MODULE III

USER INTERFACES DEVELOPMENT IN ANDROID

UI development in Android is fun. It's fun because it's relatively easy. With Android, we have a simple-to-understand framework with a limited set of out-of-the-box controls. The available screen area is generally limited. Android also takes care of a lot of the heavy lifting normally associated to designing and building quality UIs. This, combined with the fact that the user usually wants to do one specific action, allows us to easily build a good UI to deliver a good user experience. The Android SDK provides text fields, buttons, lists, grids, and so on. In addition, Android provides a collection of controls that are appropriate for mobile devices.

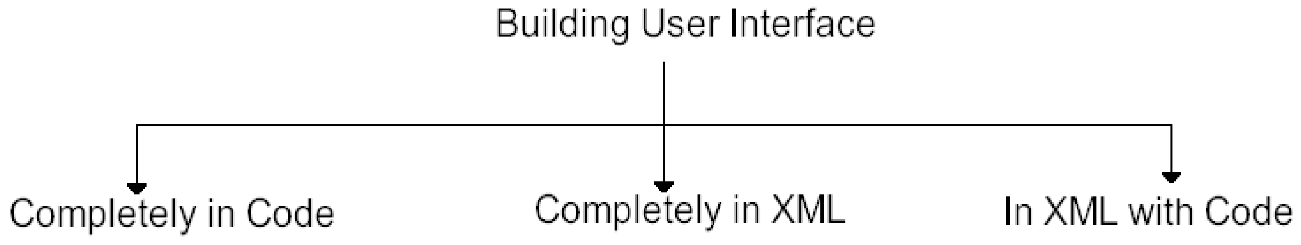
At the heart of the common controls are two classes: android.view.View and android.view.ViewGroup. As the name of the first class suggests, the View class represents a general-purpose View object. The common controls in Android ultimately extend the View class. ViewGroup is also a view, but it contains other views too. ViewGroup is the base class for a list of layout classes. Android, like Swing, uses the concept of *layouts* to manage how controls are laid out within a container view. Using layouts, as we'll see, makes it easy for us to control the position and orientation of the controls in our UIs.

We can construct UIs entirely in code. We can also define UIs in XML. We can even combine the two define the UI in XML and then refer to it, and modify it, in code. We will find the terms *view*, *control*, *widget*, *container*, and *layout* in discussions regarding UI development. If we are new to Android programming or UI development in general, we might not be familiar with these terms. We'll briefly describe them before we get started (see Table 6–1).

Table 6–1. *UI Nomenclature*

Term

	Description
View, widget, control	Each of these represents a UI element. Examples include a button, a grid, a list, a window, a dialog box, and so on. The terms *view*, *widget*, and *control* are used interchangeably in this chapter.
Container	This is a view used to contain other views. For example, a grid can be considered a container because it contains cells, each of which is a view.
Layout	This is a visual arrangement of containers and views and can include other layouts.



BUILDING UI COMPLETELY IN CODE

The first example, Listing 6–1, demonstrates how to build the UI entirely in code. To try this, create a new Android project with an activity named MainActivity and then copy the code from Listing 6–1 into our MainActivity class.

Listing 6–1. *Creating a Simple User Interface Entirely in Code*

```
package com.androidbook.controls;
import android.app.Activity;
import android.os.Bundle;
import android.view.ViewGroup.LayoutParams;
import android.widget.LinearLayout;
import android.widget.TextView;
public class MainActivity extends Activity
{
        private LinearLayout nameContainer;
        private LinearLayout addressContainer;
        private LinearLayout parentContainer;
        /** Called when the activity is first created. */
        @Override
        public void onCreate(Bundle savedInstanceState)
        {
                super.onCreate(savedInstanceState);
                createNameContainer();
                createAddressContainer();
                createParentContainer();
                setContentView(parentContainer);
        }
        private void createNameContainer()
        {
                nameContainer = new LinearLayout(this);
                nameContainer.setLayoutParams(new
                                LayoutParams(LayoutParams.FILL_PARENT,
                                LayoutParams.WRAP_CONTENT));
                nameContainer.setOrientation(LinearLayout.HORIZONTAL);
                TextView nameLbl = new TextView(this);
                nameLbl.setText("Name: ");
                TextView nameValue = new TextView(this);
                nameValue.setText("John Doe");
                nameContainer.addView(nameLbl);
                nameContainer.addView(nameValue);
        }
        private void createAddressContainer()
        {
                addressContainer = new LinearLayout(this);
                addressContainer.setLayoutParams(new
                              LayoutParams(LayoutParams.FILL_PARENT,
                    LayoutParams.WRAP_CONTENT));
            addressContainer.setOrientation(LinearLayout.VERTICAL);
            TextView addrLbl = new TextView(this);
            addrLbl.setText("Address:");
            TextView addrValue = new TextView(this);
            addrValue.setText("911 Hollywood Blvd");
            addressContainer.addView(addrLbl);
            addressContainer.addView(addrValue);
        }
        private void createParentContainer()
        {
            parentContainer = new LinearLayout(this);
            parentContainer.setLayoutParams(new
            LayoutParams(LayoutParams.FILL_PARENT,
            LayoutParams.FILL_PARENT));
            parentContainer.setOrientation(LinearLayout.VERTICAL);
            parentContainer.addView(nameContainer);
            parentContainer.addView(addressContainer);
        }
}
```

As shown in Listing 6–1, the activity contains three LinearLayout objects. As we mentioned earlier, layout objects contain logic to position objects within a portion of the screen. A LinearLayout, for example, knows how to lay out controls either vertically or horizontally. Layouts objects can contain any type of view even other layouts. The nameContainer object contains two TextView controls: one for the label Name: and the other to hold the actual name (such as John Doe). The addressContainer also contains two TextView controls. The difference between the two containers is that the nameContainer is laid out horizontally and the addressContainer is laid out vertically. Both of these containers live within the parentContainer, which is the root view of the activity.

