASE_NAME = "MyDB";
static final String DATABASE_TABLE = "contacts";
static final int DATABASE_VERSION = 1;
static final String DATABASE_CREATE =
    "create table contacts (_id integer primary key autoincrement, "
    + "name text not null, email text not null);";
```

In particular, the `DATABASE_CREATE` constant contains the SQL statement for creating the `contacts` table within the `MyDB` database.

Within the `DBAdapter` class, you also add a private class that extends the `SQLiteOpenHelper` class. `SQLiteOpenHelper` is a helper class in Android to manage database creation and version management. In particular, you must override the `onCreate()` and `onUpgrade()` methods:

```
private static class DatabaseHelper extends SQLiteOpenHelper
{
    DatabaseHelper(Context context)
    {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db)
    {
        try {
            db.execSQL(DATABASE_CREATE);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)
    {
        Log.w(TAG, "Upgrading database from version " + oldVersion + " to "
                + newVersion + ", which will destroy all old data");
        db.execSQL("DROP TABLE IF EXISTS contacts");
        onCreate(db);
    }
}
```

The `onCreate()` method creates a new database if the required database is not present. The `onUpgrade()` method is called when the database needs to be upgraded. This is achieved by checking the value defined in the `DATABASE_VERSION` constant. For this implementation of the `onUpgrade()` method, you simply drop the table and create it again.

You can then define the various methods for opening and closing the database, as well as the methods for adding/editing/deleting rows in the table:

```
//---opens the database---
public DBAdapter open() throws SQLException
{
    db = DBHelper.getWritableDatabase();
    return this;
}
//---closes the database---
public void close()
{
    DBHelper.close();
}
//---insert a contact into the database---
public long insertContact(String name, String email)
{
    ContentValues initialValues = new ContentValues();
    initialValues.put(KEY_NAME, name);
    initialValues.put(KEY_EMAIL, email);
    return db.insert(DATABASE_TABLE, null, initialValues);
}
//---deletes a particular contact---
public boolean deleteContact(long rowId)
{
    return db.delete(DATABASE_TABLE, KEY_ROWID + "=" + rowId, null) > 0;
}
//---retrieves all the contacts---
public Cursor getAllContacts()
{
    return db.query(DATABASE_TABLE, new String[] {KEY_ROWID, KEY_NAME,
            KEY_EMAIL}, null, null, null, null, null);
}
//---retrieves a particular contact---
public Cursor getContact(long rowId) throws SQLException
{
    Cursor mCursor =
            db.query(true, DATABASE_TABLE, new String[] {KEY_ROWID,
            KEY_NAME, KEY_EMAIL}, KEY_ROWID + "=" + rowId, null,
            null, null, null, null);
    if (mCursor != null) {
        mCursor.moveToFirst();
    }
    return mCursor;
}
//---updates a contact---
public boolean updateContact(long rowId, String name, String email)
{
    ContentValues args = new ContentValues();
    args.put(KEY_NAME, name);
    args.put(KEY_EMAIL, email);
    return db.update(DATABASE_TABLE, args, KEY_ROWID + "=" + rowId, null) > 0;
}
```

Notice that Android uses the `Cursor` class as a return value for queries. Think of the `Cursor` as a pointer to the result set from a database query. Using `Cursor` enables Android to more efficiently manage rows and columns as needed.

You use a `ContentValues` object to store name/value pairs. Its `put()` method enables you to insert keys with values of different data types.

To create a database in your application using the `DBAdapter` class, you create an instance of the `DBAdapter` class:

```
public DBAdapter(Context ctx)
{
    this.context = ctx;
    DBHelper = new DatabaseHelper(context);
}
```

The constructor of the `DBAdapter` class will then create an instance of the `DatabaseHelper` class to create a new database:

```
DatabaseHelper(Context context)
{
    super(context, DATABASE_NAME, null, DATABASE_VERSION);
}
```

Using the Database Programmatically

With the `DBAdapter` helper class created, you are now ready to use the database. In the following sections, you will learn how to perform the regular CRUD (create, read, update and delete) operations commonly associated with databases.

Adding Contacts

The following Try It Out demonstrates how you can add a contact to the table.

TRY IT OUT Adding Contacts to a Table (Databases.zip)

1. Using the same project created earlier, add the following bolded statements to the `MainActivity.java` file:

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

public class MainActivity extends AppCompatActivity {

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

        DBAdapter db = new DBAdapter(this);
```

```
        //---add a contact---
        db.open();
        long id = db.insertContact("Jennifer Ann",
"jenniferann@jfdimarzio.com");
        id = db.insertContact("Oscar Diggs", "oscar@oscardiggs.com");
        db.close();
    }
}
```

