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` controls we call container views. These are `View` controls that can contain other `View` 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.

Although it's a bit confusing, the term *layout* is used for two different but related purposes in Android user interface design:

- 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 controls on the screen; layout resource files may contain any number of controls.
- The term is also used to refer to a set of `ViewGroup` classes, such as `LinearLayout`, `FrameLayout`, `TableLayout`, `RelativeLayout`, and `GridLayout`. These controls are used to organize other `View` controls. We talk more about these classes later in this text.

Creating Layouts Using XML Resources

Android provides a simple way to create layout resource files in XML. These resources are stored in the `/res/layout` project directory hierarchy. This is the most common and convenient way to build Android user interfaces and is especially useful for defining screen elements and default control properties that you know about at compile time.

The source code that accompanies this chapter is available for download on the publisher website: www.informit.com/title/9780321813831.

These layout resources are then used much like templates. They are loaded with default attributes that you can modify programmatically at runtime.

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

This block of XML shows a basic layout with a single `TextView` control. The first line, which you might recognize from most XML files, is required with the android layout namespace, as shown. 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 the orientation is set to horizontal.

Finally, there is a single child `View`—in this case, a `TextView`. A `TextView` is a control that 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);
```

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.

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 resources are visual, more organized, and could be used by a separate designer with no Java skills.

Tip

The code examples provided in this section are taken from the SameLayout application. The source code for the SameLayout application is provided for download on the book's websites.

The following example shows how to programmatically have an `Activity` instantiate a `LinearLayout` and place two `TextView` controls within it as child controls. The same two string resources are used for the contents of the controls; these actions are done at runtime instead.

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

    TextView text1 = new TextView(this);
    text1.setText(R.string.string1);

    TextView text2 = new TextView(this);
    text2.setText(R.string.string2);
    text2.setTextSize(TypedValue.COMPLEX_UNIT_SP, 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 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` control. 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 sample `TextView`, the methods for getting and setting the `TextSize` property are `getTextSize()` and `setTextSize()`.

To display the `TextView` controls 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="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/TextView1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/string1" />
    <TextView
        android:id="@+id/TextView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:textSize="60sp"
        android:text="@string/string2" />
</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 1.

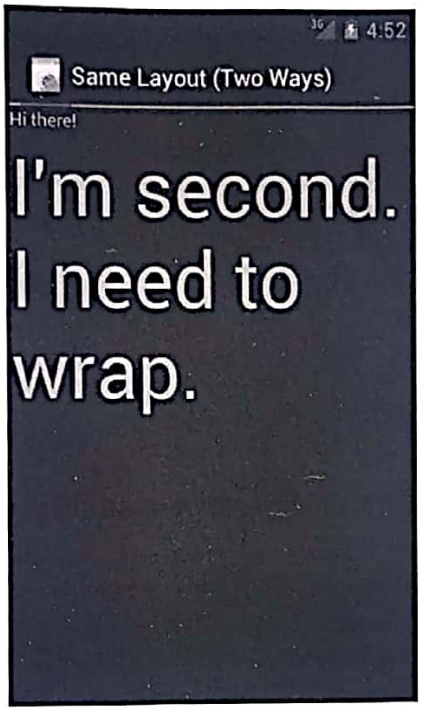


Figure 1 Two different methods to create a screen have the same result.

First, in the XML layout files, `layout_width` and `layout_height` are required attributes. Next, you see that each `TextView` control 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.

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, the layout resource is stored in the `/res/layout/resource_based_layout.xml` file:

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

Organizing Your User Interface

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` controls such as `TextView` and `Button` controls on the screen in an organized fashion.

It's important to understand the difference between `View` and `ViewGroup`. Like other `View` controls, `ViewGroup` controls represent a rectangle of screen space. What makes `ViewGroup` different from your typical control is that `ViewGroup` objects contain other `View` controls. A `View` that contains other `View` controls is called a *parent view.* The parent `View` contains `View` controls called *child views,* or *children.*

You add child `View` controls 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

Using `ViewGroup` Subclasses for Layout Design

Many of the most important subclasses of `ViewGroup` used for screen design end with "Layout." For example, the most common layout classes are `LinearLayout`, `RelativeLayout`, `TableLayout`, and `FrameLayout`. You can use each of these classes to position other `View` 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. Users do not generally interact with the layouts directly. Instead, they interact with the `View` controls they contain.

Using `ViewGroup` Subclasses as View Containers

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

Some of the 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, or container classes. A `ListView` displays each `View` control as a list item, and the user can browse between the individual

controls using vertical scrolling capability. A `Gallery` is a horizontal scrolling list of `View` controls with a center "current" item; the user can browse the `View` controls in the `Gallery` by scrolling left and right. A `TabHost` is a more complex `View` container, where each tab can contain a `View` (such as a layout) and the user selects a tab to see its contents.

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 the following:

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

Tip

Many of the code examples provided in this section are taken from the SimpleLayout application. The source code for the SimpleLayout application is provided for download on the book's websites.

All layouts, regardless of their type, have basic layout attributes. Layout attributes apply to any child `View` control 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 1.

Table 1 **Important `ViewGroup` Attributes**

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`layout_height`	Parent view Child view	Height of the view. Used on attribute for child view controls within layouts. Required in some layouts, optional in others.	Dimension value or `match_parent` or `wrap_content`
`layout_width`	Parent view Child view	Width of the view. Used on attribute for child view controls within layouts. Required in some layouts, optional in others.	Dimension value or `match_parent` or `wrap_content`
`layout_margin`	Parent view Child view	Extra space around all sides of view.	Dimension value. Use more specific margin attributes to control individual margin sides, if necessary.

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="match_parent"
    android:layout_height="match_parent">
    <TextView
        android:id="@+id/TextView01"
        android:layout_height="match_parent"
        android:layout_width="match_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"
```

