8

Designing User Interfaces with Layouts

In this chapter, we discuss how to design user interfaces for Android applications. Here we focus on the various layout controls you can use to organize screen elements in different ways. We also cover some of the more complex `View` objects we call container views. These are `View` objects that can contain other `View` objects and controls.

Creating User Interfaces in Android

Application user interfaces can be simple or complex, involving many different screens or only a few. Layouts and user interface controls can be defined as application resources or created programmatically at runtime.

Creating Layouts Using XML Resources

As discussed in previous chapters, Android provides a simple way to create layout files in XML as resources provided in the `/res/layout` project directory. This is the most common and convenient way to build Android user interfaces and is especially useful for defining static screen elements and control properties that you know in advance, and to set default attributes that you can modify programmatically at runtime.

Warning

The Eclipse layout resource designer can be a helpful tool for designing and previewing layout resources. However, the preview can't replicate exactly how the layout appears to end users. For this, you must test your application on a properly configured emulator and, more importantly, on your target devices.

You can configure almost any `ViewGroup` or `View` (or `View` subclass) attribute using the XML layout resource files. This method greatly simplifies the user interface design process, moving much of the static creation and layout of user interface controls, and basic definition of control attributes, to the XML, instead of littering the code. Developers reserve the

ability to alter these layouts programmatically as necessary, but they can set all the defaults in the XML template.

You'll recognize the following as a simple layout file with a `LinearLayout` and a single `TextView` control. This is the default layout file provided with any new Android project in Eclipse, referred to as `/res/layout/main.xml`:

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

This block of XML shows a basic layout with a single `TextView`. The first line, which you might recognize from most XML files, is required. Because it's common across all the files, we do not show it in any other examples.

Next, we have the `LinearLayout` element. `LinearLayout` is a `ViewGroup` that shows each child `View` either in a single column or in a single row. When applied to a full screen, it merely means that each child `View` is drawn under the previous `View` if the orientation is set to vertical or to the right of the previous `View` if orientation is set to horizontal.

Finally, there is a single child `View`—in this case, a `TextView`. A `TextView` is a control, which is also a `View`. A `TextView` draws text on the screen. In this case, it draws the text defined in the "`@string/hello`" string resource.

Creating only an XML file, though, won't actually draw anything on the screen. A particular layout is usually associated with a particular `Activity`. In your default Android project, there is only one activity, which sets the `main.xml` layout by default. To associate the `main.xml` layout with the activity, use the method call `setContentView()` with the identifier of the `main.xml` layout. The ID of the layout matches the XML filename without the extension. In this case, the preceding example came from `main.xml`, so the identifier of this layout is simply `main`:

```
setContentView(R.layout.main);
```

Tip

Although it's a tad confusing, the term *layout* is used for two different (but related) purposes in Android development.

In terms of resources, the /res/layout directory contains XML resource definitions often called layout resource files. These XML files provide a template for how to draw to a screen; layout resource files may contain any number of views. We talk about layout resources in Chapter 6, "Managing Application Resources."

> The term *layout* is also used to refer to a set of `ViewGroup` classes such as `LinearLayout`, `FrameLayout`, `TableLayout`, and `RelativeLayout`. These layout classes are used to organize `View` controls. We talk more about these classes later in this chapter.
>
> Therefore, you could have one or more *layouts* (such as a `LinearLayout` with two child controls—a `TextView` and an `ImageView`) defined within a *layout resource file*, such as `/res/layout/myScreen.xml`.

Creating Layouts Programmatically

You can create user interface components such as layouts at runtime programmatically, but for organization and maintainability, it's best that you leave this for the odd case rather than the norm. The main reason is because the creation of layouts programmatically is onerous and difficult to maintain, whereas the XML resource method is visual, more organized, and could be done by a separate designer with no Java skills.

> **Tip**
>
> The code examples provided in this section are taken from the SameLayout application. This source code for the SameLayout application is provided for download on the book website.

