CHAPTER 3

Laying Out Controls in Containers

IN THIS CHAPTER

A container is a view used to contain other views. Android offers a collection of view classes that act as containers for views. These container classes are called layouts, and as the name suggests, they decide the organization, size, and position of their children views.

Let's start the chapter with an introduction to different layouts used in Android applications.

Introduction to Layouts

Layouts are basically containers for other items known as `Views`, which are displayed on the screen. Layouts help manage and arrange views as well. Layouts are defined in the form of XML files that cannot be changed by our code during runtime.

Table 3.1 shows the layout managers provided by the Android SDK.

TABLE 3.1 Android Layout Managers

Layout Manager	Description
LinearLayout	Organizes its children either horizontally or vertically
RelativeLayout	Organizes its children relative to one another or to the parent
AbsoluteLayout	Each child control is given a specific location within the bounds of the container

Layout Manager	Description
FrameLayout	Displays a single view; that is, the next view replaces the previous view and hence is used to dynamically change the children in the layout
TableLayout	Organizes its children in tabular form
GridLayout	Organizes its children in grid format

The containers or layouts listed in Table 3.1 are also known as `ViewGroups` as one or more `Views` are grouped and arranged in a desired manner through them. Besides the `ViewGroups` shown here Android supports one more `ViewGroup` known as ScrollView, which is discussed in Chapter 4, "Utilizing Resources and Media."

LinearLayout

The LinearLayout is the most basic layout, and it arranges its elements sequentially, either horizontally or vertically. To arrange controls within a linear layout, the following attributes are used:

- **`android:orientation`**—Used for arranging the controls in the container in horizontal or vertical order
- **`android:layout_width`**—Used for defining the width of a control
- **`android:layout_height`**—Used for defining the height of a control
- **`android:padding`**—Used for increasing the whitespace between the boundaries of the control and its actual content
- **`android:layout_weight`**—Used for shrinking or expanding the size of the control to consume the extra space relative to the other controls in the container
- **`android:gravity`**—Used for aligning content within a control
- **`android:layout_gravity`**—Used for aligning the control within the container

Applying the `orientation` Attribute

The `orientation` attribute is used to arrange its children either in horizontal or vertical order. The valid values for this attribute are `horizontal` and `vertical`. If the value of the `android:orientation` attribute is set to `vertical`, the children in the linear layout are arranged in a column layout, one below the other. Similarly, if the value of the `android:orientation` attribute is set to `horizontal`, the controls in the linear layout are arranged in a row format, side by side. The orientation can be modified at runtime through the `setOrientation()` method. That is, by supplying the values `HORIZONTAL` or `VERTICAL` to the `setOrientation()` method, we can arrange the children of the LinearLayout in row or column format, respectively.

Applying the `height` and `width` Attributes

The default height and width of a control are decided on the basis of the text or content that is displayed through it. To specify a certain height and width to the control, we use the `android:layout_width` and `android:layout_height` attributes. We can specify the values for the `height` and `width` attributes in the following three ways:

- By supplying specific dimension values for the control in terms of `px` (pixels), `dip/dp` (device independent pixels), `sp` (scaled pixels), `pts` (points), `in` (inches), and `mm` (millimeters). For example, the `android:layout_width="20px"` attribute sets the width of the control to 20 pixels.
- By providing the value as `wrap_content`. When assigned to the control's height or width, this attribute resizes the control to expand to fit its contents. For example, when this value is applied to the width of the `TextView`, it expands so that its complete text is visible.
- By providing the value as `match_parent`. When assigned to the control's height or width, this attribute forces the size of the control to expand to fill up all the available space of the enclosing container.

NOTE

For layout elements, the value `wrap_content` resizes the layout to fit the controls added as its children. The value `match_parent` makes the layout expand to take up all the space in the parent layout.

Applying the `padding` Attribute

The `padding` attribute is used to increase the whitespace between the boundaries of the control and its actual content. Through the `android:padding` attribute, we can set the same amount of padding or spacing on all four sides of the control. Similarly, by using the `android:paddingLeft`, `android:paddingRight`, `android:paddingTop`, and `android:paddingBottom` attributes, we can specify the individual spacing on the left, right, top, and bottom of the control, respectively.

The following example sets the spacing on all four sides of the control to 5 pixels:

```
android:padding="5dip"
```

Similarly, the following example sets the spacing on the left side of the control to 5 pixels:

```
android:paddingLeft="5dip"
```

NOTE

To set the padding at runtime, we can call the `setPadding()` method.

Let's see how the controls are laid out in the LinearLayout layout using an example. Create a new Android Project called `LinearLayoutApp`. The original default content of the layout file `activity_linear_layout_app.xml` appears as shown in Listing 3.1.