After the containers have been built, the activity sets the content of the view to the root view by calling setContentView(parentContainer). When it comes time to render the UI of the activity, the root view is called to render itself. The root view then calls its children to render themselves, and the child controls call their children, and so on, until the entire UI is rendered. As shown in Listing 6–1, we have several LinearLayout controls. Two of them are laid out vertically, and one is laid out horizontally. The nameContainer is laid out horizontally. This means the two TextView controls appear side by side horizontally. The addressContainer is laid out vertically, which means the two TextView controls are stacked one on top of the other. The parentContainer is also laid out vertically, which is why the nameContainer appears above the addressContainer. Note a subtle difference between the two vertically laid-out containers, addressContainer and parentContainer. parentContainer is set to take up the entire width and height of the screen:

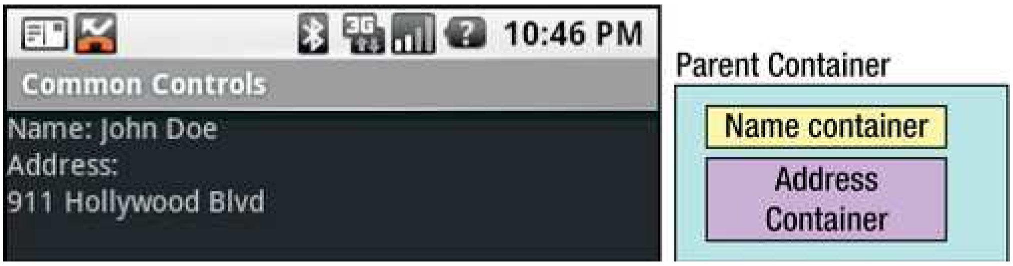
```
parentContainer.setLayoutParams(new
        LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT));
```

And addressContainer wraps its content vertically:

```
addressContainer.setLayoutParams(new
     LayoutParams(LayoutParams.FILL_PARENT,
LayoutParams.WRAP_CONTENT));
```

Said another way, WRAP_CONTENT means the view should take just the space it needs in that dimension and no more, up to what the containing view will allow. For the addressContainer, this means the container will take two lines vertically, because that's all it needs.

Figure 6–1



10:46 PM

Common Controls

Name: John Doe
Address:
911 Hollywood Blvd

BUILDING UI USING XML

XML layout files are stored under the resources (/res/) directory in a folder called layout. By default, we will get an XML layout file named main.xml, located under the res/layout folder. Double-click main.xml to see the contents. Eclipse will display a visual editor for our layout file. We probably have a string at the top of the view that says "Hello World, MainActivity!" or something like that. Click the main.xml tab at the bottom of the view to see the XML of the main.xml file. This reveals a LinearLayout and a TextView control. Using either the Layout or main.xml tab, or both, re-create Listing 6–2 in the main.xml file. Save it.

Listing 6–2. *Creating a User Interface Entirely in XML*

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <!-- NAME CONTAINER -->
        <LinearLayout
                xmlns:android="http://schemas.android.com/apk/res/android"
                android:orientation="horizontal" android:layout_width="fill_parent"
                android:layout_height="wrap_content">

                        <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="Name:" />

              <TextView android:layout_width="wrap_content"
        android:layout_height="wrap_content" android:text="John Doe" />
  </LinearLayout>

  <!-- ADDRESS CONTAINER -->
  <LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="wrap_content">

              <TextView android:layout_width="fill_parent"
        android:layout_height="wrap_content" android:text="Address:" />

              <TextView android:layout_width="fill_parent"
  android:layout_height="wrap_content" android:text="911 Hollywood Blvd."


/>
      </LinearLayout>
</LinearLayout>
```

Under our new project's src directory, there is a default .java file containing an Activity class definition. Double-click that file to see its contents. Notice the statement setContentView(R.layout.main). The XML snippet shown in Listing 6–2, combined with a call to setContentView(R.layout.main), will render the same UI as before when we generated it completely in code. The XML file is self-explanatory, but note that we have three container views defined. The first LinearLayout is the equivalent of our parent container. This container sets its orientation to vertical by setting the corresponding property like this: android:orientation="vertical". The parent container contains two LinearLayout containers, which represent nameContainer and addressContainer. Running this application will produce the same UI as our previous example application. The labels and values will be displayed as shown in Figure 6–1.

BUILDING UI IN XML WITH CODE

Listing 6–2 is a contrived example. It doesn't make any sense to hard-code the values of the TextView controls in the XML layout. Ideally, we should design our UIs in XML and then reference the controls from code. This approach enables us to bind dynamic data to the controls defined at design time. In fact, this is the recommended approach. It is fairly easy to build layouts in XML and then use code to populate the dynamic data. Listing 6–3 shows the same UI with slightly different XML. This XML assigns IDs to the TextView controls so that we can refer to them in code.

Listing 6–3. *Creating a User Interface in XML with IDs*

```
<?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">
<!-- NAME CONTAINER -->
<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="horizontal" android:layout_width="fill_parent"
        android:layout_height="wrap_content">
        <TextView android:layout_width="wrap_content"
android:layout_height="wrap_content" android:text="@string/name_text" />

        <TextView android:id="@+id/nameValue"
android:layout_width="wrap_content"
android:layout_height="wrap_content"/>
</LinearLayout>

<LinearLayout
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="wrap_content">


      <TextView android:layout_width="fill_parent"
      android:layout_height="wrap_content" android:text="@string/addr_text" />