The following example shows how to programmatically have an `Activity` instantiate a `LinearLayout` view and place two `TextView` objects within it. No resources whatsoever are used; actions are done at runtime instead.

```
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    TextView text1 = new TextView(this);
    text1.setText("Hi there!");

    TextView text2 = new TextView(this);
    text2.setText("I'm second. I need to wrap.");
    text2.setTextSize((float) 60);

    LinearLayout ll = new LinearLayout(this);
    ll.setOrientation(LinearLayout.VERTICAL);
    ll.addView(text1);
    ll.addView(text2);

    setContentView(ll);
}
```

The `onCreate()` method is called when the `Activity` is created. The first thing this method does is some normal `Activity` housekeeping by calling the constructor for the base class.

Next, two `TextView` controls are instantiated. The `Text` property of each `TextView` is set using the `setText()` method. All `TextView` attributes, such as `TextSize`, are set by

making method calls on the `TextView` object. These actions perform the same function that you have in the past by setting the properties `Text` and `TextSize` using the Eclipse layout resource designer, except these properties are set at runtime instead of defined in the layout files compiled into your application package.

Tip

The XML property name is usually similar to the method calls for getting and setting that same control property programmatically. For instance, `android:visibility` maps to the methods `setVisibility()` and `getVisibility()`. In the preceding example `TextView`, the methods for getting and setting the `TextSize` property are `getTextSize()` and `setTextSize()`.

To display the `TextView` objects appropriately, we need to encapsulate them within a container of some sort (a layout). In this case, we use a `LinearLayout` with the orientation set to `VERTICAL` so that the second `TextView` begins beneath the first, each aligned to the left of the screen. The two `TextView` controls are added to the `LinearLayout` in the order we want them to display.

Finally, we call the `setContentView()` method, part of your `Activity` class, to draw the `LinearLayout` and its contents on the screen.

As you can see, the code can rapidly grow in size as you add more `View` controls and you need more attributes for each `View`. Here is that same layout, now in an XML layout file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TextView
    android:id="@+id/TextView1"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Hi There!"
    />
<TextView
    android:id="@+id/TextView2"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:textSize="60px"
    android:text="I'm second. I need to wrap."
    />
</LinearLayout>
```

You might notice that this isn't a literal translation of the code example from the previous section, although the output is identical, as shown in Figure 8.1.

Figure 8.1 Two different methods to create a screen have the same result.

First, in the XML layout files, `layout_width` and `layout_height` are required attributes. Next, you see that each `TextView` object has a unique `id` property assigned so that it can be accessed programmatically at runtime. Finally, the `textSize` property needs to have its units defined. The XML attribute takes a `dimension` type (as described in Chapter 6) instead of a float.

The end result differs only slightly from the programmatic method. However, it's far easier to read and maintain. Now you need only one line of code to display this layout view. Again, if the layout resource is stored in the `/res/layout/resource_based_layout.xml` file, that is