LISTING 3.1 Default Code in the Layout File `activity_linear_layout_app.xml`

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/hello_world"
        tools:context=".LinearLayoutAppActivity" />
</RelativeLayout>
```

Let's apply the LinearLayout and add three `Button` controls to the layout. Modify the `activity_linear_layout_app.xml` to appear as shown in Listing 3.2.

LISTING 3.2 The `activity_linear_layout_app.xml` File on Adding Three `Button` Controls

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <Button
        android:id="@+id/Apple"
        android:text="Apple"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/Mango"
        android:text="Mango"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/Banana"
        android:text="Banana"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```

The orientation of LinearLayout is set to vertical, declaring that we want to arrange its child elements vertically, one below the other. The height and width of the layout are set to expand to fill up all the available space of the enclosing container, that is, the device screen. Three Button controls are added to the layout, which appear one below the other. The IDs and text assigned to the three Button controls are Apple, Mango, and Banana, respectively. The height of the three controls is set to wrap_content, which is enough to accommodate the text. Finally, the width of the three controls is set to match_parent, so that the width of the three controls expands to fill up the available space of the LinearLayout container. We see the output shown in Figure 3.1.

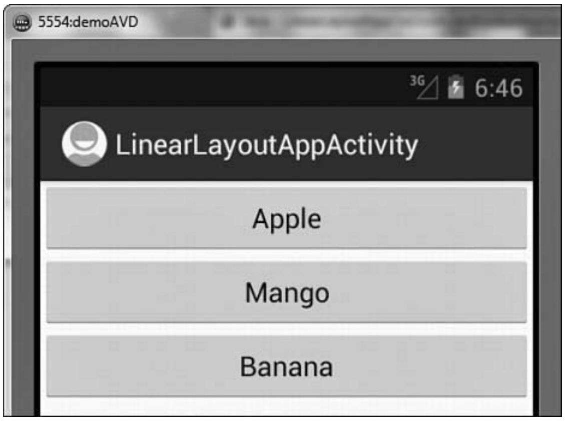


FIGURE 3.1 Three Button controls arranged vertically in LinearLayout

To see the controls appear horizontally, set the orientation attribute of the LinearLayout to horizontal. We also need to set the layout_width attribute of the three controls to wrap_content; otherwise, we will be able to see only the first Button control, the one with the Apple ID. If the layout_width attribute of any control is set to match_parent, it takes up all the available space of the container, hiding the rest of the controls behind it. By setting the values of the layout_width attributes to wrap_content, we make sure that the width of the control expands just to fit its content and does not take up all the available space. Let's modify the activity_linear_layout_app.xml to appear as shown in Listing 3.3.

LISTING 3.3 The activity_linear_layout_app.xml File on Setting Horizontal Orientation to the Button Controls

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal" >
    <Button
        android:id="@+id/Apple"
        android:text="Apple"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/Mango"
        android:text="Mango"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/Banana"
        android:text="Banana"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

The controls are arranged horizontally, as shown in Figure 3.2.

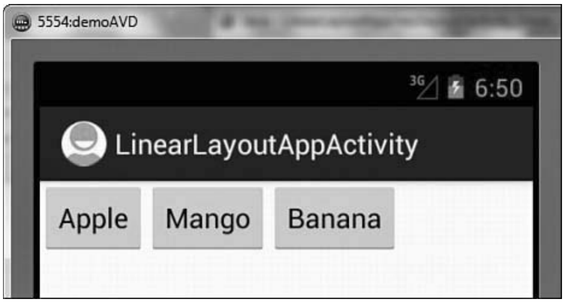


FIGURE 3.2 Three `Button` controls arranged horizontally in LinearLayout

Applying the `weight` Attribute

The `weight` attribute affects the size of the control. That is, we use `weight` to assign the capability to expand or shrink and consume extra space relative to the other controls in the container. The values of the `weight` attribute range from `0.0` to `1.0`, where `1.0` is the highest value. Let's suppose a container has two controls and one of them is assigned the `weight` of `1`. In that case, the control assigned the `weight` of `1` consumes all the empty space in the container, whereas the other control remains at its current size. If we assign a `weight` of `0.0` to both the controls, nothing happens and the controls maintain their original size. If both the attributes are assigned the same value above `0.0`, both the controls consume the extra space equally. Hence, `weight` lets us apply a size expansion ratio to the controls. To make the middle `Button` control, `Mango`, take up all the available space of the container, let's assign a `weight` attribute to the three controls. Modify the `activity_linear_layout_app.xml` file to appear as shown in Listing 3.4.

LISTING 3.4 The `activity_linear_layout_app.xml` File on Applying the `weight` Attribute to the `Button` Controls

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/Apple"
        android:text="Apple"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0.0" />
    <Button
        android:id="@+id/Mango"
        android:text="Mango"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1.0" />
    <Button
        android:id="@+id/Banana"
        android:text="Banana"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0.0" />
</LinearLayout>
```

By setting the `layout_weight` attributes of `Apple`, `Mango`, and `Banana` to `0.0`, `1.0`, and `0.0`, respectively, we allow the `Mango` button control to take up all the available space of the container, as shown in Figure 3.3 (left). If we set the value of `layout_weight` of the `Banana` button control to `1.0` and that of `Mango` back to `0.0`, then all the available space of the container is consumed by the `Banana` button control, as shown in Figure 3.3 (middle). Similarly if we set the `layout_weight` of all controls to `1.0`, the entire container space will be equally consumed by the three controls, as shown in Figure 3.3 (right).

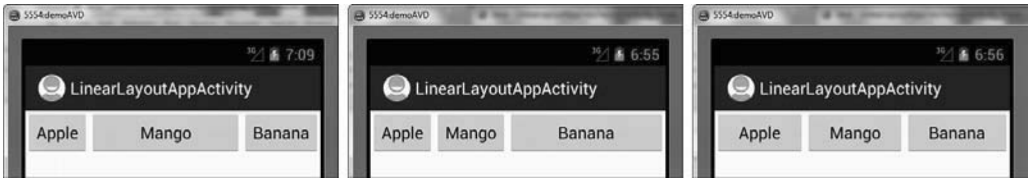


FIGURE 3.3 (left) The `weight` attribute of the Mango `Button` control set to 1.0, (middle) the `weight` attribute of the Banana `Button` control set to 1.0, and (right) all three `Button` controls set to the same `weight` attribute

Similarly if we set the weight of `Apple`, `Mango`, and `Banana` to `0.0`, `1.0`, and `0.5`, respectively, we get the output shown in Figure 3.4.

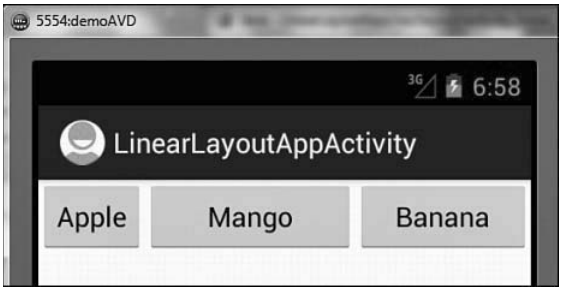


FIGURE 3.4 The `weight` attribute of the Apple, Mango, and Banana `Button` controls set to 0.0, 1.0, and 0.5

We can see that the text of the three controls is center-aligned. To align the content of a control, we use the `Gravity` attribute.

Applying the `Gravity` Attribute

The `Gravity` attribute is for aligning the content within a control. For example, to align the text of a control to the center, we set the value of its `android:gravity` attribute to `center`. The valid options for `android:gravity` include `left`, `center`, `right`, `top`, `bottom`, `center_horizontal`, `center_vertical`, `fill_horizontal`, and `fill_vertical`. The task performed by few of the said options is as follows:

- **`center_vertical`**—Places the object in the vertical center of its container, without changing its size
- **`fill_vertical`**—Grows the vertical size of the object, if needed, so it completely fills its container
- **`center_horizontal`**—Places the object in the horizontal center of its container, without changing its size
- **`fill_horizontal`**—Grows the horizontal size of the object, if needed, so it completely fills its container
- **`center`**—Places the object in the center of its container in both the vertical and horizontal axis, without changing its size

We can make the text of a control appear at the center by using the `android:gravity` attribute, as shown in this example:

```
android:gravity="center"
```

We can also combine two or more values of any attribute using the | operator. The following example centrally aligns the text horizontally and vertically within a control:

```
android:gravity="center_horizontal|center_vertical"
```

Figure 3.5 shows the `android:gravity` attribute set to left and right for the `Button` controls `Mango` and `Banana`.

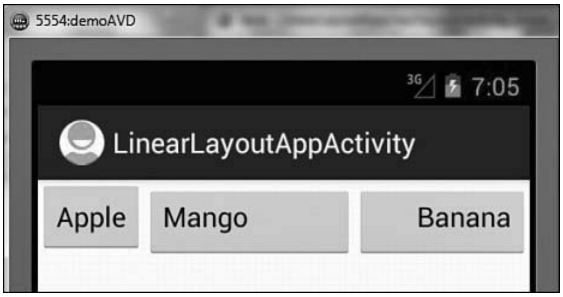


FIGURE 3.5 The text in the Mango and Banana `Button` controls aligned to the left and right, respectively, through the `android:gravity` attribute

Besides the `android:gravity` attribute, Android provides one more similar attribute, `android:layout_gravity`. Let's explore the difference between the two.

