7

Exploring User Interface Screen Elements

Most Android applications inevitably need some form of user interface. In this chapter, we discuss the user interface elements available within the Android Software Development Kit (SDK). Some of these elements display information to the user, whereas others gather information from the user.

You learn how to use a variety of different components and controls to build a screen and how your application can listen for various actions performed by the user. Finally, you learn how to style controls and apply themes to entire screens.

Introducing Android Views and Layouts

Before we go any further, we need to define a few terms. This gives you a better understanding of certain capabilities provided by the Android SDK before they are fully introduced. First, let's talk about the `View` and what it is to the Android SDK.

Introducing the Android View

The Android SDK has a Java packaged named `android.view`. This package contains a number of interfaces and classes related to drawing on the screen. However, when we refer to the `View` object, we actually refer to only one of the classes within this package: the `android.view.View` class.

The `View` class is the basic user interface building block within Android. It represents a rectangular portion of the screen. The `View` class serves as the base class for nearly all the user interface controls and layouts within the Android SDK.

Introducing the Android Control

The Android SDK contains a Java package named `android.widget`. When we refer to controls, we are typically referring to a class within this package. The Android SDK includes classes to draw most common objects, including `ImageView`, `FrameLayout`,

EditText, and Button classes. As mentioned previously, all controls are typically derived from the View class.

This chapter is primarily about controls that display and collect data from the user. We cover many of these basic controls in detail.

> **Note**
>
> Do not confuse the user interface controls in the android.widget package with App Widgets. An AppWidget (android.appwidget) is an application extension, often displayed on the Android Home screen. We discuss App Widgets in more depth in Chapter 22, "Extending Android Application Reach."

Introducing the Android Layout

One special type of control found within the android.widget package is called a layout. A layout control is still a View object, but it doesn't actually draw anything specific on the screen. Instead, it is a parent container for organizing other controls (children). Layout controls determine how and where on the screen child controls are drawn. Each type of layout control draws its children using particular rules. For instance, the LinearLayout control draws its child controls in a single horizontal row or a single vertical column. Similarly, a TableLayout control displays each child control in tabular format (in cells within specific rows and columns).

In Chapter 8, "Designing User Interfaces with Layouts," we organize various controls within layouts and other containers. These special View controls, which are derived from the android.view.ViewGroup class, are useful only after you understand the various display controls these containers can hold. By necessity, we use some of the layout View objects within this chapter to illustrate how to use the controls previously mentioned. However, we don't go into the details of the various layout types available as part of the Android SDK until the next chapter.

> **Note**
>
> Many of the code examples provided in this chapter are taken from the ViewSamples application. This source code for the ViewSamples application is provided for download on the book website.

Displaying Text to Users with TextView

One of the most basic user interface elements, or controls, in the Android SDK is the TextView control. You use it, quite simply, to draw text on the screen. You primarily use it to display fixed text strings or labels.

Frequently, the TextView control is a child control within other screen elements and controls. As with most of the user interface elements, it is derived from View and is within the android.widget package. Because it is a View, all the standard attributes such as width, height, padding, and visibility can be applied to the object. However, as a text-displaying control, you can apply many other TextView-specific attributes to control behavior and how the text is viewed in a variety of situations.

First, though, let's see how to put some quick text up on the screen. <TextView> is the XML layout file tag used to display text on the screen. You can set the android:text property of the TextView to be either a raw text string in the layout file or a reference to a string resource.

Here are examples of both methods you can use to set the android:text attribute of a TextView. The first method sets the text attribute to a raw string; the second method uses a string resource called sample_text, which must be defined in the strings.xml resource file.

```
<TextView
    android:id="@+id/TextView01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Some sample text here" />
<TextView
    android:id="@+id/TextView02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/sample_text" />
```

To display this TextView on the screen, all your Activity needs to do is call the setContentView() method with the layout resource identifier in which you defined the preceding XML shown. You can change the text displayed programmatically by calling the setText() method on the TextView object. Retrieving the text is done with the getText() method.