      <TextView android:id="@+id/addrValue"
      android:layout_width="fill_parent" android:layout_height="wrap_content" />
      </LinearLayout>
</LinearLayout>
```

In addition to adding the IDs to the TextView controls that we want to populate from code, we also have label TextView controls that we're populating with text from our strings resource file. These are the TextViews without IDs that have an android:text attribute. As we may recall from Chapter 3, the actual strings for these TextViews will come from our strings.xml file in the /res/values folder. The developers of Android have done a nice job of making just about every aspect of a control settable via XML or code. It's usually a good idea to set the control's attributes in the XML layout file rather than using code. However, there will be lots of times when we need to use code, such as setting a value to be displayed to the user.

FILL_PARENT vs. MATCH_PARENT

The constant FILL_PARENT was deprecated in Android 2.2 and replaced with MATCH_PARENT. This was strictly a name change, though. The value of this constant is still -1. Similarly, for XML layouts, fill_parent was replaced with match_parent. So what value do we use? Instead of FILL_PARENT or MATCH_PARENT, we could simply use the value -1, and you'd be fine. However, this isn't very easy to read, and we don't have an equivalent unnamed value to use with our XML layouts. There's a better way.

ANDROID'S COMMON CONTROLS

We will now start our discussion of the common controls in the Android SDK. We’ll start with text controls and then cover buttons, check boxes, radio buttons, lists, grids, date and time controls, and a map-view control. We will also talk about layout controls.

TEXT CONTROLS

Text controls are likely to be the first type of control that we’ll work with in Android. Android has a complete but not overwhelming set of text controls. In this section, we are going to discuss the TextView, EditText, AutoCompleteTextView, and MultiCompleteTextView controls. Figure 6–2 shows the controls in action.

TextView:-

A TextView displays text to the user and optionally allows them to edit it. It is a complete text editor but the basic class is not allowing to editing. The TextView control knows how to display text but does not allow editing. This might lead us to conclude that the control is essentially a dummy label. We can set the autoLink property to emai or web, and the control will find and highlight any e-mail addresses and [redacted] is utilizing the

android.text.util.Linkify class. The main attributes are id, hint, maxHeight, maxWidth, password, minHeight, minWidth, text, phoneNumber, textColor, textStyle etc.

EditText:-

The EditText control is a subclass of TextView. As suggested by the name, the EditText control allows for text editing. EditText is not as powerful as the text-editing controls that we find on the Internet, but users of Android-based devices probably won’t type documents they’ll type a couple paragraphs at most. Therefore, the class has limited but appropriate functionality and may even surprise us. For example, one of the most significant properties of an EditText is the inputType. We can set the inputType property to textAutoCorrect have the control correct common misspellings. We can set it to textCapWords to have the control capitalize words. Other options expect only phone numbers or passwords. The old default behavior of the EditText control is to display text on one line and expand as needed. The user to a single line by setting the singleLine property to true. The user will have to continue typing on the same line. With inputType, if we don’t specify textMultiLine, the EditText will default to single-line only. So if we want the old default behavior of multiline typing, we need to specify inputType with textMultiLine.

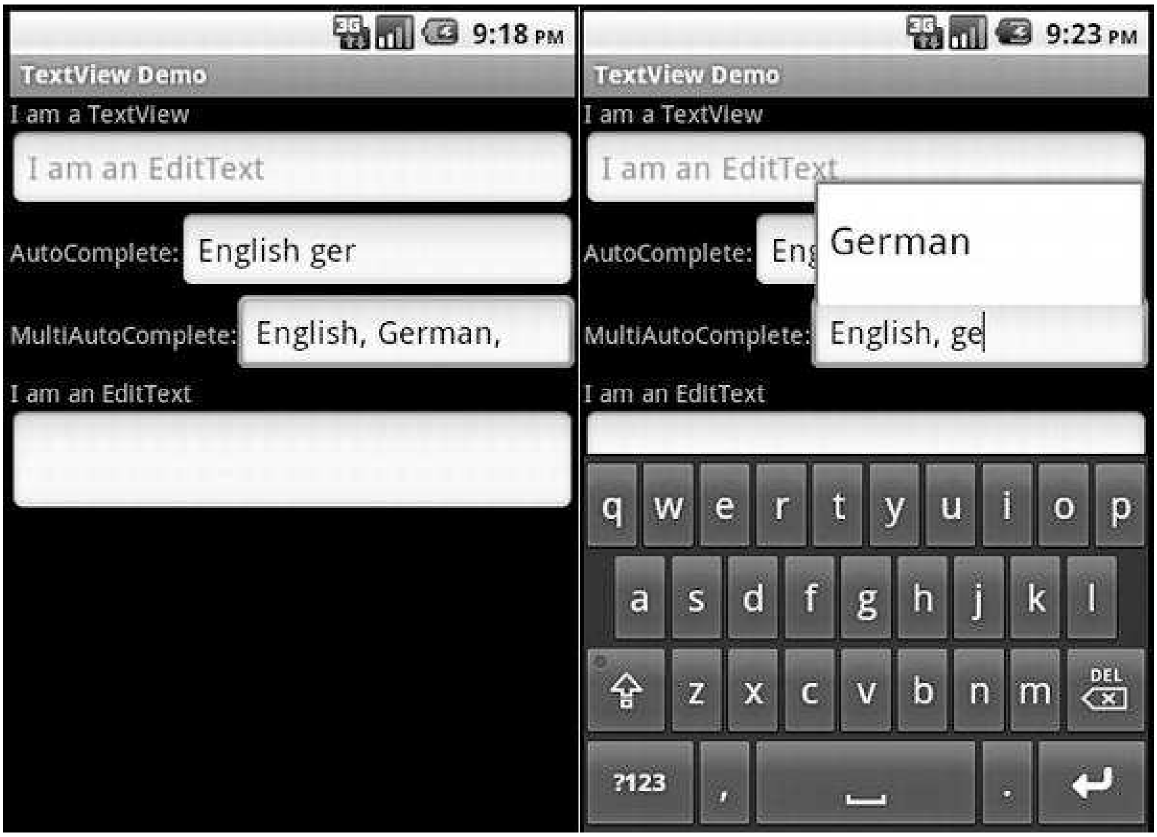
AutoCompleteTextView:-

The AutoCompleteTextView control is a TextView with auto-complete functionality. In other words, as the user types in the TextView, the control can display suggestions for selection. For example, if the user types **en**, the control suggests English. If the user types **gr**, the control recommends Greek, and so on.

MultiAutoCompleteTextView:-

The control offers suggestions only for the *entire* text in the text view. In other words, if we type a sentence, we don't get suggestions for each word. That's where MultiAutoCompleteTextView comes in. We can use the MultiAutoCompleteTextView to provide suggestions as the user types. For example, the user typed the word **English** followed by a comma, and then **Ge**, at which point the control suggested **German**. If the user were to continue, the control would offer additional suggestions. Using the MultiAutoCompleteTextView is like using the AutoCompleteTextView. The difference is that we have to tell the control where to start suggesting again. For example, in Figure 6–2, we can see that the control can offer suggestions at the beginning of the sentence and after it sees a comma. The MultiAutoCompleteTextView control requires that we give it a tokenizer that can parse the sentence and tell it whether to start suggesting again. Listing 6–8 demonstrates using the MultiAutoCompleteTextView control with the XML and then the Java code.

Figure 6–2. *Text controls in Android*



9:18 PM

TextView Demo

I am a TextView

I am an EditText

AutoComplete: English ger

MultiAutoComplete: English, German,

I am an EditText

9:23 PM

TextView Demo

I am a TextView

I am an EditText

AutoComplete: Eng

German

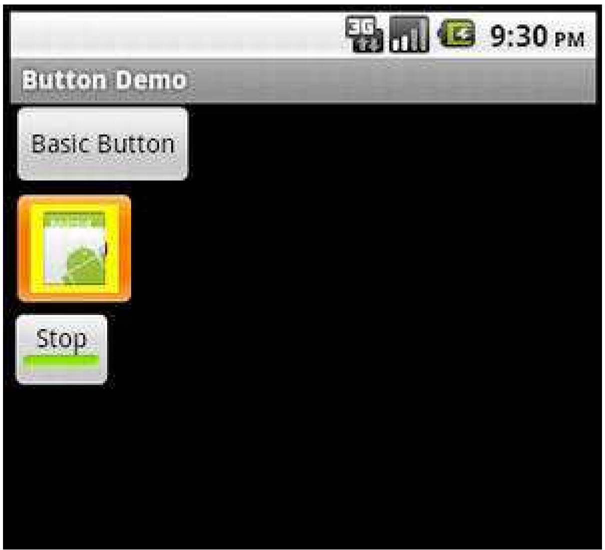
MultiAutoComplete: English, ge