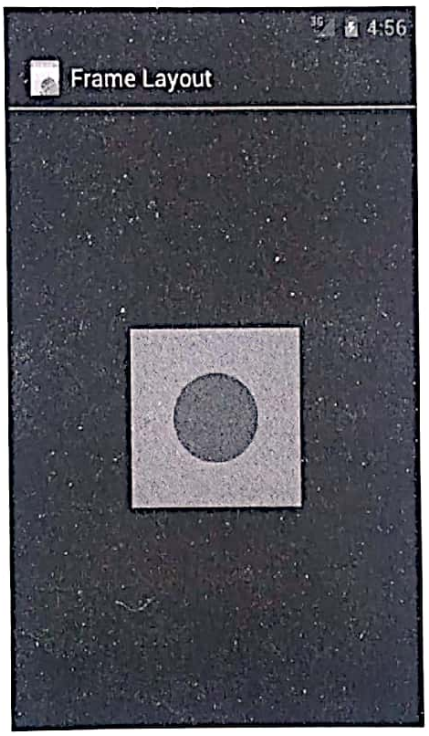
```
android:layout_marginRight="20px"
android:layout_marginTop="60px" />
```

Remember that a layout element 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 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 2, and the layout is sized to the largest child `View` in the stack.



Frame can be place in top-left side.

Figure 2 An example of `FrameLayout` usage.

You can find the layout attributes available for `FrameLayout` child `View` controls in `android.widget.FrameLayout.LayoutParams`. Table 2 describes some of the important attributes specific to `FrameLayout` views.

Table 2 **Important `FrameLayout` View Attributes**

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`foreground`	Parent view	Drawable to draw over the content.	Drawable resource.
`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`.
`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.
`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`.

Here's an example of an XML layout resource with a `FrameLayout` and two child `View` controls, both `ImageView` controls. 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` controls in a single row, as shown in Figure 3, or a single column, depending on whether its orientation attribute is set to horizontal or vertical. This is a very handy layout method for creating forms.

By default Horizontal property

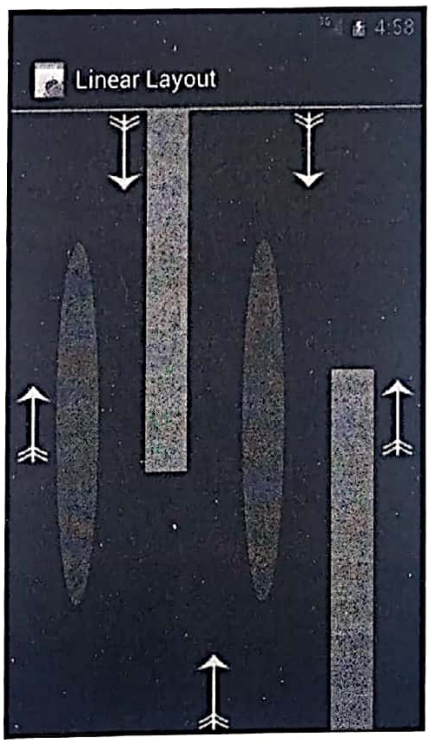


Figure 3 An example of `LinearLayout` (horizontal orientation).

You can find the layout attributes available for `LinearLayout` child `View` controls in `android.widget.LinearLayout.LayoutParams`. Table 3 describes some of the important attributes specific to `LinearLayout` views.

Table 3 **Important `LinearLayout` View Attributes**

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`orientation`	Parent view	Layout is a single row (horizontal) or single column (vertical) of controls.	Either `horizontal` or `vertical`.
`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`.
`weightSum`	Parent view	Sum of all child control weights.	A number that defines the sum of all child control weights. Default is `1`.
`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`.
`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 the `weightSum` attribute of the parent `LinearLayout` control. For example, one child control might have a value of `.3` and another 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` controls 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 4 shows how the button controls are relative to each other.

Figure 4 An example of `RelativeLayout` usage.

You can find the layout attributes available for `RelativeLayout` child `View` controls in `android.widget.RelativeLayout.LayoutParams`. Table 4 describes some of the important attributes specific to `RelativeLayout` views.