Using the `android:layout_gravity` Attribute

Where `android:gravity` is a setting used by the `View`, the `android:layout_gravity` is used by the container. That is, this attribute is used to align the control within the container. For example, to align the text within a `Button` control, we use the `android:gravity` attribute; to align the `Button` control itself in the LinearLayout (the container), we use the `android:layout_gravity` attribute. Let's add the `android:layout_gravity` attribute to align the `Button` controls themselves. To see the impact of using the `android:layout_gravity` attribute to align the `Button` controls in the LinearLayout, let's first arrange them vertically. So, let's modify `activity_linear_layout_app.xml` to make the `Button` controls appear vertically, one below the other as shown in Listing 3.5.

LISTING 3.5 The `activity_linear_layout_app.xml` File on Arranging the `Button` Controls Vertically

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/Apple"
        android:text="Apple"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/Mango"
        android:text="Mango"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <Button
        android:id="@+id/Banana"
        android:text="Banana"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

The preceding code arranges the Button controls vertically, as shown in Figure 3.6 (left). To align the Button controls Mango and Banana to the center and to the right of the LinearLayout container, add the following statements to the respective tags in the activity_linear_layout_app.xml layout file:

```
android:layout_gravity="center"
```

and

```
android:layout_gravity="right"
```

The two Button controls, Mango and Banana, are aligned at the center and to the right in the container, as shown in Figure 3.6 (middle).

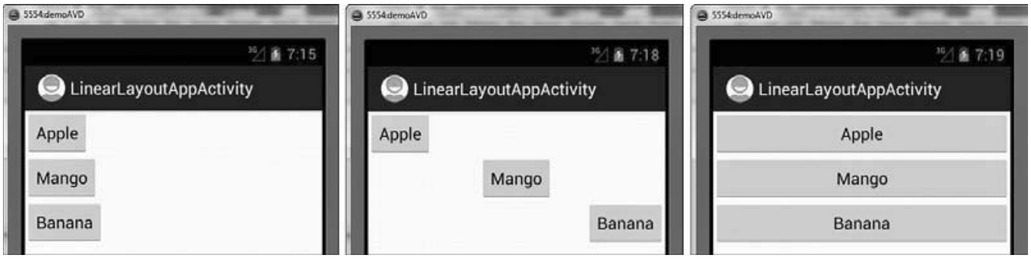


FIGURE 3.6 (left) The three Button controls vertically aligned with the width attribute set to wrap_content, (middle) the Mango and Banana Button controls aligned to the center and right of container, and (right) the width of the three Button controls expanded to take up all the available space

At the moment, the layout_width attribute of the three controls is set to wrap_content. The width of the three controls is just enough to accommodate their content. If we now set the value of the android:layout_width attribute for all three controls to match_parent, we find that all three Button controls expand in width to take up all the available space of the container, as shown in Figure 3.6 (right). Now we can apply the android:gravity attribute to align the text within the controls. Let's add the following three attributes to the Button controls Apple, Mango, and Banana:

```
android:gravity="left"
android:gravity="center"
```

and

```
android:gravity="right"
```

These lines of code align the content of the three Button controls to the left, to the center, and to the right within the control, as shown in Figure 3.7 (left). Because the three Button controls are arranged vertically in the layout (the orientation of the LinearLayout is set to vertical), the application of the weight attribute makes the controls expand vertically instead of horizontally as we saw earlier. To see the effect, let's add the following statement to the tags of all three `Button` controls:

```
android:layout_weight="0.0"
```

As expected, there will be no change in the height of any control, as the `weight` value assigned is `0.0`. Setting an equal value above `0.0` for all three controls results in equal division of empty space among them. For example, assigning the `android:layout_weight="1.0"` to all three controls results in expanding their height, as shown in Figure 3.7 (middle).

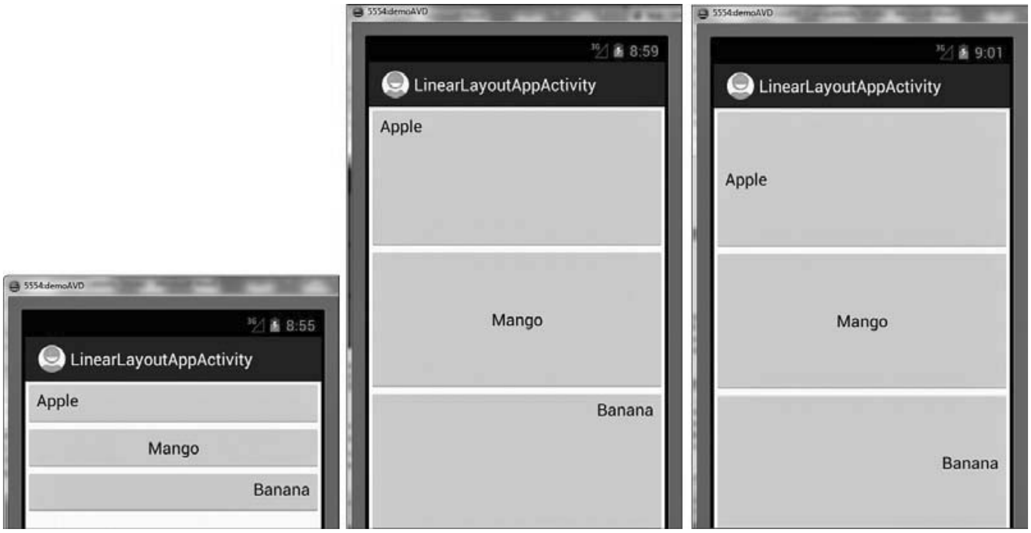


FIGURE 3.7 (left) The three `Button` controls with their text aligned to the left, center, and right, (middle) the vertical available space of the container apportioned equally among the three `Button` controls, and (right) the text of the three `Button` controls vertically aligned to the center

In the middle image of Figure 3.7, we see that the text in the `Apple` and `Banana` controls is not at the vertical center, so let's modify their `android:gravity` value, as shown here:

`android:gravity="center_vertical"` for the `Apple` control

`android:gravity="center_vertical|right"` for the `Banana` control

The `center_vertical` value aligns the content vertically to the center of the control, and the `right` value aligns the content to the right of the control. We can combine the values of the attribute using the `|` operator. After applying the values as shown in the preceding two code lines, we get the output shown in Figure 3.7 (right).

RelativeLayout

In RelativeLayout, each child element is laid out in relation to other child elements; that is, the location of a child element is specified in terms of the desired distance from the existing children. To understand the concept of relative layout practically, let's create a new Android project called `RelativeLayoutApp`. Modify its layout file `activity_relative_layout_app.xml` to appear as shown in Listing 3.6.