Now let's talk about some of the more common attributes of TextView objects.

Configuring Layout and Sizing

The TextView control has some special attributes that dictate how the text is drawn and flows. You can, for instance, set the TextView to be a single line high and a fixed width. If, however, you put a long string of text that can't fit, the text truncates abruptly. Luckily, there are some attributes that can handle this problem.

> **Tip**
>
> When looking through the attributes available to TextView objects, you should be aware that the TextView class contains all the functionality needed by editable controls. This means that many of the attributes apply only to *input* fields, which are used primarily by the subclass EditText object. For example, the autoText attribute, which helps the user by fixing common spelling mistakes, is most appropriately set on editable text fields (EditText). There is no need to use this attribute normally when you are simply displaying text.

The width of a TextView can be controlled in terms of the ems measurement rather than in pixels. An em is a term used in typography that is defined in terms of the point size of a particular font. (For example, the measure of an em in a 12-point font is 12 points.) This measurement provides better control over how much text is viewed, regardless of the

font size. Through the `ems` attribute, you can set the width of the `TextView`. Additionally, you can use the `maxEms` and `minEms` attributes to set the maximum width and minimum width, respectively, of the `TextView` in terms of `ems`.

The height of a `TextView` can be set in terms of lines of text rather than pixels. Again, this is useful for controlling how much text can be viewed regardless of the font size. The `lines` attribute sets the number of lines that the `TextView` can display. You can also use `maxLines` and `minLines` to control the maximum height and minimum height, respectively, that the `Textview` displays.

Here is an example that combines these two types of sizing attributes. This `TextView` is two lines of text high and 12 `ems` of text wide. The layout width and height are specified to the size of the `TextView` and are required attributes in the XML schema:

```
<TextView
    android:id="@+id/TextView04"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:lines="2"
    android:ems="12"
    android:text="@string/autolink_test" />
```

Instead of having the text only truncate at the end, as happens in the preceding example, we can enable the `ellipsize` attribute to replace the last couple characters with an ellipsis (...) so the user knows that not all text is displayed.

Creating Contextual Links in Text

If your text contains references to email addresses, web pages, phone numbers, or even street addresses, you might want to consider using the attribute `autoLink` (see Figure 7.1). The `autoLink` attribute has four values that you can use in combination with each other. When enabled, these `autoLink` attribute values create standard web-style links to the application that can act on that data type. For instance, setting the attribute to `web` automatically finds and links any URLs to web pages.

Your text can contain the following values for the `autoLink` attribute:

- **`none`**: Disables all linking.
- **`web`**: Enables linking of URLs to web pages.
- **`email`**: Enables linking of email addresses to the mail client with the recipient filled.
- **`phone`**: Enables linking of phone numbers to the dialer application with the phone number filled out, ready to be dialed.
- **`map`**: Enables linking of street addresses to the map application to show the location.
- **`all`**: Enables all types of linking.

Turning on the `autoLink` feature relies on the detection of the various types within the Android SDK. In some cases, the linking might not be correct or might be misleading.

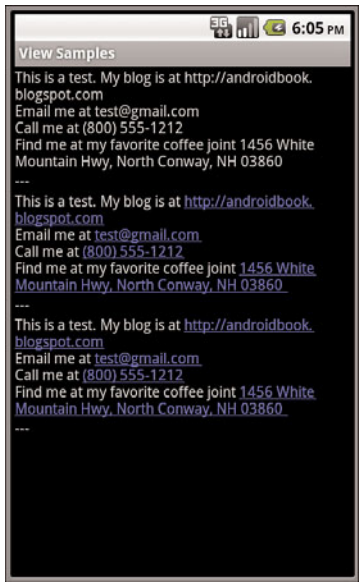

Figure 7.1 Three TextViews: Simple, AutoLink All (not clickable), and AutoLink All (clickable).