Table 4 Important `RelativeLayout` View Attributes

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

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`layout_alignTop`	Child view	Aligns top edge of child view with top edge of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`layout_alignBottom`	Child view	Aligns bottom edge of child view with bottom edge of another child view, specified by ID.	A view ID; for example, `@id/Button1`
`layout_above`	Child view	Positions bottom edge of child view above another child view, specified by ID.	A view ID; for example, `@id/Button1`
`layout_below`	Child view	Positions top edge of child view below another child view, specified by ID.	A view ID; for example, `@id/Button1`
`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`
`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` controls—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="match_parent"
    android:layout_width="match_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 5. You add individual `View` controls 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` controls). 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 5 demonstrates this); otherwise, the `View` control 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` controls instead of `TableRow` elements, if you want the `View` to take up an entire row.

You can find the layout attributes available for `TableLayout` child `View` controls in `android.widget.TableLayout.LayoutParams`. You can find the layout attributes available for `TableRow` child `View` controls in `android.widget.TableRow.LayoutParams`. Table 5 describes some of the important attributes specific to `TableLayout` controls.

Table 5 **Important `TableLayout` and `TableRow` View Attributes**

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`collapseColumns`	`TableLayout`	A comma-delimited list of column indices to collapse (zero-based)	String or string resource. For example, `"0,1,3,5"`.

column ?
merge ?

Attribute Name (All begin with android:)	Applies To	Description	Value
shrinkColumns	TableLayout	A comma-delimited list of column indices to shrink (zero-based)	String or string resource. Use "*" for all columns. For example, "0,1,3,5".
stretchColumns	TableLayout	A comma-delimited list of column indices to stretch (zero-based)	String or string resource. Use "*" for all columns. For example, "0,1,3,5".
layout_column	TableRow child view	Index of column this child view should be displayed in (zero-based)	Integer or integer resource. For example, 1.
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.

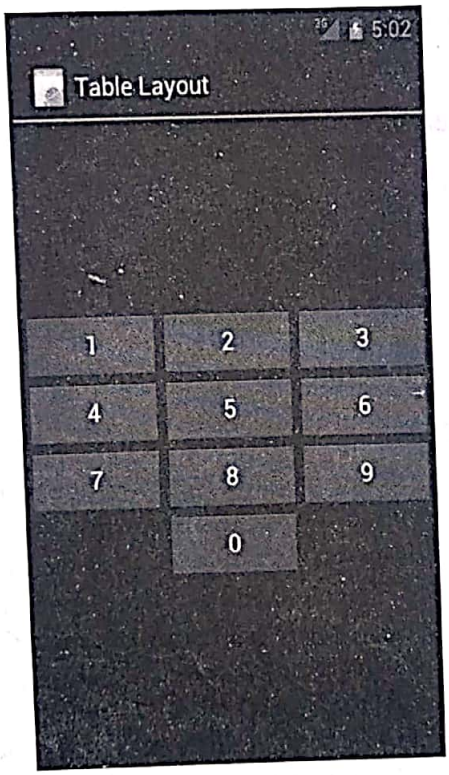


Figure 5 An example of TableLayout usage.