```
setContentView(R.layout.resource_based_layout);
```

Organizing Your User Interface

In Chapter 7, "Exploring User Interface Screen Elements," we talk about how the class `View` is the building block for user interfaces in Android. All user interface controls, such as `Button`, `Spinner`, and `EditText`, derive from the `View` class.

Now we talk about a special kind of `View` called a `ViewGroup`. The classes derived from `ViewGroup` enable developers to display `View` objects (including all the user interface controls you learn about in Chapter 7) on the screen in an organized fashion.

Understanding `View` versus `ViewGroup`

Like other `View` objects, including the controls from Chapter 7, `ViewGroup` controls represent a rectangle of screen space. What makes `ViewGroup` different from your typical control is that `ViewGroup` objects contain other `View` objects. A `View` that contains other `View` objects is called a *parent view*. The parent `View` contains `View` objects called *child views*, or *children*.

You add child `View` objects to a `ViewGroup` programmatically using the method `addView()`. In XML, you add child objects to a `ViewGroup` by defining the child `View` control as a child node in the XML (within the parent XML element, as we've seen various times using the `LinearLayout ViewGroup`).

`ViewGroup` subclasses are broken down into two categories:

- Layout classes
- View container controls

The Android SDK also provides the Hierarchy Viewer tool to help visualize the layouts you design, as discussed later in this chapter.

Using `ViewGroup` Subclasses for Layout Design

Many important subclasses of `ViewGroup` used for screen design end with the word "Layout;" for example, `LinearLayout`, `RelativeLayout`, `TableLayout`, and `FrameLayout`. You can use each of these layout classes to position groups of `View` objects (controls) on the screen in different ways. For example, we've been using the `LinearLayout` to arrange various `TextView` and `EditText` controls on the screen in a single vertical column. We could have used an `AbsoluteLayout` to specify the exact x/y coordinate locations of each control on the screen instead, but this is not easily portable across many screen resolutions. Users do not generally interact with the `Layout` objects directly. Instead, they interact with the `View` objects they contain.

Using `ViewGroup` Subclasses as View Containers

The second category of `ViewGroup` subclasses is the indirect subclasses. These special `View` objects act as `View` containers like `Layout` objects do, but they also provide some kind of functionality that enables users to interact with them like normal controls. Unfortunately, these classes are not known by any handy names; instead they are named for the kind of functionality they provide.

Some classes that fall into this category include `Gallery`, `GridView`, `ImageSwitcher`, `ScrollView`, `TabHost`, and `ListView`. It can be helpful to consider these objects as different kinds of `View` browsers. A `ListView` displays each `View` as a list item, and the user can browse between the individual `View` objects using vertical scrolling capability. A `Gallery` is a horizontal scrolling list of `View` objects with a center "current" item; the user can browse the View objects in the `Gallery` by scrolling left and right. A `TabHost` is a more complex `View` container, where each Tab can contain a `View`, and the user chooses the tab by name to see the `View` contents.

Using the Hierarchy Viewer Tool

In addition to the Eclipse layout resource designer provided with the Android plug-in, the Android Software Development Kit (SDK) provides a user interface tool called the Hierarchy Viewer. You can find the Hierarchy Viewer in the Android SDK subdirectory called `/tools`.

The Hierarchy Viewer is a visual tool that enables you to inspect your Android application's `View` objects and their parent-child relationships. You can drill down on specific `View` objects and inspect individual `View` properties at runtime. You can even save screenshots of the current application state on the emulator or the device, although this feature is somewhat unreliable.

Do the following to launch the Hierarchy Viewer with your application in the emulator:

1. Launch your Android application in the emulator.
2. Navigate to the Android SDK `/tools` directory and launch the Hierarchy Viewer.
3. Choose your emulator instance from the Device listing.