Here is an example that links email and web pages, which, in our opinion, are the most reliable and predictable:

```
<TextView
    android:id="@+id/TextView02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/autolink_test"
    android:autoLink="web|email" />
```

There are two helper values for this attribute, as well. You can set it to `none` to make sure no type of data is linked. You can also set it to `all` to have all known types linked. Figure 7.2 illustrates what happens when you click on these links. The default for a `TextView` is not to link any types. If you want the user to see the various data types highlighted but you don't want the user to click on them, you can set the `linksClickable` attribute to `false`.

Retrieving Data from Users

The Android SDK provides a number of controls for retrieving data from users. One of the most common types of data that applications often need to collect from users is text. Two frequently used controls to handle this type of job are `EditText` controls and `Spinner` controls.

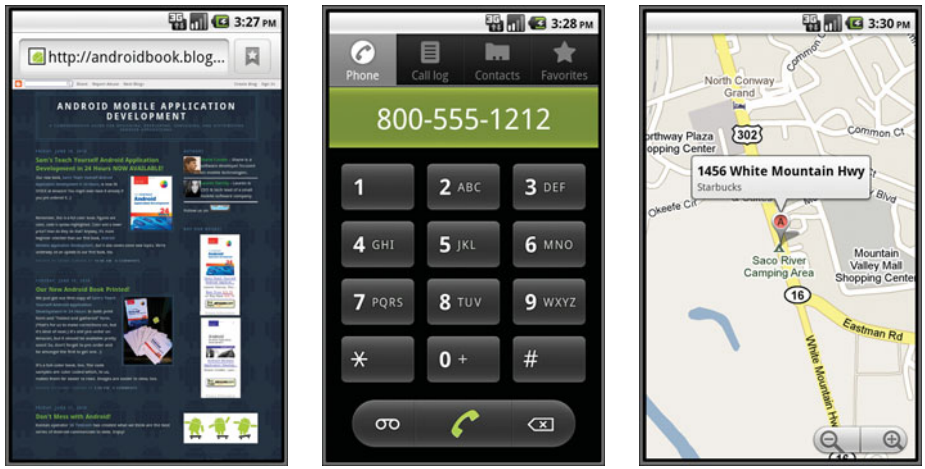

Figure 7.2 Clickable AutoLinks: URL launches browser, phone number launches dialer, and street address launches Google Maps.

Retrieving Text Input Using `EditText` Controls

The Android SDK provides a convenient control called `EditText` to handle text input from a user. The `EditText` class is derived from `TextView`. In fact, most of its functionality is contained within `TextView` but enabled when created as an `EditText`. The `EditText` object has a number of useful features enabled by default, many of which are shown in Figure 7.3.

First, though, let's see how to define an `EditText` control in an XML layout file:

```
<EditText
    android:id="@+id/EditText01"
    android:layout_height="wrap_content"
    android:hint="type here"
    android:lines="4"
    android:layout_width="fill_parent" />
```

This layout code shows a basic `EditText` element. There are a couple of interesting things to note. First, the `hint` attribute puts some text in the edit box that goes away when the user starts entering text. Essentially, this gives a hint to the user as to what should go there. Next is the `lines` attribute, which defines how many lines tall the input box is. If this is not set, the entry field grows as the user enters text. However, setting a size allows the user to scroll within a fixed sized to edit the text. This also applies to the width of the entry.

By default, the user can perform a long press to bring up a context menu. This provides to the user some basic copy, cut, and paste operations as well as the ability to change the input method and add a word to the user's dictionary of frequently used words (shown in Figure 7.4). You do not need to provide any additional code for this useful behavior to benefit your users. You can also highlight a portion of the text from code, too. A call to `setSelection()` does this, and a call to `selectAll()` highlights the entire text entry field.

Figure 7.3 Various styles of **`EditText`** controls and **`Spinner`** and **`Button`** controls.