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` control into the second column explicitly:

```
<TableLayout xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:id="@+id/TableLayout01"
    android:layout_width="match_parent"
    android:layout_height="match_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 `GridLayout`

Introduced in Android 4.0 (API Level 14), the `GridLayout` organizes its children inside a grid. But don't confuse it with `GridView`; this layout grid is dynamically created. Unlike a `TableLayout`, child `View` controls in a `GridLayout` can span rows and columns and are more flat and efficient in terms of layout rendering. In fact, it is the child `View` controls of a `GridLayout` that tell the layout where they want to be placed. Figure 6 shows an example of a `GridLayout` with five child controls.

You can find the layout attributes available for `GridLayout` child `View` controls in `android.widget.GridLayout.LayoutParams`. Table 6 describes some of the important attributes specific to `GridLayout` controls.

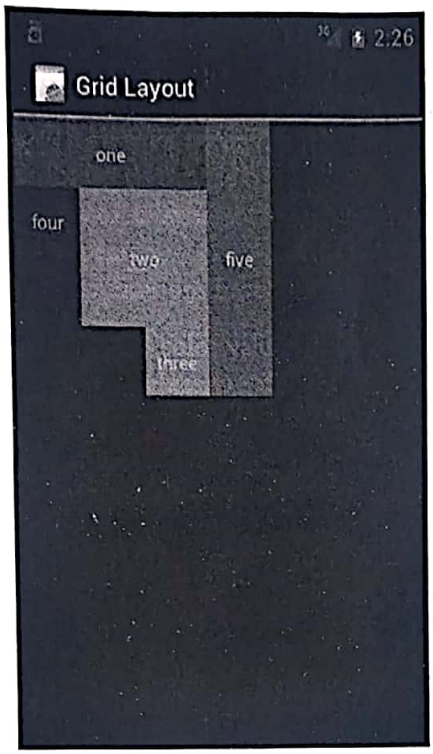


Figure 6 An example of `GridLayout` usage.

Table 6 **Important `GridLayout` View Attributes**

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`columnCount`	`GridLayout`	Defines a fixed number of columns for the grid.	A whole number; for example `"4"`.
`rowCount`	`GridLayout`	Defines a fixed number of rows for the grid.	A whole number; for example `"3"`.
`orientation`	`GridLayout`	When a row or column value is not specified on a child, this is used to determine whether the next child is down a row or over a column.	Can be `vertical` (down a row) or `horizontal` (over a column).

Table 6 **Continued**

Attribute Name (All begin with `android:`)	Applies To	Description	Value
`layout_column`	Child view of `GridLayout`	Index of column this child view should be displayed in (zero-based).	Integer or integer resource. For example, `1`.
`layout_columnSpan`	Child view of `GridLayout`	Number of columns this child view should span across.	Integer or integer resource greater than or equal to 1. For example, `3`
`layout_row`	Child view of `GridLayout`	Index of row this child view should be displayed in (zero-based).	Integer or integer resource. For example, `1`.
`layout_rowSpan`	Child view of `GridLayout`	Number of columns this child view should span down.	Integer or integer resource greater than or equal to 1. For example, `3`.
`layout_gravity`	Child view of `GridLayout`	Specifies the "direction" in which the view should be placed within the grid cells it will occupy.	One or more constants separated by "|". Some of the constants available are `baseline`, `top`, `bottom`, `left`, `right`, `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center`, and `fill`. Defaults to `baseline|left`.

The following is an example of an XML layout resource with a `GridLayout` view resulting in four rows and four columns. Each child control occupies a certain number of rows and columns. Because the default span attribute value is `1`, we only specify when the element will take up more than one row or column. For instance, the first `TextView` is one row high and three columns wide. The height and width of each of the `View` controls is specified to control the look of the result; otherwise, the `GridLayout` control will automatically assign sizing.

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/gridLayout1"
    android:layout_width="match_parent"
```

```
android:layout_height="match_parent"
android:columnCount="4"
android:rowCount="4" >

<TextView
    android:layout_width="150dp"
    android:layout_height="50dp"
    android:layout_column="0"
    android:layout_columnSpan="3"
    android:layout_row="0"
    android:background="#ff0000"
    android:gravity="center"
    android:text="one" />

 <TextView
     android:layout_width="100dp"
     android:layout_height="100dp"
     android:layout_column="1"
     android:layout_columnSpan="2"
     android:layout_row="1"
     android:layout_rowSpan="2"
     android:background="#ff7700"
     android:gravity="center"
     android:text="two" />

  <TextView
      android:layout_width="50dp"
      android:layout_height="50dp"
      android:layout_column="2"
      android:layout_row="3"
      android:background="#00ff00"
      android:gravity="center"
      android:text="three" />

   <TextView
       android:layout_width="50dp"
       android:layout_height="50dp"
       android:layout_column="0"
       android:layout_row="1"
       android:background="#0000ff"
       android:gravity="center"
       android:text="four" />

    <TextView
```

```
        android:layout_width="50dp"
        android:layout_height="200dp"
        android:layout_column="3"
        android:layout_row="0"
        android:layout_rowSpan="4"
        android:background="#0077ff"
        android:gravity="center"
        android:text="five" />

</GridLayout>
```

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` controls and is, itself, a `View` control, it can contain other layouts.

Tip

Want to create a certain amount of space between `View` controls without using a nested layout? Check out the new `Space` view (`android.widget.Space`) added in Ice Cream Sandwich (Android 4.0).

Figure 7 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 Container Control Classes

Layouts are not the only controls that can contain other `View` controls. Although layouts are useful for positioning other `View` controls on the screen, they aren't interactive. Now let's talk about the other kind of `ViewGroup`: the containers. These `View` controls encapsulate other, simpler `View` types and give the user the ability to interactively browse the child `View` controls in a standard fashion. Much like layouts, these controls each have a special, well-defined purpose.

Figure 7 An example of multiple layouts used together.

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

- Lists, grids, and galleries
- Tabs with `TabHost` and `TabControl`
- `ScrollView` and `HorizontalScrollView` for scrolling
- `ViewFlipper`, `ImageSwitcher`, and `TextSwitcher` for switching
- `SlidingDrawer` for hiding and showing content

Tip

Many of the code examples provided in this chapter are taken from the AdvancedLayouts application. The source code for the AdvancedLayouts application is provided for download on the book's websites.

Using Data-Driven Containers

Some of the `View` container controls are designed for displaying repetitive `View` controls in a particular way. Examples of this type of `View` container control include `ListView`, `GridView`, and `GalleryView`:

- **`ListView`**: Contains a vertically scrolling, horizontally filled list of `View` controls, each of which typically contains a row of data; the user can choose an item to perform some action upon.
- **`GridView`**: Contains a grid of `View` controls, with a specific number of columns; this container is often used with image icons; the user can choose an item to perform some action upon.
- **`GalleryView`**: Contains a horizontally scrolling list of `View` controls, also often used with image icons; the user can select an item to perform some action upon.