4. Select the application you want to view from the windows available. For example, to load an application from this book, choose one such as the ParisView project from Chapter 6.
5. Click Load View Hierarchy button on the menu bar.

By default, the Hierarchy Viewer loads the Layout View of your application. This includes the parent-child view relationships shown as a Tree View. In addition, a property pane shows the various properties for each View node in the tree when they are selected. A wire-frame model of the `View` objects on the screen is shown and a red box highlights the currently selected view, which correlates to the same location on the screen.

Tip

You'll have better luck navigating your application `View` objects with the Hierarchy Viewer tool if you set your `View` object `id` properties to friendly names you can remember instead of the auto-generated sequential `id` tags provided by default. For example, a `Button` control called `SubmitButton` is more descriptive than `Button01`.

Figure 8.2 shows the Hierarchy Viewer loaded with the ParisView project from Chapter 6, which was a one-screen application with a single `LinearLayout` with a `TextView` and an `ImageView` child control within it, all encapsulated within a `ScrollView` control (for scrolling ability). The bulk of the application is shown in the right sub-tree, starting with `LinearLayout` with the identifier `ParisViewLayout`. The other sub-tree is the Application title bar. A simple double-click on each child node opens that `View` object individually in its own window.

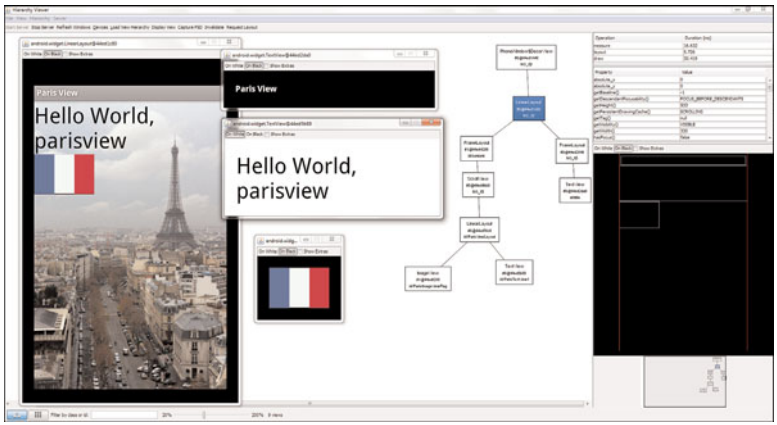

Figure 8.2 The ParisView application, shown in the Hierarchy Viewer tool (Layout View).

Each `View` can be separately displayed in its own window by selecting the appropriate `View` in the tree and choosing the Display View button on the menu bar. In Figure 8.2, you can also see that Display View is enabled on each of the child nodes: the `ImageView` with the flag, the `TextView` with the text, as well as the `LinearLayout` parent node (which includes its children), and lastly the application title bar.

You can use the Pixel Perfect view to closely inspect your application using a loupe (see Figure 8.3). You can also load PNG mockup files to overlay your user interface and adjust your application's look. You can access the Pixel Perfect view by clicking the button with the nine pixels on it at the bottom left of the Hierarchy Viewer. Click the button with the three boxes depicting the Layout view to return.

The Hierarchy Viewer tool is invaluable for debugging drawing issues related to `View` controls. If you wonder why something isn't drawing or if a `View` is even available, try launching the Hierarchy Viewer and checking that problem `View` objects' properties.

You can use the Hierarchy Viewer tool to interact and debug your application user interface. Specifically, developers can use the Invalidate and Request Layout buttons on the menu bar that correspond to `View.invalidate()` and `View.requestLayout()` functions of the UI thread. These functions initiate `View` objects and draw or redraw them as necessary upon events.

Finally, you can also use the Hierarchy Viewer to deconstruct how other applications (especially sample applications) have handled their layout and displays. This can be helpful if you'd like to re-create a layout similar to another application, especially if it uses stock `View` types. However, you can also run across `View` types not provided in the SDK, and you need to implement those custom classes for yourself.

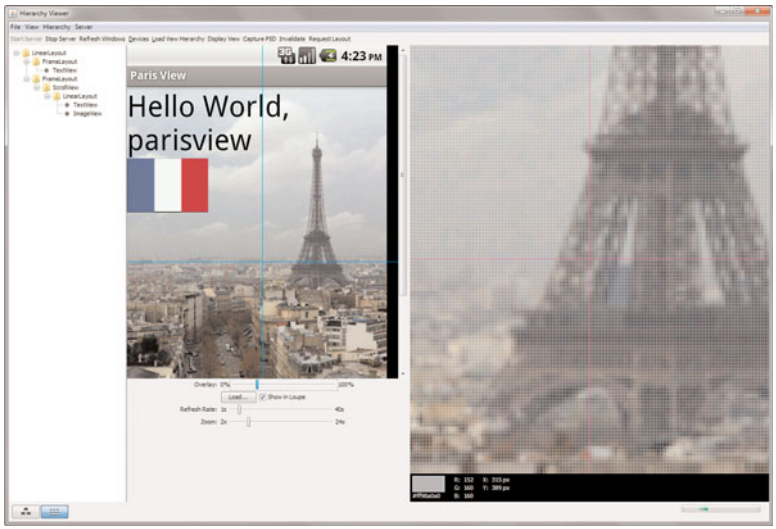

Figure 8.3 The ParisView application, shown in the Hierarchy Viewer tool (Pixel Perfect View).

Using Built-In Layout Classes

We talked a lot about the `LinearLayout` layout, but there are several other types of layouts. Each layout has a different purpose and order in which it displays its child `View` controls on the screen. Layouts are derived from `android.view.ViewGroup`.

The types of layouts built-in to the Android SDK framework include

- `FrameLayout`
- `LinearLayout`
- `RelativeLayout`
- `TableLayout`

Tip

Many of the code examples provided in this section are taken from the SimpleLayout application. This source code for the SimpleLayout application is provided for download on the book website.

All layouts, regardless of their type, have basic layout attributes. Layout attributes apply to any child `View` within that layout. You can set layout attributes at runtime programmatically, but ideally you set them in the XML layout files using the following syntax:

```
android:layout_attribute_name="value"
```

There are several layout attributes that all `ViewGroup` objects share. These include size attributes and margin attributes. You can find basic layout attributes in the

`ViewGroup.LayoutParams` class. The margin attributes enable each child `View` within a layout to have padding on each side. Find these attributes in the `ViewGroup.MarginLayoutParams` classes. There are also a number of `ViewGroup` attributes for handling child `View` drawing bounds and animation settings.

Some of the important attributes shared by all `ViewGroup` subtypes are shown in Table 8.1.

Table 8.1 **Important `ViewGroup` Attributes**

Attribute Name	Applies To	Description	Value
`android:layout_height`	Parent view Child view	Height of the view. Required attribute for child view controls in layouts.	Specific dimension value, `fill_parent`, or `wrap_content`. The `match_parent` option is available in API Level 8+.
`android:layout_width`	Parent view Child view	Width of the view. Required attribute for child view controls in layouts.	Specific dimension value, `fill_parent`, or `wrap_content`. The `match_parent` option is available in API Level 8+.
`android:layout_margin`	Child view	Extra space on all sides of the view.	Specific dimension value.

Here's an XML layout resource example of a `LinearLayout` set to the size of the screen, containing one `TextView` that is set to its full height and the width of the `LinearLayout` (and therefore the screen):