I am an EditText

BUTTON CONTROLS

Buttons are common in any widget toolkit, and Android is no exception. Android offers the typical set of buttons as well as a few extras. We will discuss three types of button controls: the basic button, the image button, and the toggle button. The figure shows a UI with these controls. The button at the top is the basic button, the middle button is an image button, and the last one is a toggle button.

The Basic Button Control:-



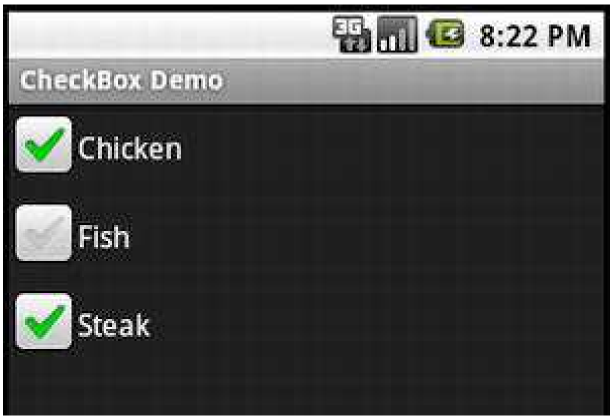
The basic button class in Android is android.widget.Button. There's not much to this type of button, beyond how we use it to handle click events. We register for the on-click event by calling the setOnClickListener() method with an OnClickListener. When the button is clicked, the onClick() method of the listener is called. The handler method will be called with target set to the View object representing the button that was clicked. Notice how the switch statement in the click handler method uses the resource IDs of the buttons to select the logic to run. Using this method means we won't have to explicitly create each Button object in our code, and we can reuse the same method across multiple buttons. This makes things easier to understand and maintain. This works with the other button types as well.

The ImageButton Control:-

Android provides an image button via android.widget.ImageButton. Using an image button is similar to using the basic button. The image file for the button must exist under /res/drawable. In our case, we're simply reusing the Android icon for the button. Note that we only need to do one or the other. We don't need to specify the button image in both the XML file and in code. One of the nice features of an image button is that we can specify a transparent background for the button. The result will be a clickable image that acts like a button but can look like whatever we want it to look like. Just set android:background="@null" for the image button.

The ToggleButton Control:-

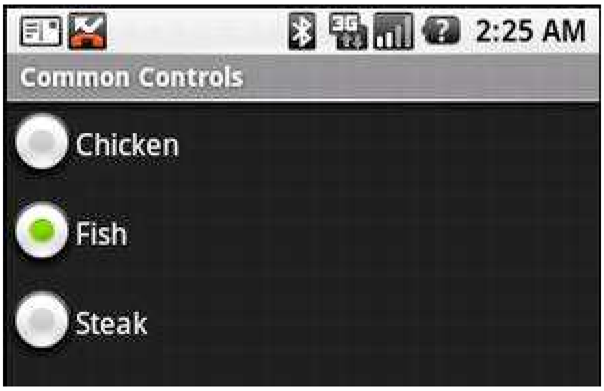
The ToggleButton control, like a check box or a radio button, is a two-state button. This button can be in either the ON or OFF state. As shown in Figure 6–3, the ToggleButton's default behavior is to show a green bar when in the ON state and a grayed-out bar when in the OFF state. Moreover, the default behavior also sets the button's text to ON when it's in the ON state and OFF when it's in the OFF state. We can modify the text for the ToggleButton if ON/OFF is not appropriate for our application. For example, if we have a background process that we want to start and stop via a ToggleButton, we could set the button's text to Stop and Run by using android:textOn and android:textOff properties.



CHECKBOX CONTROL:-

The CheckBox control is another two-state button that allows the user to toggle its state. The difference is that, for many situations, the users don't view it as a button that invokes immediate action. From Android's point of view, however, it is a button, and we can do anything with a check box that we can do with a button. We manage the state of a check box by calling setChecked() or toggle(). We can obtain the state by calling isChecked().If we need to implement specific logic when a check box is checked or unchecked, we can register for the on-checked event by calling setOnCheckedChangeListener() with an implementation of the OnCheckedChangeListener interface. We'll then have to implement the onCheckedChanged() method, which will be called when the check box is checked or unchecked. The nice part of setting up the OnCheckedChangeListener is that we are passed the new state of the CheckBox button. We could instead use the OnClickListener technique as we used with basic buttons. When the onClick() method is called, we would need to determine the new state of the button by casting it appropriately and then calling isChecked() on it.

RADIO BUTTON CONTROLS



RadioButton controls are an integral part of any UI toolkit. A radio button gives the users several choices and forces them to select a single item. To enforce this single-selection model, radio buttons generally belong to a group, and each group is forced to have only one item selected at a time. To create a group of radio buttons in Android, first create a RadioGroup, and then populate the group with radio buttons. Note that the radio buttons within the radio group are, by default, unchecked to begin with, although we can set one to checked in the XML definition. To set one of the radio buttons to the checked state programmatically, we can obtain a reference to the radio button and call setChecked().We can also use the toggle() method to toggle the state of the radio button. As withthe CheckBox control, we will be notified of on-checked or on-unchecked events if we call the setOnCheckedChangeListener() with an implementation of the OnCheckedChangeListener interface. There is a slight difference here, though. This is a different class than before. This time, it's technically the RadioGroup.OnCheckedChangeListener class, whereas before it was the CompoundButton.OnCheckedChangeListener class. The RadioGroup can also contain views other than the radio button.

IMAGE VIEW CONTROL

This is used to display an image, where the image can come from a file, a content provider, or a resource such as a drawable. We can even specify just a color, and the ImageView will display that color. Listing 6–21 shows some

XML examples of ImageViews, followed by some code that shows how to create an ImageView.

Listing 6–21. *ImageViews in XML and in Code*