These containers are all types of `AdapterView` controls. An `AdapterView` control contains a set of child `View` controls to display data from some data source. An `Adapter` generates these child `View` controls from a data source. Because this is an important part of all these container controls, we talk about the `Adapter` objects first.

In this section, you learn how to bind data to `View` controls using `Adapter` objects. In the Android SDK, an `Adapter` reads data from some data source and generates the data for a `View` control based on some rules, depending on the type of `Adapter` used. This `View` is used to populate the child `View` controls of a particular `AdapterView`.

The most common `Adapter` classes are the `CursorAdapter` and the `ArrayAdapter`. The `CursorAdapter` gathers data from a `Cursor`, whereas the `ArrayAdapter` gathers data from an array. A `CursorAdapter` is a good choice to use when using data from a database. The `ArrayAdapter` is a good choice to use when there is only a single column of data or when the data comes from a resource array.

You should know some common elements of `Adapter` objects. When creating an `Adapter`, you provide a layout identifier. This layout is the template for filling in each row of data. The template you create contains identifiers for particular controls that the `Adapter` assigns data to. A simple layout can contain as little as a single `TextView` control. When making an `Adapter`, refer to both the layout resource and the identifier of the `TextView` control. The Android SDK provides some common layout resources for use in your application.

Using `ArrayAdapter`

An `ArrayAdapter` binds each element of the array to a single `View` control within the layout resource. Here is an example of creating an `ArrayAdapter`:

```
private String[] items = {
    "Item 1", "Item 2", "Item 3" };
ArrayAdapter adapt =
    new ArrayAdapter<String>
        (this, R.layout.textview, items);
```

In this example, we have a `String` array called `items`. This is the array used by the `ArrayAdapter` as the source data. We also use a layout resource, which is the `View` that is repeated for each item in the array. This is defined as follows:

```
<TextView xmlns:android=
    "http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:textSize="20px" />
```

This layout resource contains only a single `TextView`. However, you can use a more complex layout with constructors that also take the resource identifier of a `TextView` within the layout. Each child `View` within the `AdapterView` that uses this `Adapter` gets one `TextView` instance with one of the strings from the `String` array.

If you have an array resource defined, you can also directly set the `entries` attribute for an `AdapterView` to the resource identifier of the array to automatically provide the `ArrayAdapter`.

Using `CursorAdapter`

A `CursorAdapter` binds one or more columns of data to one or more `View` controls within the layout resource provided. This is best shown with an example. We will also discuss `Cursor` objects later in this text and in *Android Wireless Application Development, Volume II: Advanced Topics* as well, when we discuss databases and content providers.

The following example demonstrates creating a `CursorAdapter` by querying the `Contacts` content provider. The `CursorAdapter` requires the use of a `Cursor`.

```
Cursor names = managedQuery(
    Contacts.Phones.CONTENT_URI, null, null, null, null);
startManagingCursor(names);
ListAdapter adapter = new SimpleCursorAdapter(
    this, R.layout.two_text,
    names, new String[] {
        Contacts.Phones.NAME,
        Contacts.Phones.NUMBER
    }, new int[] {
        R.id.scratch_text1,
        R.id.scratch_text2
    });
```

In this example, we present a couple of new concepts. First, you need to know that the `Cursor` must contain a field named `_id`. In this case, we know that the `Contacts` content provider does have this field. This field is used later when we handle the user selecting a particular item.

Note

Although the `Contacts` class has been deprecated by the `ContactsContract` class introduced in Android 2.0, `Contacts` is the only method for accessing contact information that works on both older and newer editions of Android without modification.

We make a call to `managedQuery()` to get the `Cursor`. Then, we instantiate a `SimpleCursorAdapter` as a `ListAdapter`. Our layout, `R.layout.two_text`, has two `TextView` controls in it, which are used in the last parameter. `SimpleCursorAdapter` enables us to match up columns in the database with particular controls in our layout. For each row returned from the query, we get one instance of the layout within our `AdapterView`.

Binding Data to the `AdapterView`

Now that you have an `Adapter` object, you can apply this to one of the `AdapterView` controls. Any of them works. Although the `Gallery` technically takes a `SpinnerAdapter`, the instantiation of `SimpleCursorAdapter` also returns a `SpinnerAdapter`. Here is an example of this with a `ListView`, continuing on from the previous sample code:

```
((ListView)findViewById(R.id.list)).setAdapter(adapter);
```

The call to the `setAdapter()` method of the `AdapterView`, a `ListView` in this case, should come after your call to `setContentView()`. This is all that is required to bind data to your `AdapterView`. Figure 8 shows the same data in a `ListView`, `Gallery`, and `GridView`.