```
<LinearLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:id="@+id/TextView01"
        android:layout_height="fill_parent"
        android:layout_width="fill_parent" />
</LinearLayout>
```

Here is an example of a `Button` object with some margins set via XML used in a layout resource file:

```
<Button
    android:id="@+id/Button01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Press Me"
    android:layout_marginRight="20px"
    android:layout_marginTop="60px" />
```

Remember that layout elements can cover any rectangular space on the screen; it doesn't need to be the entire screen. Layouts can be nested within one another. This provides great flexibility when developers need to organize screen elements. It is common to start with a `FrameLayout` or `LinearLayout` (as you've seen in many of the Chapter 7 examples) as the parent layout for the entire screen and then organize individual screen elements inside the parent layout using whichever layout type is most appropriate.

Now let's talk about each of the common layout types individually and how they differ from one another.

Using `FrameLayout`

A `FrameLayout` view is designed to display a stack of child `View` items. You can add multiple views to this layout, but each `View` is drawn from the top-left corner of the layout. You can use this to show multiple images within the same region, as shown in Figure 8.4, and the layout is sized to the largest child `View` in the stack.

You can find the layout attributes available for `FrameLayout` child `View` objects in `android.control.FrameLayout.LayoutParams`. Table 8.2 describes some of the important attributes specific to `FrameLayout` views.

Table 8.2 **Important `FrameLayout View` Attributes**

Attribute Name	Applies To	Description	Value
`android:foreground`	Parent view	Drawable to draw over the content.	Drawable resource.
`android:foregroundGravity`	Parent view	Gravity of foreground drawable.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.

Table 8.2 **Continued**

Attribute Name	Applies To	Description	Value
`android:measureAllChildren`	Parent view	Restrict size of layout to all child views or just the child views set to `VISIBLE` (and not those set to `INVISIBLE`).	True or false.
`android:layout_gravity`	Child view	A gravity constant that describes how to place the child `View` within the parent.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.

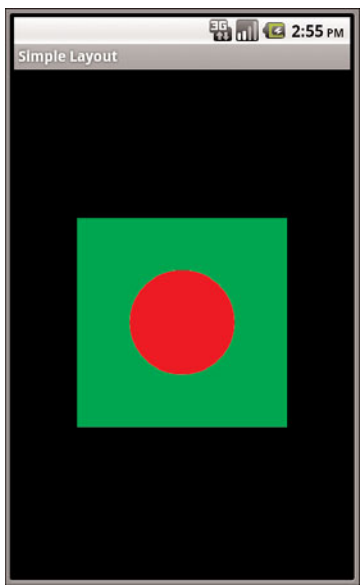

Figure 8.4 An example of **`FrameLayout`** usage.

Here's an example of an XML layout resource with a `FrameLayout` and two child `View` objects, both `ImageView` objects. The green rectangle is drawn first and the red oval is drawn on top of it. The green rectangle is larger, so it defines the bounds of the `FrameLayout`:

```
<FrameLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
```

```
    android:id="@+id/FrameLayout01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center">
    <ImageView
        android:id="@+id/ImageView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/green_rect"
        android:minHeight="200px"
        android:minWidth="200px" />
    <ImageView
        android:id="@+id/ImageView02"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/red_oval"
        android:minHeight="100px"
        android:minWidth="100px"
        android:layout_gravity="center" />
</FrameLayout>
```

Using `LinearLayout`

A `LinearLayout` view organizes its child `View` objects in a single row, shown in Figure 8.5, or column, depending on whether its orientation attribute is set to horizontal or vertical. This is a very handy layout method for creating forms.

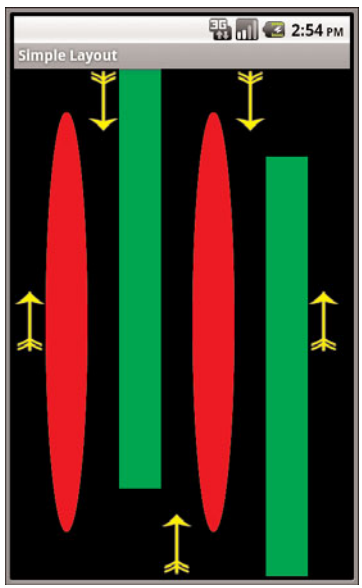

Figure 8.5 An example of **`LinearLayout`** (horizontal orientation).

You can find the layout attributes available for `LinearLayout` child `View` objects in `android.control.LinearLayout.LayoutParams`. Table 8.3 describes some of the important attributes specific to `LinearLayout` views.

Table 8.3 **Important `LinearLayout` View Attributes**

Attribute Name	Applies To	Description	Value
`android:orientation`	Parent view	Layout is a single row (horizontal) or single column (vertical).	Horizontal or vertical.
`android:gravity`	Parent view	Gravity of child views within layout.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.
`android:layout_gravity`	Child view	The gravity for a specific child view. Used for positioning of views.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.
`android:layout_weight`	Child view	The weight for a specific child view. Used to provide ratio of screen space used within the parent control.	The sum of values across all child views in a parent view must equal `1`. For example, one child control might have a value of `.3` and another have a value of `.7`.

Using `RelativeLayout`

The `RelativeLayout` view enables you to specify where the `child view` controls are in relation to each other. For instance, you can set a child `View` to be positioned "above" or "below" or "to the left of" or "to the right of" another `View`, referred to by its unique identifier. You can also align child `View` objects relative to one another or the parent layout edges. Combining `RelativeLayout` attributes can simplify creating interesting user interfaces without resorting to multiple layout groups to achieve a desired effect. Figure 8.6 shows how each of the button controls is relative to each other.

You can find the layout attributes available for `RelativeLayout` child `View` objects in `android.control.RelativeLayout.LayoutParams`. Table 8.4 describes some of the important attributes specific to `RelativeLayout` views.

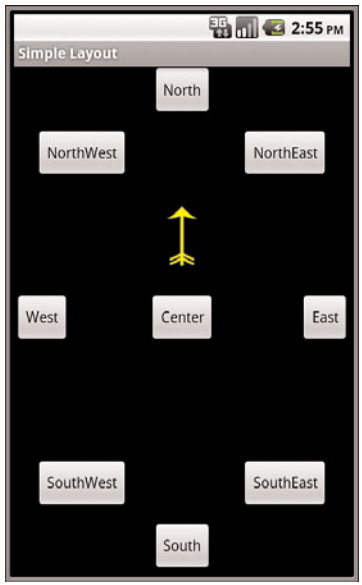

Figure 8.6 An example of **`RelativeLayout`** usage.

Table 8.4 **Important `RelativeLayout` View Attributes**

Attribute Name	Applies To	Description	Value
`android:gravity`	Parent view	Gravity of child views within layout.	One or more constants separated by "|". The constants available are `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`.
`android:layout_centerInParent`	Child view	Centers child view horizontally and vertically within parent view.	True or false.

Table 8.4 **Continued**

Attribute Name	Applies To	Description	Value
`android:layout_centerHorizontal`	Child view	Centers child view horizontally within parent view.	True or false.
`android:layout_centerVertical`	Child view	Centers child view vertically within parent view.	True or false.
`android:layout_alignParentTop`	Child view	Aligns child view with top edge of parent view.	True or false.
`android:layout_alignParentBottom`	Child view	Aligns child view with bottom edge of parent view.	True or false.
`android:layout_alignParentLeft`	Child view	Aligns child view with left edge of parent view.	True or false.
`android:layout_alignParentRight`	Child view	Aligns child view with right edge of parent view.	True or false.
`android:layout_alignRight`	Child view	Aligns child view with right edge of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_alignLeft`	Child view	Aligns child view with left edge of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_alignTop`	Child view	Aligns child view with top edge of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_alignBottom`	Child view	Aligns child view with bottom edge of another child view, specified by ID.	A view ID; for example, `@id/Button1`

Table 8.4 Continued: `RelativeLayout` View Attributes

Attribute Name	Applies To	Description	Value
`android:layout_above`	Child view	Positions bottom edge of child view above another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_below`	Child view	Positions top edge of child view below another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_toLeftOf`	Child view	Positions right edge of child view to the left of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`android:layout_toRightOf`	Child view	Positions left edge of child view to the right of another child view, specified by ID.	A view ID; for example, `@id/Button1`

Here's an example of an XML layout resource with a `RelativeLayout` and two child `View` objects, a `Button` object aligned relative to its parent, and an `ImageView` aligned and positioned relative to the `Button` (and the parent):

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/RelativeLayout01"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent">
    <Button
        android:id="@+id/ButtonCenter"
        android:text="Center"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true" />
    <ImageView
        android:id="@+id/ImageView01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_above="@id/ButtonCenter"
        android:layout_centerHorizontal="true"
        android:src="@drawable/arrow" />
</RelativeLayout>
```

Warning

The `AbsoluteLayout` class has been deprecated. `AbsoluteLayout` uses specific x and y coordinates for child view placement. This layout can be useful when pixel-perfect placement is required. However, it's less flexible because it does not adapt well to other device configurations with different screen sizes and resolutions. Under most circumstances, other popular layout types such as `FrameLayout` and `RelativeLayout` suffice in place of `AbsoluteLayout`, so we encourage you to use these layouts instead when possible.

Using `TableLayout`

A `TableLayout` view organizes children into rows, as shown in Figure 8.7. You add individual `View` objects within each row of the table using a `TableRow` layout `View` (which is basically a horizontally oriented `LinearLayout`) for each row of the table. Each column of the `TableRow` can contain one `View` (or layout with child `View` objects). You place `View` items added to a `TableRow` in columns in the order they are added. You can specify the column number (zero-based) to skip columns as necessary (the bottom row shown in Figure 8.7 demonstrates this); otherwise, the `View` object is put in the next column to the right. Columns scale to the size of the largest `View` of that column. You can also include normal `View` objects instead of `TableRow` elements, if you want the `View` to take up an entire row.