The `EditText` object is essentially an editable `TextView`. This means that you can read text from it in the same way as you did with `TextView`: by using the `getText()` method. You can also set initial text to draw in the text entry area using the `setText()` method. This is useful when a user edits a form that already has data. Finally, you can set the `editable` attribute to `false`, so the user cannot edit the text in the field but can still copy text out of it using a long press.

Helping the User with Auto Completion

In addition to providing a basic text editor with the `EditText` control, the Android SDK also provides a way to help the user with entering commonly used data into forms. This functionality is provided through the auto-complete feature.

There are two forms of auto-complete. One is the more standard style of filling in the entire text entry based on what the user types. If the user begins typing a string that matches a word in a developer-provided list, the user can choose to complete the word with just a tap. This is done through the `AutoCompleteTextView` control (see Figure 7.5, left). The second method allows the user to enter a list of items, each of which has auto-complete functionality (see Figure 7.5, right). These items must be separated in some way by providing a `Tokenizer` to the `MultiAutoCompleteTextView` object that handles this method. A common `Tokenizer` implementation is provided for comma-separated lists and is used by specifying the `MultiAutoCompleteTextView.CommaTokenizer` object. This can be helpful for lists of specifying common tags and the like.

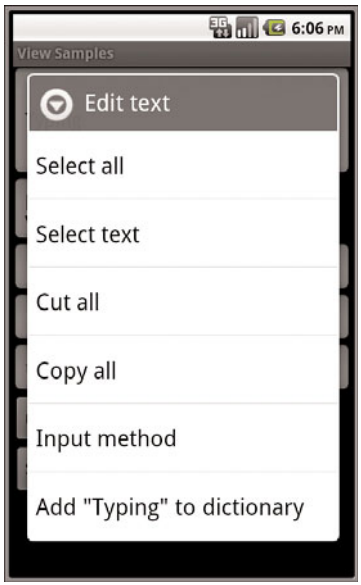

Figure 7.4 A long press on **`EditText`** controls typically launches a Context menu for Select, Cut, and Paste.

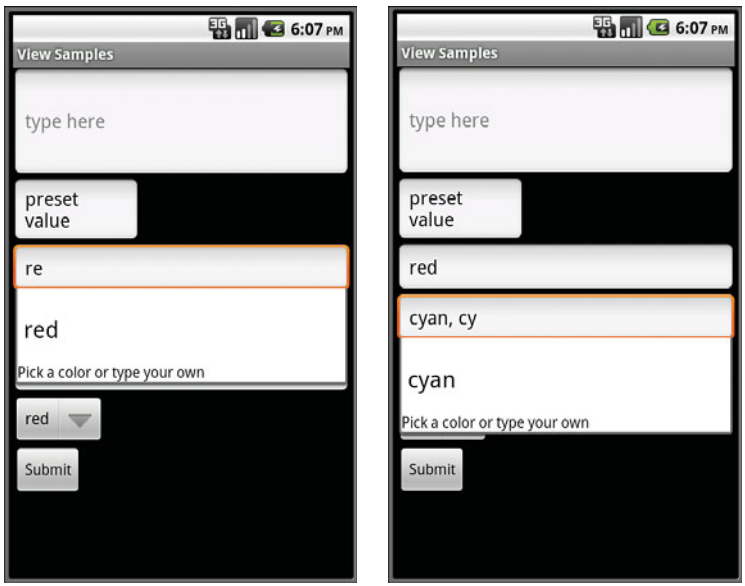

Figure 7.5 Using **`AutoCompleteTextView`** (left) and **`MultiAutoCompleteTextView`** (right).