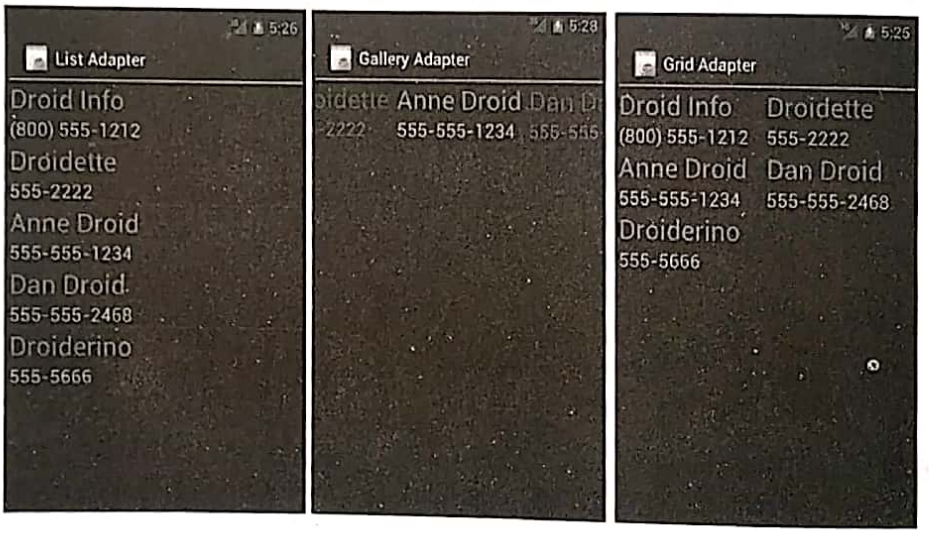


Figure 8 `ListView`, `Gallery`, and `GridView`: same data, same list item, different layout views.

Handling Selection Events

You often use `AdapterView` controls to present data from which the user should select. All three of the discussed controls—`ListView`, `GridView`, and `Gallery`—enable your application to monitor for click events in the same way. You need to call `setOnItemClickListener()` on your `AdapterView` and pass in an implementation of

the `AdapterView.OnItemClickListener` class. Here is a sample implementation of this class:

```
av.setOnItemClickListener(
    new AdapterView.OnItemClickListener() {
    public void onItemClick(
        AdapterView<?> parent, View view,
        int position, long id) {
        Toast.makeText(Scratch.this, "Clicked _id="+id,
           Toast.LENGTH_SHORT).show();
    }
});
```

In the preceding example, `av` is our `AdapterView`. The implementation of the `onItemClick()` method is where all the interesting work happens. The `parent` parameter is the `AdapterView` where the item was clicked. This is useful if your screen has more than one `AdapterView` on it. The `View` parameter is the specific `View` within the item that was clicked. The position is the zero-based position within the list of items that the user selects. Finally, the `id` parameter is the value of the `_id` column for the particular item that the user selects. This is useful for querying for further information about that particular row of data that the item represents.

Your application can also listen for long-click events on particular items. Additionally, your application can listen for selected items. Although the parameters are the same, your application receives a call as the highlighted item changes. This can be in response to the user scrolling with the arrow keys and not selecting an item for action.

Using `ListActivity`

The `ListView` control is commonly used for full-screen menus or lists of items from which a user selects. As such, you might consider using `ListActivity` as the base class for such screens. Using the `ListActivity` can simplify these types of screens.

First, to handle item events, you now need to provide an implementation in your `ListActivity`. For instance, the equivalent of `onListItemClickListener` is to implement the `onListItemClick()` method within your `ListActivity`.

Second, to assign an `Adapter`, you need a call to the `setListAdapter()` method. You do this after the call to the `setContentView()` method. However, this hints at some of the limitations of using `ListActivity`.

To use `ListActivity`, the layout that is set with the `setContentView()` method must contain a `ListView` with the identifier set to `android:list`; this cannot be changed. Second, you can also have a `View` with an identifier set to `android:empty` to have a `View` display when no data is returned from the `Adapter`. Finally, this works only with `ListView` controls, so it has limited use. However, when it does work for your application, it can save on some coding.

Tip

You can create `ListView` headers and footers using `ListView.FixedViewInfo` with the `ListView` methods `addHeaderView()` and `addFooterView()`.

Organizing Screens with Tabs

The Android SDK has a flexible way to provide a tab interface to the user. Much like `ListView` and `ListActivity`, there are two ways to create tabbing on the Android platform. You can either use the `TabActivity`, which simplifies a screen with tabs, or you can create your own tabbed screens from scratch. Both methods rely on the `TabHost` control.

Warning

The Eclipse Layout Resource editor does not display `TabHost` controls properly in design mode. In order to design this kind of layout, you should stick to the XML layout mode. You must use the Android emulator or an Android device to view the tabs.

Using `TabActivity`

A screen layout with tabs consists of a `TabActivity` and a `TabHost`. The `TabHost` consists of `TabSpecs`, a nested class of `TabHost`, which contains the tab information including the tab title and the contents of the tab. The contents of the tab can either be a predefined `View`, an `Activity` launched through an `Intent` object, or a factory-generated `View` using the `TabContentFactory` interface.