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

How It Works

In this example, you create an instance of the `DBAdapter` class:

```
DBAdapter db = new DBAdapter(this);
```

The `insertContact()` method returns the ID of the inserted row. If an error occurs during the operation, it returns –1.

Retrieving All the Contacts

To retrieve all the contacts in the `contacts` table, use the `getAllContacts()` method of the `DBAdapter` class, as the following Try It Out shows.

TRY IT OUT Retrieving All Contacts from a Table (Databases.zip)

1. Using the same project created earlier, add the following bolded statements to the `MainActivity.java` file:

```
import android.database.Cursor;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

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

        DBAdapter db = new DBAdapter(this);
        //---add a contact---
        db.open();
        long id = db.insertContact("Jennifer Ann", "jenniferann@jfdimarzio.com");
        id = db.insertContact("Oscar Diggs", "oscar@oscardiggs.com");
        db.close();
        db.open();
        Cursor c = db.getAllContacts();
        if (c.moveToFirst())
```

```
        {
            do {
                DisplayContact(c);
            } while (c.moveToNext());
        }
        db.close();
    }

    public void DisplayContact(Cursor c)
    {
        Toast.makeText(this,
                "id: " + c.getString(0) + "\n" +
        "Name: " + c.getString(1) + "\n" +
        "Email:  " + c.getString(2),
            Toast.LENGTH_LONG).show();
    }
}
```

2. Press Shift+F9 to debug the application on the Android emulator. Figure 7-11 shows the `Toast` class displaying the contacts retrieved from the database.

FIGURE 7-11

How It Works

The `getAllContacts()` method of the `DBAdapter` class retrieves all the contacts stored in the database. The result is returned as a `Cursor` object. To display all the contacts, you first need to call the `moveToFirst()` method of the `Cursor` object. If it succeeds (which means at least one row is available), then you display the details of the contact using the `DisplayContact()` method. To move to the next contact, call the `moveToNext()` method of the `Cursor` object.

Retrieving a Single Contact

To retrieve a single contact using its ID, call the `getContact()` method of the `DBAdapter` class, as the following Try It Out shows.

TRY IT OUT Retrieving a Contact from a Table (Databases.zip)

1. Using the same project created earlier, add the following bolded statements to the `MainActivity.java` file:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    DBAdapter db = new DBAdapter(this);
    /*
    //---add a contact---
    ...
    //--get all contacts---
    ...
    db.close();
    */

    //---get a contact---
    db.open();
    Cursor c = db.getContact(2);
    if (c.moveToFirst())
        DisplayContact(c);
    else
        Toast.makeText(this, "No contact found", Toast.LENGTH_LONG).show();
    db.close();
}
```

2. Press Shift+F9 to debug the application on the Android emulator. The details of the second contact are displayed using the `Toast` class.

How It Works

The `getContact()` method of the `DBAdapter` class retrieves a single contact using its ID. You pass in the ID of the contact. In this case, you pass in an ID of 2 to indicate that you want to retrieve the second contact:

```
Cursor c = db.getContact(2);
```

The result is returned as a `Cursor` object. If a row is returned, you display the details of the contact using the `DisplayContact()` method. Otherwise, you display a message using the `Toast` class.

Updating a Contact

To update a particular contact, call the `updateContact()` method in the `DBAdapter` class by passing the ID of the contact you want to update, as the following Try It Out shows.

TRY IT OUT Updating a Contact in a Table (Databases.zip)

1. Using the same project created earlier, add the following bolded statements to the `MainActivity.java` file:

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        DBAdapter db = new DBAdapter(this);
        /*
        //---add a contact---
        ...
        //--get all contacts---
        ...
        //---get a contact---
        ...
        db.close();
        */

        //---update contact---
        db.open();
        if (db.updateContact(1, "Oscar Diggs", "oscar@oscardiggs.com"))
            Toast.makeText(this, "Update successful.",
Toast.LENGTH_LONG).show();
        else
            Toast.makeText(this, "Update failed.", Toast.LENGTH_LONG).show();
        db.close();
    }
```

2. Press Shift+F9 to debug the application on the Android emulator. A message is displayed if the update is successful.

How It Works

The `updateContact()` method in the `DBAdapter` class updates a contact's details by using the ID of the contact you want to update. It returns a Boolean value, indicating whether the update was successful.