```
<ImageView android:id="@+id/image1"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/icon" />

<ImageView android:id="@+id/image2"
        android:layout_width="125dip" android:layout_height="25dip"
        android:src="#555555" />

<ImageView android:id="@+id/image3"
        android:layout_width="wrap_content"
android:layout_height="wrap_content"/>

<ImageView android:id="@+id/image4"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/manatee02"
        android:scaleType="centerInside"
        android:maxWidth="35dip" android:maxHeight="50dip" />

ImageView imgView = (ImageView)findViewById(R.id.image3);
imgView.setImageResource( R.drawable.icon );
imgView.setImageBitmap(BitmapFactory.decodeResource(
        this.getResources(), R.drawable.manatee14) );
imgView.setImageDrawable(
Drawable.createFromPath("/mnt/sdcard/dave2.jpg") );
imgView.setImageURI(Uri.parse("file://mnt/sdcard/dave2.jpg"));
```

In this example, we have four images defined in XML. The first is simply the icon for our application. The second is a gray bar that is wider than it is tall. The third definition does not specify an image source in the XML, but we associate an ID with this one (image3) that we can use from our code to set the image. The fourth image is another of our drawable image files where we not only specify the source of the image file but also set the maximum dimensions of the image on the screen and define what to do if the image is larger than our maximum size. In this case, we tell the ImageView to center and scale the image so it fits inside the size we specified.

In the Java code of Listing 6–21 we show several ways to set the image of image3. We first of course must get a reference to the ImageView by finding it using its resource ID. The first setter method, setImageResource(), simply uses the image's resource ID to locate the image file to supply the image for our ImageView. The second setter uses the BitmapFactory to read in an image resource into a Bitmap object and then sets the ImageView to that Bitmap. Note that we could have done some modifications to the Bitmap before applying it to our ImageView, but in our case, we used it as is. In addition, the BitmapFactory has several methods of creating a Bitmap, including from a byte array and an InputStream. We could use the InputStream method to read an image from a web server, create the Bitmap image, and then set the ImageView from there.

The third setting uses a Drawable for our image source. In this case, we're showing the source of the image coming from the SD card. We 'll need to put some sort of image file out on the SD card with the proper name for this to work for us. Similar to BitmapFactory, the Drawable class has a few different ways to construct Drawables, including from an XML stream.

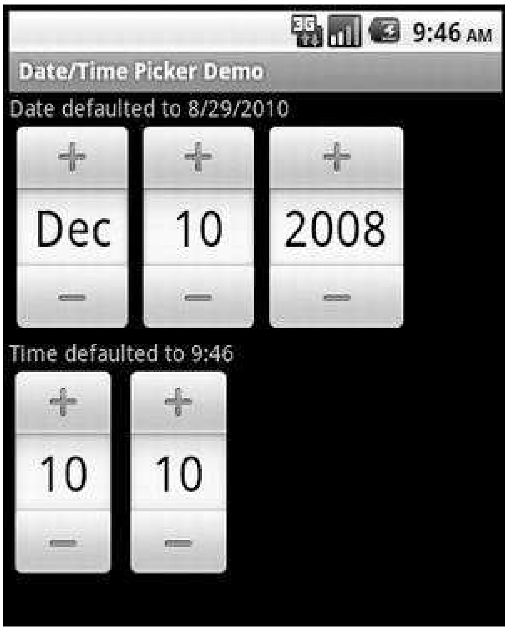
The final setter method takes the URI of an image file and uses that as the image source. For this last call, don't think that we can use any image URI as the source. This method is really only intended to be used for local images on the device, not for images that we might find through HTTP. To use Internet-based images as the source for our ImageView, we 'd most likely use BitmapFactory and an InputStream.

DATE AND TIME CONTROLS