Tabs aren't as complex as they might sound at first. Each tab is effectively a container for a `View`. That `View` can come from any `View` that is ready to be shown, such as an XML layout file. Alternatively, that `View` can come from launching an `Activity`. The following example demonstrates each of these methods using `View` controls and `Activity` objects created in the previous examples of this chapter:

```
public class TabLayout
    extends TabActivity
    implements android.widget.TabHost.TabContentFactory {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TabHost tabHost = getTabHost();
        LayoutInflater.from(this).inflate(
            R.layout.example_layout,
            tabHost.getTabContentView(), true);
        tabHost.addTab(tabHost.newTabSpec("tab1")
            .setIndicator("Grid").setContent(
                new Intent(this, GridLayout.class)));
        tabHost.addTab(tabHost.newTabSpec("tab2")
            .setIndicator("List").setContent(
```

```
            new Intent(this, List.class)));
        tabHost.addTab(tabHost.newTabSpec("tab3")
            .setIndicator("Basic").setContent(
                R.id.two_texts));
        tabHost.addTab(tabHost.newTabSpec("tab4")
            .setIndicator("Factory").setContent(
                this));
    }

    public View createTabContent(String tag) {
        if (tag.compareTo("tab4") == 0) {
            TextView tv = new TextView(this);
            Date now = new Date();
            tv.setText("I'm from a factory. Created: "
                + now.toString());
            tv.setTextSize((float) 24);
            return (tv);
        } else {
            return null;
        }
    }
}
```

This example creates a tabbed layout view with four tabs on it, as shown in Figure 9. The first tab is from the recent `GridView` example. The second tab is from the `ListView` example before that. The third tab is the basic layout with two `TextView` controls, fully defined in an XML layout file, as previously demonstrated. Finally, the fourth tab is created with a factory.

The first action is to get the `TabHost` instance. This is the object that enables us to add `Intent` objects and `View` identifiers for drawing the screen. A `TabActivity` provides a method to retrieve the current `TabHost` object.

The next action is only loosely related to tab views. The `LayoutInflater` is used to turn the XML definition of a `View` into the actual `View` controls. This would normally happen when calling `setContentView()`, but we're not doing that. The use of the `LayoutInflater` is required for referencing the `View` controls by identifier, as is done for the third tab.

The code finishes up by adding each of the four tabs to the `TabHost` in the order that they will be presented. This is accomplished by multiple calls to the `addTab()` method of `TabHost`. The first two calls are essentially the same. Each one creates a new `Intent` with the name of an `Activity` that launches within the tab. These are the same `Activity` classes used previously for full-screen display. Even if the `Activity` isn't designed for full-screen use, this should work seamlessly.

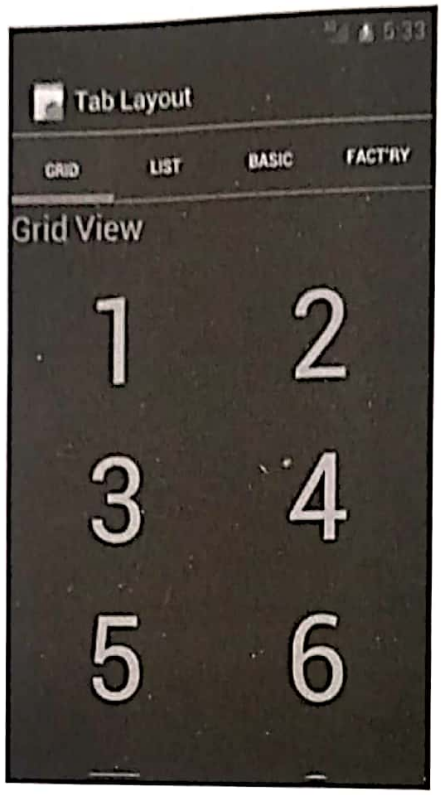


Figure 9 Four tabs displayed.

Next, on the third tab, a layout `View` is added using its identifier. In the preceding call to the `LayoutInflater`, the layout file also contains an identifier matching the one used here at the top level of a `LinearLayout` definition. This is the same one used previously to show a basic `LinearLayout` example. Again, there was no need to change anything in this view for it to display correctly in a tab.

Next, a tab referencing the content as the `TabActivity` class is added. This is possible because the class itself also implements `TabHost.TabContentFactory`, which requires implementing the `createTabContent()` method. The view is created the first time the user selects the tab, so no other information is needed here. The tag that creates this tab must be kept track of, though, as that's how the tabs are identified to the `TabHost`.

Finally, the method `createTabContent()` is implemented for use with the fourth tab. The first task here is to check the tag to see if it's the one kept track of for the fourth tab. When that is confirmed, an instance of the `TextView` control is created and a text string assigned to it, which contains the current time. The size of the text is set to 24 pixels. The time stamp used in this string can be used to demonstrate when the view is created and that it's not re-created by simply changing tabs.

The flexibility of tabs that Android provides is great for adding navigation to an application that has a bunch of views already defined. Few changes, if any, need to be made to existing `View` and `Activity` objects for them to work within the context of a `TabHost`.