Figure 8.7 An example of **`TableLayout`** usage.

You can find the layout attributes available for `TableLayout` child `View` objects in `android.control.TableLayout.LayoutParams`. You can find the layout attributes available for `TableRow` child `View` objects in `android.control.TableRow.LayoutParams`. Table 8.5 describes some of the important attributes specific to `TableLayout View` objects.

Table 8.5 **Important `TableLayout` and `TableRow` View Attributes**

Attribute Name	Applies To	Description	Value
`android: collapseColumns`	`TableLayout`	A comma-delimited list of column indices to collapse (0-based)	String or string resource. For example, "`0,1,3,5`"
`android: shrinkColumns`	`TableLayout`	A comma-delimited list of column indices to shrink (0-based)	String or string resource. Use "`*`" for all columns. For example, "`0,1,3,5`"
`andriod: stretchColumns`	`TableLayout`	A comma-delimited list of column indices to stretch (0-based)	String or string resource. Use "`*`" for all columns. For example, "`0,1,3,5`"
`android: layout_column`	`TableRow` child view	Index of column this child view should be displayed in (0-based)	Integer or integer resource. For example, `1`
`android: layout_span`	`TableRow` child view	Number of columns this child view should span across	Integer or integer resource greater than or equal to `1`. For example, `3`

Here's an example of an XML layout resource with a `TableLayout` with two rows (two `TableRow` child objects). The `TableLayout` is set to stretch the columns to the size of the screen width. The first `TableRow` has three columns; each cell has a `Button` object. The second `TableRow` puts only one `Button` view into the second column explicitly:

```
<TableLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/TableLayout01"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:stretchColumns="*">
    <TableRow
        android:id="@+id/TableRow01">
        <Button
            android:id="@+id/ButtonLeft"
```

```
            android:text="Left Door" />
        <Button
            android:id="@+id/ButtonMiddle"
            android:text="Middle Door" />
        <Button
            android:id="@+id/ButtonRight"
            android:text="Right Door" />
    </TableRow>
    <TableRow
        android:id="@+id/TableRow02">
        <Button
            android:id="@+id/ButtonBack"
            android:text="Go Back"
            android:layout_column="1" />
    </TableRow>
</TableLayout>
```

Using Multiple Layouts on a Screen

Combining different layout methods on a single screen can create complex layouts. Remember that because a layout contains `View` objects and is, itself, a `View`, it can contain other layouts. Figure 8.8 demonstrates a combination of layout views used in conjunction to create a more complex and interesting screen.

> **Warning**
>
> Keep in mind that individual screens of mobile applications should remain sleek and relatively simple. This is not just because this design results in a more positive user experience; cluttering your screens with complex (and deep) View hierarchies can lead to performance problems. Use the Hierarchy Viewer to inspect your application layouts; you can also use the `layoutopt` command-line tool to help optimize your layouts and identify unnecessary components. This tool often helps identify opportunities to use layout optimization techniques, such as the `<merge>` and `<include>` tags.

Using Built-In `View` Container Classes

Layouts are not the only controls that can contain other `View` objects. Although layouts are useful for positioning other `View` objects on the screen, they aren't interactive. Now let's talk about the other kind of `ViewGroup`: the containers. These `View` objects encapsulate other, simpler `View` types and give the user some interactive ability to browse the child

View objects in a standard fashion. Much like layouts, these controls each have a special, well-defined purpose.

Figure 8.8 An example of multiple layouts used together.

The types of ViewGroup containers built-in to the Android SDK framework include

- Lists, grids, and galleries
- Switchers with ViewFlipper, ImageSwitcher, and TextSwitcher
- Tabs with TabHost and TabControl
- Scrolling with ScrollView and HorizontalScrollView
- Hiding and showing content with the SlidingDrawer

Tip

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