Both of the auto-complete text editors use an adapter to get the list of text that they use to provide completions to the user. This example shows how to provide an `AutoCompleteTextView` for the user that can help them type some of the basic colors from an array in the code:

```
final String[] COLORS = {
    "red", "green", "orange", "blue", "purple",
    "black", "yellow", "cyan", "magenta" };
ArrayAdapter<String> adapter =
    new ArrayAdapter<String>(this,
        android.R.layout.simple_dropdown_item_1line,
        COLORS);
AutoCompleteTextView text = (AutoCompleteTextView)
    findViewById(R.id.AutoCompleteTextView01);
text.setAdapter(adapter);
```

In this example, when the user starts typing in the field, if he starts with one of the letters in the `COLORS` array, a drop-down list shows all the available completions. Note that this does not limit what the user can enter. The user is still free to enter any text (such as puce). The adapter controls the look of the drop-down list. In this case, we use a built-in layout made for such things. Here is the layout resource definition for this `AutoCompleteTextView` control:

```
<AutoCompleteTextView
    android:id="@+id/AutoCompleteTextView01"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:completionHint="Pick a color or type your own"
    android:completionThreshold="1" />
```

There are a couple more things to notice here. First, you can choose when the completion drop-down list shows by filling in a value for the `completionThreshold` attribute. In this case, we set it to a single character, so it displays immediately if there is a match. The default value is two characters of typing before it displays auto-completion options. Second, you can set some text in the `completionHint` attribute. This displays at the bottom of the drop-down list to help users. Finally, the drop-down list for completions is sized to the `TextView`. This means that it should be wide enough to show the completions and the text for the `completionHint` attribute.

The `MultiAutoCompleteTextView` is essentially the same as the regular auto-complete, except that you must assign a `Tokenizer` to it so that the control knows where each auto-completion should begin. The following is an example that uses the same adapter as the previous example but includes a `Tokenizer` for a list of user color responses, each separated by a comma:

```
MultiAutoCompleteTextView mtext =
    (MultiAutoCompleteTextView) findViewById(R.id.MultiAutoCompleteTextView01);
mtext.setAdapter(adapter);
mtext.setTokenizer(new MultiAutoCompleteTextView.CommaTokenizer());
```

As you can see, the only change is setting the `Tokenizer`. Here we use the built-in comma `Tokenizer` provided by the Android SDK. In this case, whenever a user chooses a color from the list, the name of the color is completed, and a comma is automatically added so that the user can immediately start typing in the next color. As before, this does not limit what the user can enter. If the user enters "maroon" and places a comma after it, the auto-completion starts again as the user types another color, regardless of the fact that it didn't help the user type in the color maroon. You can create your own `Tokenizer` by implementing the `MultiAutoCompleteTextView.Tokenizer` interface. You can do this if you'd prefer entries separated by a semicolon or some other more complex separators.

Constraining User Input with Input Filters

There are often times when you don't want the user to type just anything. Validating input after the user has entered something is one way to do this. However, a better way to avoid wasting the user's time is to filter the input. The `EditText` control provides a way to set an `InputFilter` that does only this.

The Android SDK provides some `InputFilter` objects for use. There are `InputFilter` objects that enforce such rules as allowing only uppercase text and limiting the length of the text entered. You can create custom filters by implementing the `InputFilter` interface, which contains the single method called `filter()`. Here is an example of an `EditText` control with two built-in filters that might be appropriate for a two-letter state abbreviation:

```
final EditText text_filtered =
    (EditText) findViewById(R.id.input_filtered);
text_filtered.setFilters(new InputFilter[] {
    new InputFilter.AllCaps(),
    new InputFilter.LengthFilter(2)
});
```

The `setFilters()` method call takes an array of `InputFilter` objects. This is useful for combining multiple filters, as shown. In this case, we convert all input to uppercase. Additionally, we set the maximum length to two characters long. The `EditText` control looks the same as any other, but if you try to type lowercase, the text is converted to uppercase, and the string is limited to two characters. This does not mean that all possible inputs are valid, but it does help users to not concern themselves with making the input too long or bother with the case of the input. This also helps your application by guaranteeing that any text from this input is a length of two. It does not constrain the input to only letters, though. Input filters can also be defined in XML.