Note

It is possible to design tabbed layouts without using the `TabActivity` class. However, this requires a bit of knowledge about the underpinnings of the `TabHost` and `TabWidget` controls. To define a set of tabs within an XML layout file without using `TabActivity`, begin with a `TabHost` (for example, `TabHost1`). This `TabHost` must have the ID `@android:id/tabhost`. Inside the `TabHost`, include a vertically oriented `LinearLayout` that must contain a `TabWidget` (which must have the ID `@android:id/tabs`) and a `FrameLayout` (which must have the ID `@android:id/tabcontent`). The contents of each tab are then defined within the `FrameLayout`.

After you've defined the `TabHost` properly in XML, you must load and initialize it using the `TabHost setup()` method on your activity's `onCreate()` method. First, you need to create a `TabSpec` for each tab, setting the tab indicator using the `setIndicator()` method and the tab content using the `setContent()` method. Next, add each tab using the `addTab()` method of the `TabHost`. Finally, you should set the default tab of the `TabHost`, using a method such as `setCurrentTabByTag()`.

Note

In Android 3.0 and up, including Ice Cream Sandwich, the `ActionBar` control allows setting tabs, as well. Instead of loading views, these tabs allow the loading of `Fragments`. See http://goo.gl/ZRR5j for more information.

Adding Scrolling Support

One of the easiest ways to provide vertical scrolling for a screen is by using the `ScrollView` (vertical scrolling) and `HorizontalScrollView` (horizontal scrolling) controls. Either control can be used as a wrapper container, causing all child `View` controls to have one continuous scroll bar. The `ScrollView` and `HorizontalScrollView` controls can have only one child, though, so it's customary to have that child be a layout, such as a `LinearLayout`, which then contains all the "real" child controls to be scrolled through.

Tip

The code examples of scrolling in this section are provided in the SimpleScrolling application. The source code for the SimpleScrolling application is available for download on the book's websites.

Figure 10 shows a screen with and without a `ScrollView` control.

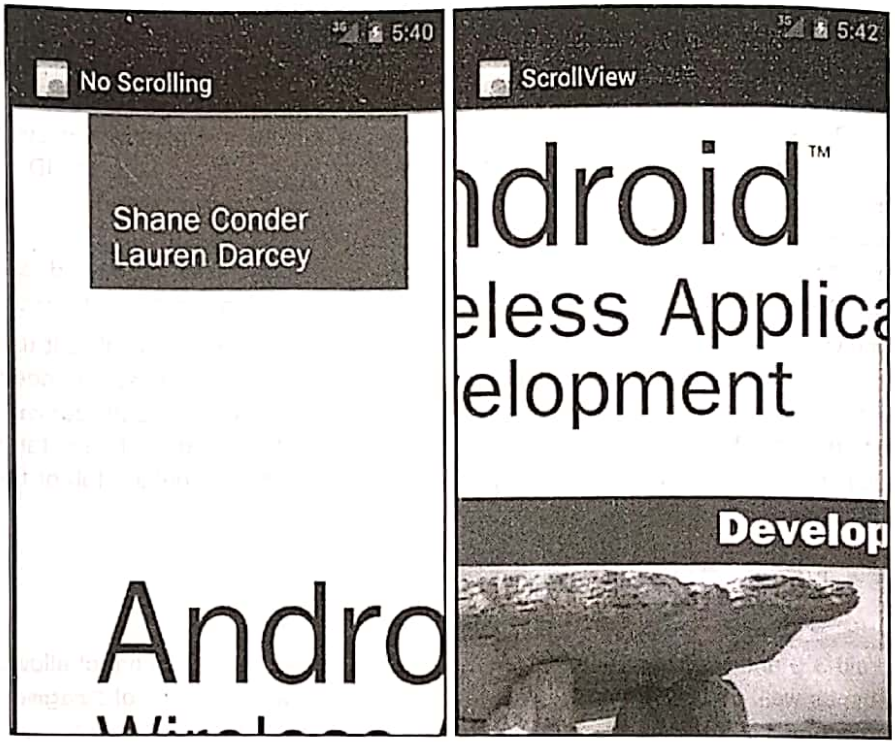


Figure 10 A screen without (left) and with (right) a `ScrollView` control.

Exploring Other View Containers

Many other user interface controls are available within the Android SDK. Some of these controls are listed here:

- **Switchers**: A `ViewSwitcher` control contains only two child `View` controls and only one of those is shown at a time. It switches between the two, animating as it does so. Primarily, the `ImageSwitcher` and `TextSwitcher` objects are used. Each one provides a way to set a new child `View`, either a `Drawable` resource or a text string, and then animates from what is displayed to the new contents.
- **Sliding drawer**: Another `View` container is the `SlidingDrawer` control. This control includes two parts: a handle and a container view. The user drags the handle open and the internal contents are shown; then the user can drag the handle shut and the content disappears. The `SlidingDrawer` can be used horizontally or vertically and is always used from within a layout representing the larger screen. This makes the `SlidingDrawer` especially useful for application configurations such as game controls. A user can pull out the drawer, pause the game, change some features, and then close the `SlidingDrawer` to resume the game.