Deleting a Contact

To delete a contact, use the `deleteContact()` method in the `DBAdapter` class by passing the ID of the contact you want to update, as the following Try It Out shows.

TRY IT OUT Deleting a Contact from a Table (Databases.zip)

1. Using the same project created earlier, add the following bolded statements to the `MainActivity.java` file:

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    DBAdapter db = new DBAdapter(this);
     /*
     //---add a contact---
     ...
     //--get all contacts---
     ...
     //---get a contact---
     ...
     //---update contact---
     ...
     db.close();
     */

     //---delete a contact---
     db.open();
     if (db.deleteContact(1))
         Toast.makeText(this, "Delete successful.",
Toast.LENGTH_LONG).show();
     else
         Toast.makeText(this, "Delete failed.", Toast.LENGTH_LONG).show();
     db.close();
}
```

2. Press Shift+F9 to debug the application on the Android emulator. A message is displayed if the deletion was successful.

How It Works

The `deleteContact()` method in the `DBAdapter` class deletes a contact using the ID of the contact you want to delete. It returns a Boolean value, indicating whether the deletion was successful.

Upgrading the Database

Sometimes, after creating and using the database, you might need to add additional tables, change the schema of the database, or add columns to your tables. In this case, you need to migrate your existing data from the old database to a newer one.

To upgrade the database, change the `DATABASE_VERSION` constant to a value higher than the previous one. For example, if its previous value was 1, change it to 2:

```
public class DBAdapter {
    static final String KEY_ROWID = "_id";
    static final String KEY_NAME = "name";
    static final String KEY_EMAIL = "email";
```

```
static final String TAG = "DBAdapter";
static final String DATABASE_NAME = "MyDB";
static final String DATABASE_TABLE = "contacts";
static final int DATABASE_VERSION = 2;
```

> ***NOTE*** *Before you run this example, be sure to comment out the block of delete statements described in the previous section. If you don't, the deletion fails because the table in the database is dropped (deleted).*

When you run the application one more time, you see the following message in the logcat window of Android Studio:

```
DBAdapter(8705): Upgrading database from version 1 to 2, which
will destroy all old data
```

For simplicity, simply drop the existing table and create a new one. In real life, you usually back up your existing table and then copy it over to the new table.

SUMMARY

In this chapter, you were introduced to the different ways to save persistent data to your Android device. For simple unstructured data, using the `SharedPreferences` object is the ideal solution. If you need to store bulk data then consider using the traditional file system. Finally, for structured data, it is more efficient to store it in a relational database management system. For this, Android provides the SQLite database, which you can access easily using the APIs exposed.

Note that for the `SharedPreferences` object and the SQLite database, the data is accessible only by the application that creates it. In other words, it is not shareable. If you need to share data among different applications, you need to create a *content provider*. Content providers are discussed in more detail in Chapter 8.

EXERCISES

1. How do you display the preferences of your application using an activity?
2. Name the method that enables you to obtain the external storage path for an Android device.
3. What method is called when a database needs to be upgraded?

You can find answers to the exercises in the appendix.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	KEY CONCEPTS
Saving simple user data	Use the `SharedPreferences` object.
Sharing data among activities in the same application	Use the `getSharedPreferences()` method.
Saving to a file	Use the `FileOutputStream` and `OutputStreamReader` classes.
Reading from a file	Use the `FileInputStream` and `InputStreamReader` classes.
Saving to external storage	Use the `getExternalStorageDirectory()` method to return the path to the external storage.
Accessing files in the `res/raw folder`	Use the `openRawResource()` method in the `Resources` object (obtained via the `getResources()` method).
Creating a database helper class	Extend the `SQLiteOpenHelper` class.