Giving Users Input Choices Using `Spinner` Controls

Sometimes you want to limit the choices available for users to type. For instance, if users are going to enter the name of a state, you might as well limit them to only the valid states

because this is a known set. Although you could do this by letting them type something and then blocking invalid entries, you can also provide similar functionality with a `Spinner` control. As with the auto-complete method, the possible choices for a spinner can come from an `Adapter`. You can also set the available choices in the layout definition by using the entries attribute with an array resource (specifically a string-array that is referenced as something such as `@array/state-list`). The `Spinner` control isn't actually an `EditText`, although it is frequently used in a similar fashion. Here is an example of the XML layout definition for a `Spinner` control for choosing a color:

```
<Spinner
    android:id="@+id/Spinner01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:entries="@array/colors"
    android:prompt="@string/spin_prompt" />
```

This places a `Spinner` control on the screen (see Figure 7.6). When the user selects it, a pop-up shows the prompt text followed by a list of the possible choices. This list allows only a single item to be selected at a time, and when one is selected, the pop-up goes away.

There are a couple of things to notice here. First, the `entries` attribute is set to the values that shows by assigning it to an array resource, referred to here as `@array/colors`.

Figure 7.6 Filtering choices with a **`spinner`** control.

Note

See Chapter 6, "Managing Application Resources," for information on how to create an array resource.

Second, the `prompt` attribute is defined to a string resource. Unlike some other string attributes, this one is required to be a string resource. The prompt displays when the pop-up comes up and can be used to tell the user what kinds of values that can be selected from.

Because the `Spinner` control is not a `TextView`, but a list of `TextView` objects, you can't directly request the selected text from it. Instead, you have to retrieve the selected `View` and extract the text directly:

```
final Spinner spin = (Spinner) findViewById(R.id.Spinner01);
TextView text_sel = (TextView)spin. getSelectedView();
String selected_text = text_sel.getText();
```

As it turns out, you can request the currently selected `View` object, which happens to be a `TextView` in this case. This enables us to retrieve the text and use it directly. Alternatively, we could have called the `getSelectedItem()` or `getSelectedItemId()` methods to deal with other forms of selection.

Using Buttons, Check Boxes, and Radio Groups

Another common user interface element is the button. In this section, you learn about different kinds of buttons provided by the Android SDK. These include the basic `Button`, `ToggleButton`, `CheckBox`, and `RadioButton`. You can find examples of each button type in Figure 7.7.

A basic `Button` is often used to perform some sort of action, such as submitting a form or confirming a selection. A basic `Button` control can contain a text or image label.

A `CheckBox` is a button with two states—checked or unchecked. You often use `CheckBox` controls to turn a feature on or off or to pick multiple items from a list.

A `ToggleButton` is similar to a `CheckBox`, but you use it to visually show the state. The default behavior of a toggle is like that of a power on/off button.

A `RadioButton` provides selection of an item. Grouping `RadioButton` controls together in a container called a `RadioGroup` enables the developer to enforce that only one `RadioButton` is selected at a time.

Using Basic Buttons

The `android.widget.Button` class provides a basic button implementation in the Android SDK. Within the XML layout resources, buttons are specified using the `Button` element. The primary attribute for a basic button is the text field. This is the label that appears on the middle of the button's face. You often use basic `Button` controls for buttons with text such as "Ok," "Cancel," or "Submit."

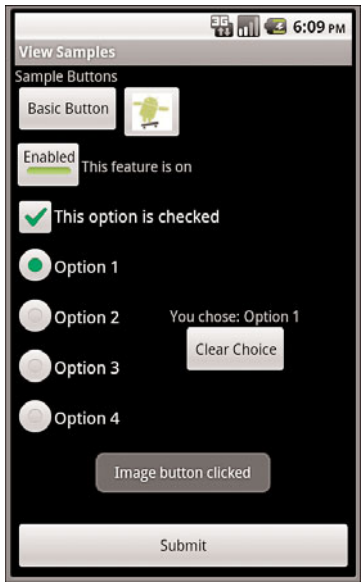

Figure 7.7 Various types of button controls.