Date and time controls are common in many widget tool kits. Android offers several date- and time-based controls. We are going to introduce the DatePicker, TimePicker, DigitalClock, and AnalogClock controls.

The DatePicker and TimePicker Controls:-

As the names suggest, we use the DatePicker control to select a date and the TimePicker control to pick a time. If we look at the XML layout, we can see that defining these controls is easy. As with any other control in the Android toolkit, we can access the controls programmatically to initialize them or to retrieve data from them. Note that for the month, the internal value is zero-based, which means that January is 0 and December is 11. For the TimePicker, the number of hours and minutes is set to 10. Note also that this control supports 24–hour view. If we do not set values for these controls, the default values will be the current date and time as known to the device. Finally, note that Android offers versions of these controls as modal windows, such as DatePickerDialog and TimePickerDialog. These controls are useful if we want to display the control to the user and force the user to make a selection.



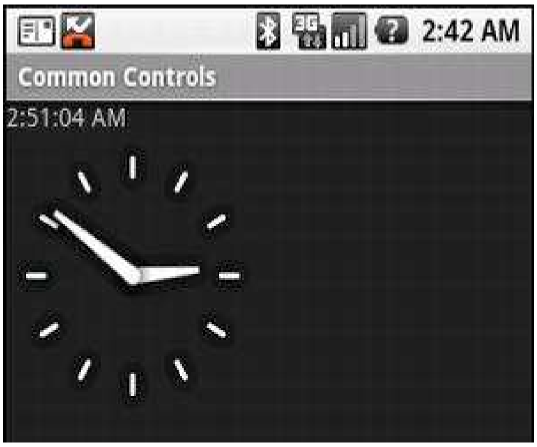
Date defaulted to 8/29/2010

Dec 10 2008

Time defaulted to 9:46

10 10

The DigitalClock and AnalogClock Controls:-



2:51:04 AM

Android also offers DigitalClock and AnalogClock controls. As shown, the digital clock supports seconds in addition to hours and minutes. The analog clock in Android is a two-handed clock, with one hand for the hour indicator and the other hand for the minute indicator. These two controls are really just for displaying the current time, as they don't let we modify the date or time. In other words, they are controls whose only capability is to display the current time. Thus, if we want to change the date or time, we'll need to stick to the DatePicker/TimePicker or DatePickerDialog/TimePickerDialog. The nice part about these two clocks, though, is that they will update themselves without having to do anything. That is, the seconds tick away in the DigitalClock, and the hands move on the AnalogClock without anything extra from us.

MAP VIEW CONTROL

The com.google.android.maps.MapView control can display a map. We can instantiate this control either via XML layout or code, but the activity that uses it must extend MapActivity. MapActivity takes care of multithreading requests to load a map, perform caching, and so on. Listing 6–25 shows an example instantiation of a MapView.

Listing 6–25. *Creating a MapView Control via XML Layout*

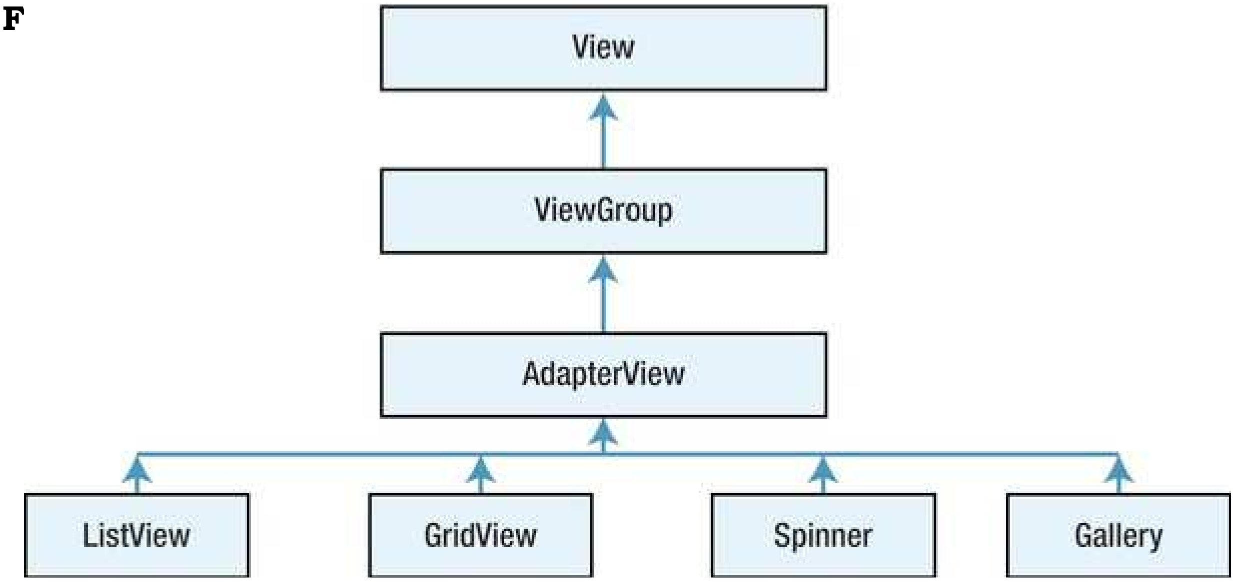
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:orientation="vertical" android:layout_width="fill_parent"
        android:layout_height="fill_parent">

        <com.google.android.maps.MapView
                android:layout_width="fill_parent"
                android:layout_height="fill_parent"
                android:enabled="true"
                android:clickable="true"
                android:apiKey="myAPIKey"/>
</LinearLayout>
```

UNDERSTANDING ADAPTERS

List controls are used to display collections of data. But instead of using a single type of control to manage both the display and the data, Android separates these two responsibilities into list controls and adapters. List controls are classes that extend android.widget.AdapterView and include ListView, GridView, Spinner, and Gallery.