LISTING 3.6 The `activity_relative_layout_app.xml` File on Arranging the `Button` Controls in the RelativeLayout Container

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <Button
        android:id="@+id/Apple"
        android:text="Apple"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="15dip"
        android:layout_marginLeft="20dip" />
    <Button
        android:id="@+id/Mango"
        android:text="Mango"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="28dip"
        android:layout_toRightOf="@id/Apple"
        android:layout_marginLeft="15dip"
        android:layout_marginRight="10dip"
        android:layout_alignParentTop="true" />
    <Button
        android:id="@+id/Banana"
        android:text="Banana"
        android:layout_width="200dip"
        android:layout_height="50dip"
        android:layout_marginTop="15dip"
        android:layout_below="@id/Apple"
        android:layout_alignParentLeft="true" />
    <Button
        android:id="@+id/Grapes"
        android:text="Grapes"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:minWidth="100dp"
        android:layout_alignParentRight="true"
        android:layout_below="@id/Banana" />
    <Button
        android:id="@+id/Kiwi"
        android:text="Kiwi"
        android:layout_width="100dip"
        android:layout_height="wrap_content"
        android:layout_below="@id/Banana"
        android:paddingTop="15dip"
        android:paddingLeft="25dip"
        android:paddingRight="25dip" />
</RelativeLayout>
```

Before we understand how the controls in the previous code block are placed, let's have a quick look at different attributes used to set the positions of the layout controls.

Layout Control Attributes

The attributes used to set the location of the control relative to a container are

- **`android:layout_alignParentTop`**—The top of the control is set to align with the top of the container.
- **`android:layout_alignParentBottom`**—The bottom of the control is set to align with the bottom of the container.
- **`android:layout_alignParentLeft`**—The left side of the control is set to align with the left side of the container.
- **`android:layout_alignParentRight`**—The right side of the control is set to align with the right side of the container.
- **`android:layout_centerHorizontal`**—The control is placed horizontally at the center of the container.
- **`android:layout_centerVertical`**—The control is placed vertically at the center of the container.
- **`android:layout_centerInParent`**—The control is placed horizontally and vertically at the center of the container.

The attributes to control the position of a control in relation to other controls are

- **`android:layout_above`**—The control is placed above the referenced control.
- **`android:layout_below`**—The control is placed below the referenced control.
- **`android:layout_toLeftOf`**—The control is placed to the left of the referenced control.
- **`android:layout_toRightOf`**—The control is placed to the right of the referenced control.

The attributes that control the alignment of a control in relation to other controls are

- `android:layout_alignTop`— The top of the control is set to align with the top of the referenced control.

- **`android:layout_alignBottom`**—The bottom of the control is set to align with the bottom of the referenced control.
- **`android:layout_alignLeft`**—The left side of the control is set to align with the left side of the referenced control.
- **`android:layout_alignRight`**—The right side of the control is set to align with the right side of the referenced control.
- **`android:layout_alignBaseline`**—The baseline of the two controls will be aligned.

For spacing, Android defines two attributes: `android:layout_margin` and `android:padding`. The `android:layout_margin` attribute defines spacing for the container, while `android:padding` defines the spacing for the view. Let's begin with padding.

- **`android:padding`**—Defines the spacing of the content on all four sides of the control. To define padding for each side individually, use `android:paddingLeft`, `android:paddingRight`, `android:paddingTop`, and `android:paddingBottom`.
- **`android:paddingTop`**—Defines the spacing between the content and the top of the control.
- **`android:paddingBottom`**—Defines the spacing between the content and the bottom of the control.
- **`android:paddingLeft`**—Defines the spacing between the content and the left side of the control.
- **`android:paddingRight`**—Defines the spacing between the content and the right side of the control.

Here are the attributes that define the spacing between the control and the container:

- **`android:layout_margin`**—Defines the spacing of the control in relation to the controls or the container on all four sides. To define spacing for each side individually, we use the `android:layout_marginLeft`, `android:layout_marginRight`, `android:layout_marginTop`, and `android:layout_marginBottom` options.
- **`android:layout_marginTop`**—Defines the spacing between the top of the control and the related control or container.
- **`android:layout_marginBottom`**—Defines the spacing between the bottom of the control and the related control or container.
- **`android:layout_marginRight`**—Defines the spacing between the right side of the control and the related control or container.
- **`android:layout_marginLeft`**—Defines the spacing between the left side of the control and the related control or container.

The layout file `activity_relative_layout_app.xml` arranges the controls as follows:

> The `Apple` button control is set to appear at a distance of `15dip` from the top and `20dip` from the left side of the `RelativeLayout` container. The width of the `Mango` button control is set to consume the available horizontal space. The text `Mango` appears at a distance of `28dip` from all sides of the control. The `Mango` control is set to appear to the right of the `Apple` control. The control is set to appear at a distance of `15dip` from the control on the left and `10dip` from the right side of the relative layout container. Also, the top of the `Button` control is set to align with the top of the container.
>
> The `Banana` button control is assigned the `width` and `height` of `200dip` and `50dip`, respectively. The control is set to appear `15dip` below the `Apple` control. The left side of the control is set to align with the left side of the container.
>
> The `Grapes` button control is set to appear below the `Banana` button control, and its width is set to expand just enough to accommodate its content. The height of the control is set to take up all available vertical space. The text `Grapes` is automatically aligned vertically; that is, it appears at the center of the vertical height when the `height` attribute is set to `match_parent`. The minimum width of the control is set to `100dip`. The right side of the control is set to align with the right side of the container.
>
> The `Kiwi` Button control is set to appear below the `Banana` control. Its width is set to `100dip`, and the height is set to just accommodate its content. The text `Kiwi` is set to appear at the distance of `15dip`, `25dip`, and `25dip` from the top, left, and right boundary of the control.

We don't need to make any changes to the `RelativeLayoutAppActivity.java` file. Its original content is as shown in Listing 3.7.

LISTING 3.7 The Default Code in the Activity File `RelativeLayoutAppActivity.java`

```
package com.androidunleashed.relativelayoutapp;

import android.app.Activity;
import android.os.Bundle;

public class RelativeLayoutDemoActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_relative_layout_app);
    }
}
```

When the application is run, we see the output shown in Figure 3.8.

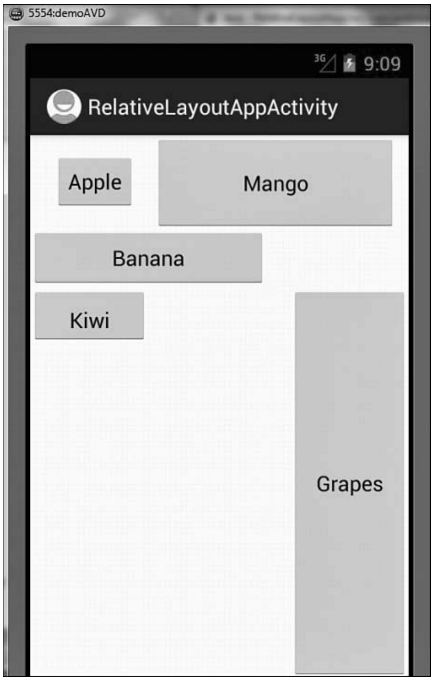


FIGURE 3.8 The five `Button` controls' layout relative to each other

We can make the text `Grapes` appear centrally at the top row by adding the following line:

```
android:gravity="center_horizontal"
```

So, its tag appears as follows:

```
<Button
    android:id="@+id/Grapes"
    android:text="Grapes"
    android:layout_width="wrap_content"
    android:layout_height="match_parent"
    android:minWidth="100dp"
    android:layout_alignParentRight="true"
    android:layout_below="@id/Banana"
    android:gravity="center_horizontal" />
```

The output is modified to appear as shown in Figure 3.9.

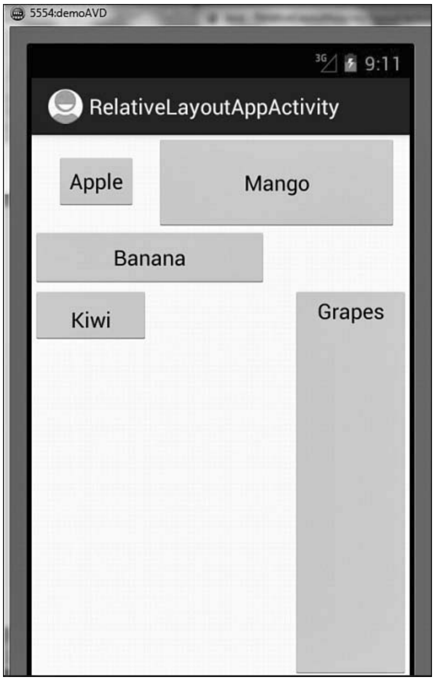


FIGURE 3.9 The Grapes `Button` control aligned horizontally at the center

Let's explore the concept of laying out controls in the RelativeLayout container by writing an application. The application that we are going to create is a simple `Login Form` application that asks the user to enter a `User ID` and `Password`. The `TextView`, `EditText`, and `Button` controls in the application are laid out in a RelativeLayout container (see Figure 3.10—left). If either the `User ID` or `Password` is left blank, the message `The User ID or password is left blank. Please Try Again` is displayed. If the correct `User ID` and `Password`, in this case, `guest`, are entered, then a welcome message is displayed. Otherwise, the message `The User ID or password is incorrect. Please Try Again` is displayed.

So, let's create the application. Launch the Eclipse IDE and create a new Android application called `LoginForm`. Arrange four `TextView` controls, two `EditText` controls, and a `Button` control in RelativeLayout, as shown in the layout file `activity_login_form.xml` displayed in Listing 3.8.

LISTING 3.8 The `activity_login_form.xml` on Laying Out the `TextView`, `EditText`, and `Button` Controls in the RelativeLayout Container

```
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >
    <TextView
        android:id="@+id/sign_msg"
        android:text = "Sign In"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:typeface="serif"
        android:textSize="25dip"
        android:textStyle="bold"
        android:padding="10dip"
        android:layout_centerHorizontal="true"/>
    <TextView
        android:id="@+id/user_msg"
        android:text = "User ID:"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="10dip"
        android:layout_below="@+id/sign_msg" />
    <EditText
        android:id="@+id/user_ID"
        android:layout_height="wrap_content"
        android:layout_width="250dip"
        android:layout_below="@+id/sign_msg"
        android:layout_toRightOf="@+id/user_msg"
        android:singleLine="true" />
    <TextView
        android:id="@+id/password_msg"
        android:text = "Password:"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/user_msg"
        android:layout_margin="10dip"
        android:paddingTop="10dip"/>
    <EditText
        android:id="@+id/password"
        android:layout_height="wrap_content"
        android:layout_width="250dp"
        android:singleLine="true"
        android:layout_below="@+id/user_ID"
        android:layout_toRightOf="@+id/password_msg"
        android:password="true" />
    <Button
        android:id="@+id/login_button"
        android:text="Sign In"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_marginTop="10dip"
        android:layout_below="@+id/password_msg"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/response"
        android:layout_below="@+id/login_button"/>
</RelativeLayout>
```

The controls in the application are arranged in the RelativeLayout, as explained here:

- Through the `TextView` control `sign_msg`, the text `Sign In` is displayed horizontally centered at the top. It is displayed in bold serif font, `25 dip` in size. The text is padded with a space of `10dip` on all four sides of its container.
- Another `TextView` control, `user_msg`, displays the text `User ID` below the `TextView sign_msg`. The `TextView` is placed `10dip` from all four sides.
- An `EditText` control `user_ID` is displayed below `sign_msg` and to the right of `user_msg`. The width assigned to the `TextView` control is `250 dip` and is set to `single-line` mode, so if the user types beyond the given width, the text scrolls to accommodate extra text but does not run over to the second line.
- A `TextView password_msg` control displaying the text `Password:` is displayed below the `TextView user_msg`. The `TextView` control is placed at a spacing of `10dip` from all four sides, and the text `Password:` is displayed at `10dip` from the control's top boundary.
- An `EditText` control `password` is displayed below the `EditText user_ID` and to the right of the `TextView password_msg`. The `width` assigned to the `TextView` control is `250` dip and is set to `single-line` mode. In addition, the typed characters are converted into dots for security.
- A `Button` control `login_button` with the caption `Sign In` is displayed below the `TextView password_msg`. The button is horizontally centered and is set to appear at `10dip` distance from the `EditText` control `password`.
- A `TextView` control `response` is placed below the `Button login_button`. It is used to display messages to the user when the `Sign In` button is pressed after entering `User ID` and `Password`.

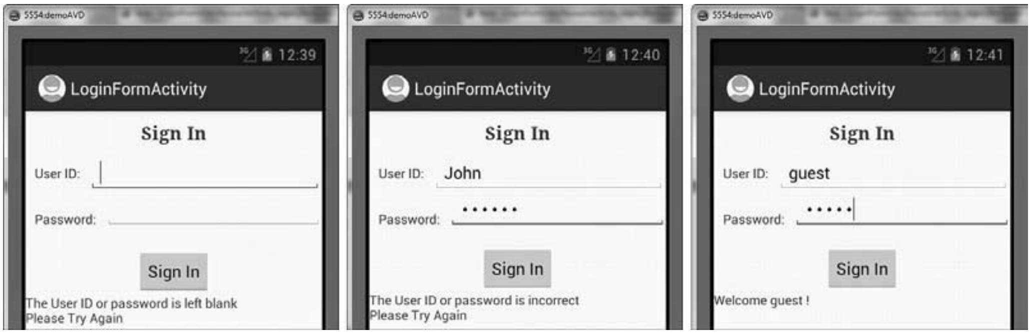
To authenticate the user, we need to access the `User ID` and `Password` that is entered and match these values against the valid `User ID` and `Password`. In addition, we want to validate the `EditText` controls to confirm that none of them is blank. We also want to welcome the user if he or she is authorized. To do all this, we write the code in the activity file `LoginFormActivity.java` as shown in Listing 3.9.

LISTING 3.9 Code Written in the Java Activity File `LoginFormActivity.java`