Tip

You can find many common application string values in the Android system resource strings, exposed in `android.R.string`. There are strings for common button text such as "yes," "no," "ok," "cancel," and "copy." For more information on system resources, see Chapter 6.

The following XML layout resource file shows a typical `Button` control definition:

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

A button won't do anything, other than animate, without some code to handle the click event. Here is an example of some code that handles a click for a basic button and displays a `Toast` message on the screen:

```
setContentView(R.layout.buttons);
final Button basic_button = (Button) findViewById(R.id.basic_button);
basic_button.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
```

```
            Toast.makeText(ButtonsActivity.this,
                "Button clicked", Toast.LENGTH_SHORT).show();
        }
});
```

Tip

A `Toast` (`android.widget.Toast`) is a simple dialog-like message that displays for a second or so and then disappears. `Toast` messages are useful for providing the user with non-essential confirmation messages; they are also quite handy for debugging.

To handle the click event for when a button is pressed, we first get a reference to the `Button` by its resource identifier. Next, the `setOnClickListener()` method is called. It requires a valid instance of the class `View.OnClickListener`. A simple way to provide this is to define the instance right in the method call. This requires implementing the `onClick()` method. Within the `onClick()` method, you are free to carry out whatever actions you need. Here, we simply display a message to the users telling them that the button was, in fact, clicked.

A button with its primary label as an image is an `ImageButton`. An `ImageButton` is, for most purposes, almost exactly like a basic button. Click actions are handled in the same way. The primary difference is that you can set its `src` attribute to be an image. Here is an example of an `ImageButton` definition in an XML layout resource file:

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/image_button"
    android:src="@drawable/droid" />
```

In this case, a small `drawable` resource is referenced. Refer to Figure 7.7 to see what this "Android" button looks like. (It's to the right of the basic button.)

Using Check Boxes and Toggle Buttons

The check box button is often used in lists of items where the user can select multiple items. The Android check box contains a text attribute that appears to the side of the check box. This is used in a similar way to the label of a basic button. In fact, it's basically a `TextView` next to the button.

Here's an XML layout resource definition for a `CheckBox` control:

```
<CheckBox
    android:id="@+id/checkbox"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Check me?" />
```

You can see how this `CheckBox` is displayed in Figure 7.7 (center).

The following example shows how to check for the state of the button programmatically and change the text label to reflect the change:

```
final CheckBox check_button = (CheckBox) findViewById(R.id.checkbox);
check_button.setOnClickListener(new View.OnClickListener() {
    public void onClick (View v) {
        TextView tv = (TextView)findViewById(R.id.checkbox);
            tv.setText(check_button.isChecked() ?
                "This option is checked" :
                "This option is not checked");
    }
});
```

This is similar to the basic button. A check box automatically shows the check as enabled or disabled. This enables us to deal with behavior in our application rather than worrying about how the button should behave. The layout shows that the text starts out one way but, after the user presses the button, the text changes to one of two different things depending on the checked state. As the code shows, the `CheckBox` is also a `TextView`.

A `Toggle Button` is similar to a check box in behavior but is usually used to show or alter the on or off state of something. Like the `CheckBox`, it has a state (checked or not). Also like the check box, the act of changing what displays on the button is handled for us. Unlike the `CheckBox`, it does not show text next to it. Instead, it has two text fields. The first attribute is `textOn`, which is the text that displays on the button when its checked state is on. The second attribute is `textOff`, which is the text that displays on the button when its checked state is off. The default text for these is "ON" and "OFF," respectively.

The following layout code shows a definition for a toggle button that shows "Enabled" or "Disabled" based on the state of the button:

```
<ToggleButton
    android:id="@+id/toggle_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Toggle"
    android:textOff="Disabled"
    android:textOn="Enabled" />
```

This type of button does not actually display the value for the `text` attribute, even though it's a valid attribute to set. Here, the only purpose it serves is to demonstrate that it doesn't display. You can see what this `ToggleButton` looks like in Figure 7.7 (center).

Using `RadioGroups` and `RadioButtons`

You often use radio buttons when a user should be allowed to only select one item from a small group of items. For instance, a question asking for gender can give three options: male, female, and unspecified. Only one of these options should be checked at a time. The `RadioButton` objects are similar to `CheckBox` objects. They have a text label next to them, set via the `text` attribute, and they have a state (checked or unchecked). However, you can

group `RadioButton` objects inside a `RadioGroup` that handles enforcing their combined states so that only one `RadioButton` can be checked at a time. If the user selects a `RadioButton` that is already checked, it does not become unchecked. However, you can provide the user with an action to clear the state of the entire `RadioGroup` so that none of the buttons are checked.

Here we have an XML layout resource with a `RadioGroup` containing four `RadioButton` objects (shown in Figure 7.7, toward the bottom of the screen). The `RadioButton` objects have text labels, "Option 1," and so on. The XML layout resource definition is shown here:

```
<RadioGroup
    android:id="@+id/RadioGroup01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <RadioButton
        android:id="@+id/RadioButton01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 1"></RadioButton>
    <RadioButton
        android:id="@+id/RadioButton02"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 2"></RadioButton>
    <RadioButton
        android:id="@+id/RadioButton03"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 3"></RadioButton>
    <RadioButton
        android:id="@+id/RadioButton04"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Option 4"></RadioButton>
</RadioGroup>
```

You handle actions on these `RadioButton` objects through the `RadioGroup` object. The following example shows registering for clicks on the `RadioButton` objects within the `RadioGroup`:

```
final RadioGroup group = (RadioGroup)findViewById(R.id.RadioGroup01);
final TextView tv = (TextView)
    findViewById(R.id.TextView01);

group.setOnCheckedChangeListener(new
    RadioGroup.OnCheckedChangeListener() {
        public void onCheckedChanged(
```

```
            RadioGroup group, int checkedId) {
            if (checkedId != -1) {
                RadioButton rb = (RadioButton)
                    findViewById(checkedId);
                if (rb != null) {
                    tv.setText("You chose: " + rb.getText());
                }
            } else {
                tv.setText("Choose 1");
            }
        }
});
```

As this layout example demonstrates, there is nothing special that you need to do to make the `RadioGroup` and internal `RadioButton` objects work properly. The preceding code illustrates how to register to receive a notification whenever the `RadioButton` selection changes.

The code demonstrates that the notification contains the resource identifier for the specific `RadioButton` that the user chose, as assigned in the layout file. To do something interesting with this, you need to provide a mapping between this resource identifier (or the text label) to the corresponding functionality in your code. In the example, we query for the button that was selected, get its text, and assign its text to another `TextView` control that we have on the screen.

As mentioned, the entire `RadioGroup` can be cleared so that none of the `RadioButton` objects are selected. The following example demonstrates how to do this in response to a button click outside of the `RadioGroup`:

```
final Button clear_choice = (Button) findViewById(R.id.Button01);
clear_choice.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        RadioGroup group = (RadioGroup)
            findViewById(R.id.RadioGroup01);
        if (group != null) {
            group.clearCheck();
        }
    }
}
```

The action of calling the `clearCheck()` method triggers a call to the `onCheckedChangedListener()` callback method. This is why we have to make sure that the resource identifier we received is valid. Right after a call to the `clearCheck()` method, it is not a valid identifier but instead is set to the value `-1` to indicate that no `RadioButton` is currently checked.