F



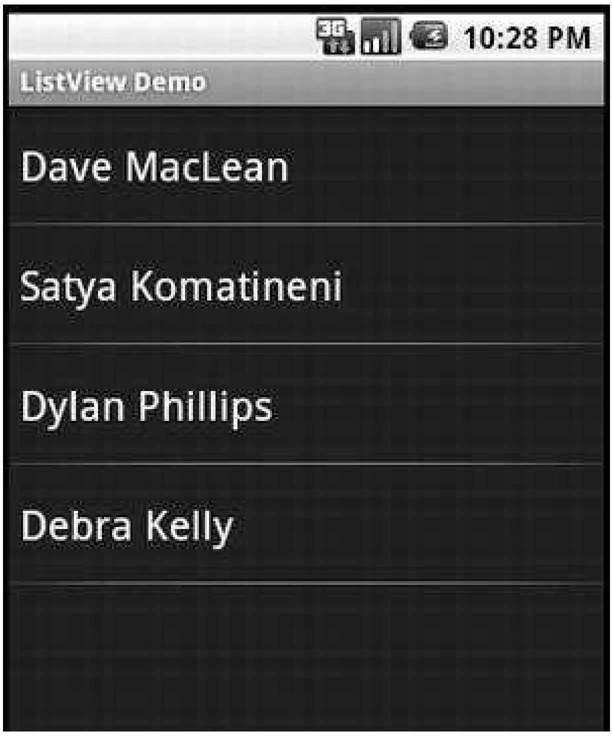
AdapterView itself extends android.widget.ViewGroup, which means that ListView, GridView, and so on are container controls. In other words, list controls contain collections of child views. The purpose of an adapter is to manage the data for an AdapterView and to provide the child views for it. Let's see how this works by examining the SimpleCursorAdapter.

ADAPTER VIEWS

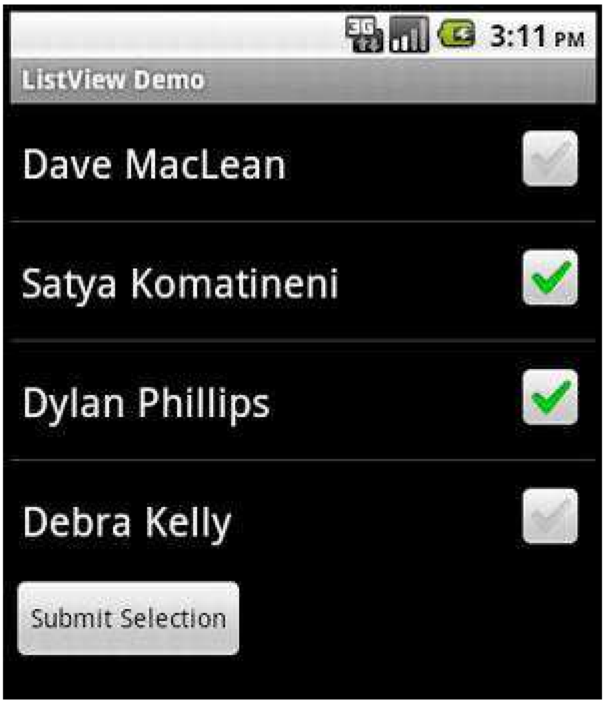
Now that we've been introduced to adapters, it is time to put them to work for us, providing data for list controls. In this section, we're going to first cover the basic list control, the ListView. Then, we'll describe how to create our own custom adapter, and finally, we'll describe the other types of list controls: GridViews, spinners, and the gallery.

LIST VIEW

The ListView control displays a list of items vertically. That is, if we've got a list of items to view and the number of items extends beyond what we can currently see in the display, we can scroll to see the rest of the items. We generally use a ListView by writing a new activity that extends android.app.ListActivity. ListActivity contains a ListView, and we set the data for the ListView by calling the setListAdapter() method. As we described previously, adapters link list controls to the data and help prepare the child views for the list control. Items in a ListView can be clicked to take immediate action or selected to act on the set of selected



items later. We're going to start really simple and then add functionality as we go.



ListView Demo

Dave MacLean □

Satya Komatineni ☑

Dylan Phillips ☑

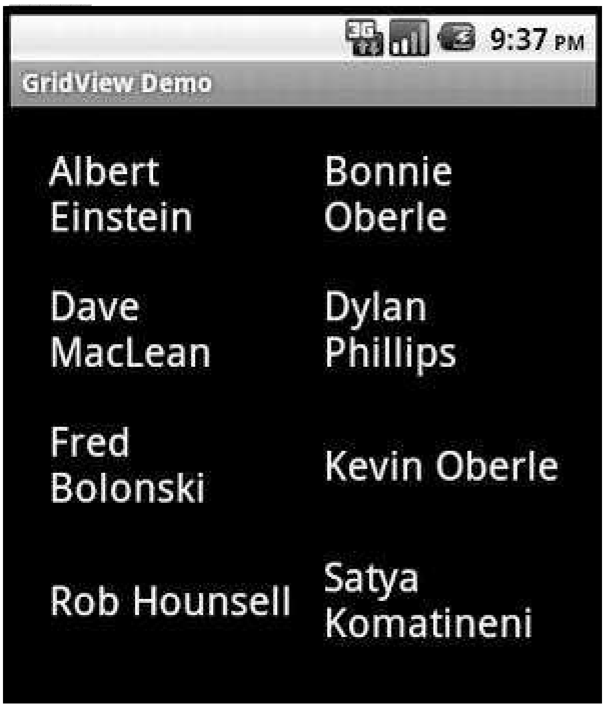
Debra Kelly □

Clickable Items in a ListView:-

We're able to scroll up and down the list to see all our contact names, but that's about it. What if we want to do something a little more interesting with this example, like launch the Contact application when a user clicks one of the items in our ListView?

Adding Other Controls with a ListView:-

If we want additional controls in our main layout, we can provide our own layout XML file, put in a ListView, and add other desired controls. For example, we could add a button below the ListView in the UI to submit an action on the selected items, as shown in the figure.



