Introducing Layouts

Layout Managers (more generally, "layouts") are extensions of the ViewGroup class designed to control the position of child controls on a screen. Layouts can be nested, letting you create arbitrarily complex interfaces using a combination of Layout Managers.

The Android SDK includes some simple layouts to help you construct your UI. It's up to you to select the right combination of layouts to make your interface easy to understand and use.

The following list includes some of the more versatile layout classes available:

❑ **FrameLayout** The simplest of the Layout Managers, the *Frame Layout* simply pins each child view to the top left corner. Adding multiple children stacks each new child on top of the previous, with each new View obscuring the last.

❑ **LinearLayout** A *Linear Layout* adds each child View in a straight line, either vertically or horizontally. A vertical layout has one child View per row, while a horizontal layout has a single row of Views. The Linear Layout Manager allows you to specify a "weight" for each child View that controls the relative size of each within the available space.

❑ **RelativeLayout** Using the *Relative Layout*, you can defi ne the positions of each of the child Views relative to each other and the screen boundaries.

❑ **TableLayout** The *Table Layout* lets you lay out Views using a grid of rows and columns. Tables can span multiple rows and columns, and columns can be set to shrink or grow.

❑ **AbsoluteLayout** In an *Absolute Layout*, each child View's position is defi ned in absolute coordinates. Using this class, you can guarantee the exact layout of your components, but at a price. Compared to the previous managers, describing a layout in absolute terms means that your layout can't dynamically adjust for different screen resolutions and orientations.

The Android documentation describes the features and properties of each layout class in detail, so rather than repeating it here, I'll refer you to http://code.google.com/android/devel/ui/layout.html.

Later in this chapter, you'll also learn how to create compound controls (widgets made up of several interconnected Views) by extending these layout classes.

Using Layouts

The preferred way to implement layouts is in XML using external resources. A layout XML must contain a single root element. This root node can contain as many nested layouts and Views as necessary to construct an arbitrarily complex screen.

The following XML snippet shows a simple layout that places a TextView above an EditText control using a LinearLayout confi gured to lay out vertically:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="vertical"
android:layout_width="fill_parent"
android:layout_height="fill_parent">
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Enter Text Below"
/>
<EditText
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="Text Goes Here!"
/>
</LinearLayout>
```

Implementing layouts in XML decouples the presentation layer from View and Activity code. It also lets you create hardware specific variations that are dynamically loaded without requiring code changes.

When it's preferred, or required, you can implement layouts in code. When assigning Views to layouts, it's important to apply LayoutParameters using the setLayoutParams method, or passing them in to

the addView call as shown below:

```
LinearLayout ll = new LinearLayout(this);
ll.setOrientation(LinearLayout.VERTICAL);
TextView myTextView = new TextView(this);
EditText myEditText = new EditText(this);
myTextView.setText(“Enter Text Below”);
myEditText.setText(“Text Goes Here!”);
int lHeight = LinearLayout.LayoutParams.FILL_PARENT;
int lWidth = LinearLayout.LayoutParams.WRAP_CONTENT;
ll.addView(myTextView, new LinearLayout.LayoutParams(lHeight, lWidth));
ll.addView(myEditText, new LinearLayout.LayoutParams(lHeight, lWidth));
setContentView(ll);
```