```java
package com.androidunleashed.loginform;

import android.app.Activity;
import android.os.Bundle;
import android.view.View.OnClickListener;
import android.widget.Button;
import android.widget.EditText;
import android.view.View;
import android.widget.TextView;

public class LoginFormActivity extends Activity implements OnClickListener  {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_login_form);
        Button b = (Button)this.findViewById(R.id.login_button);
        b.setOnClickListener(this);
    }

    public void onClick(View v) {
        EditText userid = (EditText) findViewById(R.id.user_ID);
        EditText password = (EditText) findViewById(R.id.password);
        TextView resp = (TextView)this.findViewById(R.id.response);
        String usr = userid.getText().toString();
        String pswd = password.getText().toString();
        if(usr.trim().length() == 0 || pswd.trim().length() == 0){
            String str = "The User ID or password is left blank \nPlease Try Again";
            resp.setText(str);
        }
        else{
            if(usr.equals("guest") && pswd.equals("guest")) resp.setText("Welcome " +
            usr+ " ! ");
           else resp.setText("The User ID or password is incorrect \nPlease Try Again");
        }
    }
}
```

The `Button` control is accessed from the layout file and is mapped to the `Button` object `b`. This activity implements the `OnClickListener` interface. Hence, the class implements the callback method `onClick()`, which is invoked when a click event occurs on the `Button` control.

In the `onClick()` method, the `user_ID` and `password` `EditText` controls are accessed from the layout file and mapped to the `EditText` objects `userid` and `password`. Also, the `TextView` control `response` is accessed from the layout file and is mapped to the `TextView` object `resp`. The `User ID` and `password` entered by the user in the two `EditText` controls are accessed through the objects `userid` and `password` and assigned to the two Strings `usr` and `pswd`, respectively. The data in the `usr` and `pswd` strings is checked for authentication. If the user has left any of the `EditText` controls blank, the message `The User ID or password is left blank. Please Try Again` is displayed, as shown in Figure 3.10 (left). If the `User ID` and `password` are correct, then a welcome message is displayed (see Figure 3.10—right). Otherwise, the message `The User ID or password is incorrect. Please Try Again` is displayed, as shown in Figure 3.10 (middle).



John

••••••

The User ID or password is left blank
Please Try Again

The User ID or password is incorrect
Please Try Again

guest

•••••

Welcome guest !

FIGURE 3.10 (left) The Login Form displays an error if fields are left blank, (middle) the Password Incorrect message displays if the user ID or password is incorrect, and (right) the Welcome message displays when the correct user ID and password are entered.

AbsoluteLayout

Each child in an AbsoluteLayout is given a specific location within the bounds of the container. Such fixed locations make AbsoluteLayout incompatible with devices of different screen size and resolution. The controls in AbsoluteLayout are laid out by specifying their exact *X* and *Y* positions. The coordinate 0,0 is the origin and is located at the top-left corner of the screen.

Let's write an application to see how controls are positioned in AbsoluteLayout. Create a new Android Project called `AbsoluteLayoutApp`. Modify its layout file, `activity_absolute_layout_app.xml`, as shown in Listing 3.10.

LISTING 3.10 The Layout File `activity_absolute_layout_app.xml` on Arranging Controls in the AbsoluteLayout Container

```
<AbsoluteLayout 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="wrap_content"
        android:layout_height="wrap_content"
        android:text="New Product Form"
        android:textSize="20sp"
        android.textStyle="bold"
        android:layout_x="90dip"
        android:layout_y="2dip"/>
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Product Code:"
        android:layout_x="5dip"
        android:layout_y="40dip" />
    <EditText
        android:id="@+id/product_code"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:minWidth="100dip"
        android:layout_x="110dip"
        android:layout_y="30dip" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Product Name:"
        android:layout_x="5dip"
        android:layout_y="90dip"/>
    <EditText
        android:id="@+id/product_name"
        android:layout_width="200dip"
        android:layout_height="wrap_content"
        android:minWidth="200dip"
        android:layout_x="110dip"
        android:layout_y="80dip"
        android:scrollHorizontally="true" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Product Price:"
        android:layout_x="5dip"
        android:layout_y="140dip" />
    <EditText
        android:id="@+id/product_price"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:minWidth="100dip"
        android:layout_x="110dip"
        android:layout_y="130dip" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/click_btn"
        android:text="Add New Product"
        android:layout_x="80dip"
        android:layout_y="190dip" />
</AbsoluteLayout>
```

The controls in `activity_absolute_layout_app.xml` are as follows:

- The `New Product Form TextView` is set to appear `90dip` from the left and `2dip` from the top side of the container. The size of the text is set to `20sp`, and its style is set to `bold`.
- The `Product Code TextView` is set to appear `5dip` from the left and `40dip` from the top side of the container.
- The `product_code EditText` control is set to appear `110dip` from the left and `30dip` from the top side of the container. The minimum width of the control is set to `100dp`.
- The `ProductName TextView` control is set to appear `5dip` from the left and `90dip` from the top side of the container.
- The `product_name EditText` control is set to appear `110dip` from the left and `80dip` from the top side of the container. The minimum width of the control is set to `200dip`, and its text is set to scroll horizontally when the user types beyond its width.
- The `Product Price TextView` is set to appear `5dip` from the left and `140dip` from the top side of the container.
- The `product_price EditText` control is set to appear `110dip` from the left and `130dip` from the top side of the container. The minimum width of the control is set to `100dip`.
- The `click_btn Button`, `Add New Product`, is set to appear `80dip` from the left and `190dip` from the top side of the container.

If we don't specify the x, y coordinates of a control in AbsoluteLayout, it is placed in the origin point, that is, at location 0,0. If the value of the x and y coordinates is too large, the control does not appear on the screen. The values of the x and y coordinates are specified in any units, such as `sp`, `in`, `mm`, and `pt`.

After specifying the locations of controls in the layout file `activity_absolute_layout_app.xml`, we can run the application. There is no need to make any changes in the file `AbsoluteLayoutAppActivity.java`. When the application is run, we get the output shown in Figure 3.11.

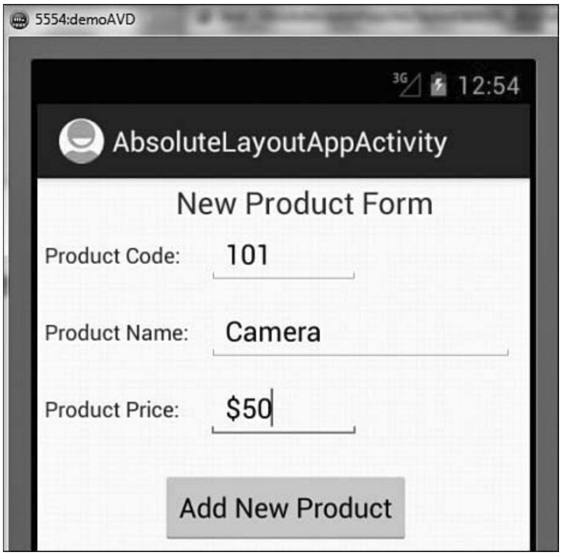


FIGURE 3.11 Different controls laid out in AbsoluteLayout

The AbsoluteLayout class is not used often, as it is not compatible with Android phones of different screen sizes and resolutions.

The next layout we are going to discuss is FrameLayout. Because we will learn to display images in FrameLayout, let's first take a look at the `ImageView` control that is often used to display images in Android applications.

Using `ImageView`

An `ImageView` control is used to display images in Android applications. An image can be displayed by assigning it to the `ImageView` control and including the `android:src` attribute in the XML definition of the control. Images can also be dynamically assigned to the `ImageView` control through Java code.

A sample ImageView tag when used in the layout file is shown here:

```
<ImageView
    android:id="@+id/first_image"
    android:src = "@drawable/bintupic"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:scaleType="fitXY"
    android:adjustViewBounds="true"
    android:maxHeight="100dip"
    android:maxWidth="250dip"
    android:minHeight="100dip"
    android:minWidth="250dip"
    android:resizeMode="horizontal|vertical" />
```

Almost all attributes that we see in this XML definition should be familiar, with the exception of the following ones:

- **`android:src`**—Used to assign the image from drawable resources. We discuss drawable resources in detail in Chapter 4. For now, assume that the image in the `res/drawable` folder is set to display through the `ImageView` control via this attribute.

  Example:

  ```
  android:src = "@drawable/bintupic"
  ```

  You do not need to specify the image file extension. JPG and GIF files are supported, but the preferred image format is PNG.

- **`android:scaleType`**—Used to scale an image to fit its container. The valid values for this attribute include `fitXY`, `center`, `centerInside`, and `fitCenter`. The value `fitXY` independently scales the image around the X and Y axes without maintaining the aspect ratio to match the size of container. The value `center` centers the image in the container without scaling it. The value `centerInside` scales the image uniformly, maintaining the aspect ratio so that the width and height of the image fit the size of its container. The value `fitCenter` scales the image while maintaining the aspect ratio, so that one of its X or Y axes fits the container.

- **`android:adjustViewBounds`**—If set to `true`, the attribute adjusts the bounds of the `ImageView` control to maintain the aspect ratio of the image displayed through it.

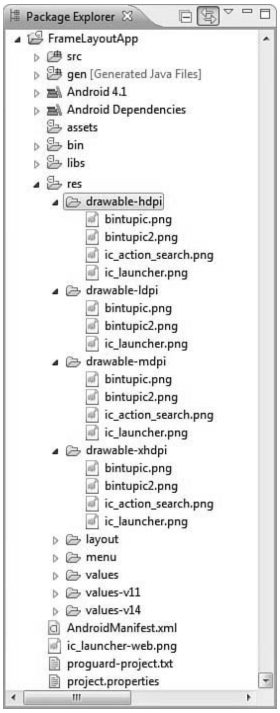
- **`android:resizeMode`**—The `resizeMode` attribute is used to make a control resizable so we can resize it horizontally, vertically, or around both axes. We need to click and hold the control to display its resize handles. The resize handles can be dragged in the desired direction to resize the control. The available values for the `resizeMode` attribute include `horizontal`, `vertical`, and `none`. The `horizontal` value resizes the control around the horizontal axis, the `vertical` value resizes around the vertical axis, the `both` value resizes around both the horizontal and vertical axes, and the value `none` prevents resizing.

FrameLayout

FrameLayout is used to display a single `View`. The `View` added to a FrameLayout is placed at the top-left edge of the layout. Any other `View` added to the FrameLayout overlaps the previous `View`; that is, each `View` stacks on top of the previous one. Let's create an application to see how controls can be laid out using FrameLayout.

In the application we are going to create, we will place two `ImageView` controls in the FrameLayout container. As expected, only one `ImageView` will be visible, as one `ImageView` will overlap the other `ImageView`, assuming both `ImageView` controls are of the same size. We will also display a button on the `ImageView`, which, when selected, displays the hidden `ImageView` underneath.

Let's start with the application. Create a new Android project called `FrameLayoutApp`. To display images in Android applications, the image is first copied into the `res/drawable` folder and from there, it is referred to in the layout and other XML files. We look at the procedure for displaying images, as well as the concept of drawable resources, in detail in Chapter 4. For the time being, it is enough to know that to enable the image(s) to be referred to in the layout files placed in the `res/drawable` folder, the image needs to exist in the `res/drawable` folder. There are four types of drawable folders: `drawable-xhdpi`, `drawable-hdpi`, `/res/drawable-mdpi`, and `/res/drawable-ldpi`. We have to place images of different resolutions and sizes in these folders. The graphics with the resolutions `320 dpi`, `240dpi`, `160 dpi`, and `120dpi` (`96 x 96 px`, `72 x 72 px`, `48 x 48 px`, and `36 x 36 px`), are stored in the `res/drawable-xhdpi`, `res/drawable-hdpi`, `res/drawable-mdpi`, and `res/drawable-ldpi` folders, respectively. The application picks up the appropriate graphic from the correct folder. So, if we copy two images called `bintupic.png` and `bintupic2.png` of the preceding size and resolution and paste them into the four `res/drawable` folders, the `Package Explorer` resembles Figure 3.12.



```
Package Explorer
FrameLayoutApp
  src
  gen [Generated Java Files]
  Android 4.1
  Android Dependencies
  assets
  bin
  libs
  res
    drawable-hdpi
      bintupic.png
      bintupic2.png
      ic_action_search.png
      ic_launcher.png
    drawable-ldpi
      bintupic.png
      bintupic2.png
      ic_launcher.png
    drawable-mdpi
      bintupic.png
      bintupic2.png
      ic_action_search.png
      ic_launcher.png
    drawable-xhdpi
      bintupic.png
      bintupic2.png
      ic_action_search.png
      ic_launcher.png
    layout
    menu
    values
    values-v11
    values-v14
  AndroidManifest.xml
  ic_launcher-web.png
  proguard-project.txt
  project.properties
```

FIGURE 3.12 The `Package Explorer` window showing the two images, `bintupic.png` and `bintupic2.png`, dropped into the `res/drawable` folders

To display two `ImageViews` and a `TextView` in the application, let's write the code in the layout file `activity_frame_layout_app.xml` as shown in Listing 3.11.

LISTING 3.11 The Layout File `activity_frame_layout_app.xml` on Arranging the `ImageView` and `TextView` Controls in the FrameLayout Container

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ImageView
        android:id="@+id/first_image"
        android:src = "@drawable/bintupic"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="fitXY" />
    <ImageView
        android:id="@+id/second_image"
        android:src = "@drawable/bintupic2"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:scaleType="fitXY" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Click the image to switch"
        android:layout_gravity="center_horizontal|bottom"
        android:padding="5dip"
        android:textColor="#ffffff"
        android:textStyle="bold"
        android:background="#333333"
        android:layout_marginBottom="10dip" />
</FrameLayout>
```

The `first_image` and `second_image ImageView` controls are set to display the images `bintupic.png` and `bintupic2.png`, respectively. To make the two images stretch to cover the entire screen, the `scaleType` attribute in the `ImageView` tag is set to `fitXY`. A `TextView`, `Click the image to switch`, is set to display at the horizontally centered position and at a distance of `10dip` from the bottom of the container. The spacing between the text and the boundary of the `TextView` control is set to `5dip`. The background of the text is set to a dark color, the foreground color is set to white, and its style is set to bold. When a user selects the current image on the screen, the image should switch to show the hidden image. For this to occur, we need to write code in the activity file as shown in Listing 3.12.

LISTING 3.12 Code Written in the Java Activity File `FrameLayoutAppActivity.java`

```java
package com.androidunleashed.framelayoutapp;

import android.app.Activity;
import android.os.Bundle;
import android.widget.ImageView;
import android.view.View.OnClickListener;
import android.view.View;

public class FrameLayoutAppActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_frame_layout_app);
        final ImageView first_image = (ImageView)this.findViewById(R.id.first_image);
        final ImageView second_image = (ImageView)this.findViewById(R.id.second_image);
        first_image.setOnClickListener(new OnClickListener(){
            public void onClick(View view) {
                second_image.setVisibility(View.VISIBLE);
                view.setVisibility(View.GONE);
            }
        });
        second_image.setOnClickListener(new OnClickListener(){
            public void onClick(View view) {
                first_image.setVisibility(View.VISIBLE);
                view.setVisibility(View.GONE);
            }
        });
    }
}
```

The two `first_image` and `second_image` `ImageView` controls are located through the `findViewById` method of the Activity class and assigned to the two `ImageView` objects, `first_image` and `second_image`, respectively. We register the click event by calling the `setOnClickListener()` method with an `OnClickListener`. An anonymous listener is created on the fly to handle click events for the `ImageView`. When the `ImageView` is clicked, the `onClick()` method of the listener is called. In the `onClick()` method, we switch the images; that is, we make the current `ImageView` invisible and the hidden `ImageView` visible. When the application runs, we see the output shown in Figure 3.13 (left). The application shows an image, and the other image is hidden behind it because in FrameLayout one View overlaps the other. When the user clicks the image, the images are switched, as shown in Figure 3.13 (right).

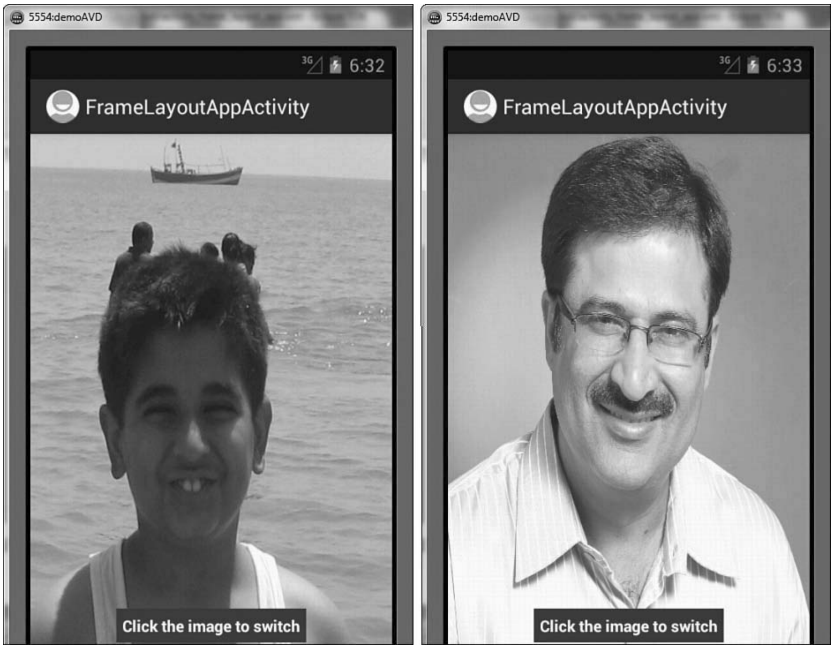


FIGURE 3.13 (left) An image and a `TextView` laid out in FrameLayout, and (right) the images switch when clicked

TableLayout

The TableLayout is used for arranging the enclosed controls into rows and columns. Each new row in the TableLayout is defined through a `TableRow` object. A row can have zero or more controls, where each control is called a `cell`. The number of columns in a TableLayout is determined by the maximum number of cells in any row. The width of a column is equal to the widest cell in that column. All elements are aligned in a column; that is, the width of all the controls increases if the width of any control in the column is increased.

NOTE

We can nest another TableLayout within a table cell, as well.

Operations Applicable to TableLayout

We can perform several operations on TableLayout columns, including stretching, shrinking, collapsing, and spanning columns.

Stretching Columns

The default width of a column is set equal to the width of the widest column, but we can stretch the column(s) to take up available free space using the `android:stretchColumns` attribute in the TableLayout. The value assigned to this attribute can be a single column number or a comma-delimited list of column numbers. The specified columns are stretched to take up any available space on the row.

Examples:

- **`android:stretchColumns="1"`**—The second column (because the column numbers are zero-based) is stretched to take up any available space in the row.
- **`android:stretchColumns="0,1"`**—Both the first and second columns are stretched to take up the available space in the row.
- **`android:stretchColumns="*"`**—All columns are stretched to take up the available space.

Shrinking Columns

We can shrink or reduce the width of the column(s) using the `android:shrinkColumns` attribute in the TableLayout. We can specify either a single column or a comma-delimited list of column numbers for this attribute. The content in the specified columns word-wraps to reduce their width.

NOTE

By default, the controls are not word-wrapped.

Examples:

- **`android:shrinkColumns="0"`**—The first column's width shrinks or reduces by word-wrapping its content.
- **`android:shrinkColumns="*"`**—The content of all columns is word-wrapped to shrink their widths.

Collapsing Columns

We can make the column(s) collapse or become invisible through the `android:collapseColumns` attribute in the TableLayout. We can specify one or more comma-delimited columns for this attribute. These columns are part of the table information but are invisible. We can also make column(s) visible and invisible through coding by passing the `Boolean` values `false` and `true`, respectively, to the `setColumnCollapsed()` method in the TableLayout. For example:

- **`android:collapseColumns="0"`**—The first column appears collapsed; that is, it is part of the table but is invisible. It can be made visible through coding by using the `setColumnCollapsed()` method.