GridView Demo

Albert Einstein	Bonnie Oberle
Dave MacLean	Dylan Phillips
Fred Bolonski	Kevin Oberle
Rob Hounsell	Satya Komatineni

GRID VIEW

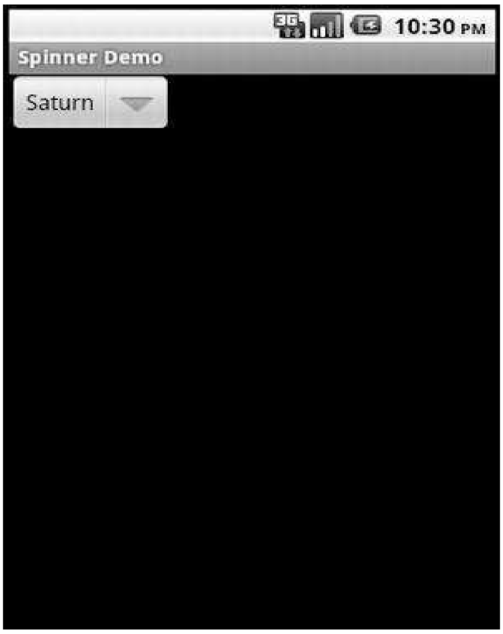
Android has a GridView control that can display data in the form of a grid. We use the term *data* here; the contents of the grid can be text, images, and so on. The GridView control displays information in a grid. The usage pattern for the GridView is to define the grid in the XML layout and then bind the data to the grid using an android.widget.ListAdapter. Don't forget to add the uses-permission tag to the AndroidManifest.xml file to make this example work.

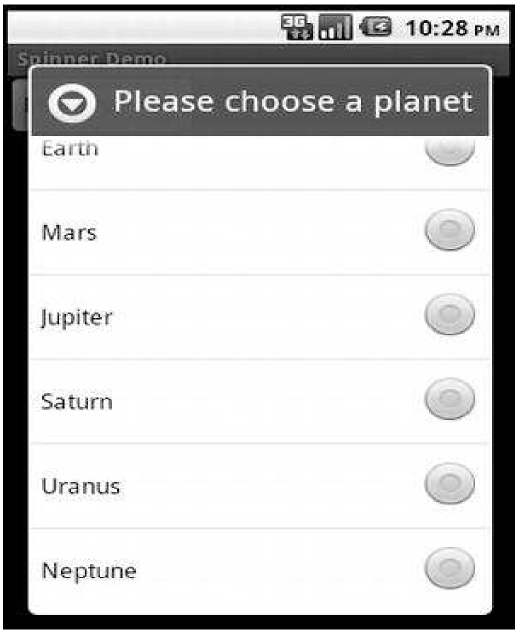
We've no doubt noticed that the adapter used by the grid is a ListAdapter. Lists are generally one-dimensional, whereas grids are two-dimensional. We can conclude, then, that the grid actually displays list-oriented data. And it turns out that the list is displayed by rows. That is, the list goes across the first row, then across the second row, and so on.

As before, we have a list control that works with an adapter to handle the data management, and the generation of the child views. The same techniques we used before should work just fine with GridViews. One exception relates to making selections: there is no way to specify multiple choices in a GridView.

SPINNER CONTROL

The Spinner control is like a drop-down menu. It is typically used to select from a relatively short list of choices. If the choice list is too long for the display, a scrollbar is automatically added for us. Although a spinner is technically a list control, it will appear to us more like a simple TextView control.





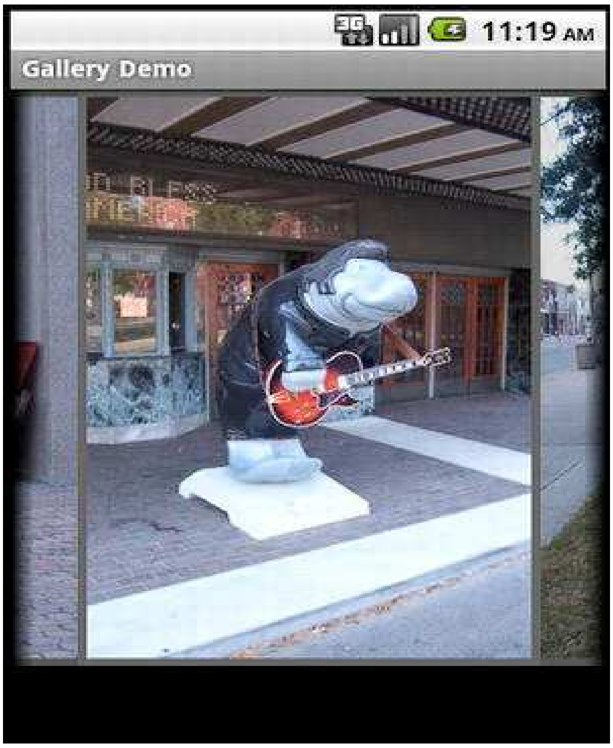
In other words, only one value will be displayed when the spinner is at rest. The purpose of the spinner is to allow the user to choose from a set of predetermined values: when the user clicks the small arrow, a list is displayed, and the user is expected to pick a new value. Populating this list is done in the same way as the other list controls: with an adapter. Because a spinner is often used like a drop-down menu, it is common to see the adapter get the list choices from a resource file. Notice the new attribute called android:prompt for setting a prompt at the top of the list to choose from. The actual text for our spinner prompt is in our /res/values/strings.xml file. As we should expect, the Spinner class has a method for setting the prompt in code as well.

STYLES AND THEMES

Android provides several ways to alter the style of views in our application. We'll first cover using markup tags in strings and then how to use spannables to change specific visual attributes of text. But what if we want to control how things look using a common specification for several views or across an entire activity or application?

GALLERY CONTROL

The Gallery control is a horizontally scrollable list control that always focuses at the center of the list. This control generally functions as a photo gallery in touch mode. The Gallery control is typically used to display images, so our adapter is likely going to be specialized for images. We'll show a custom image adapter in next section on custom adapters. See the figure.

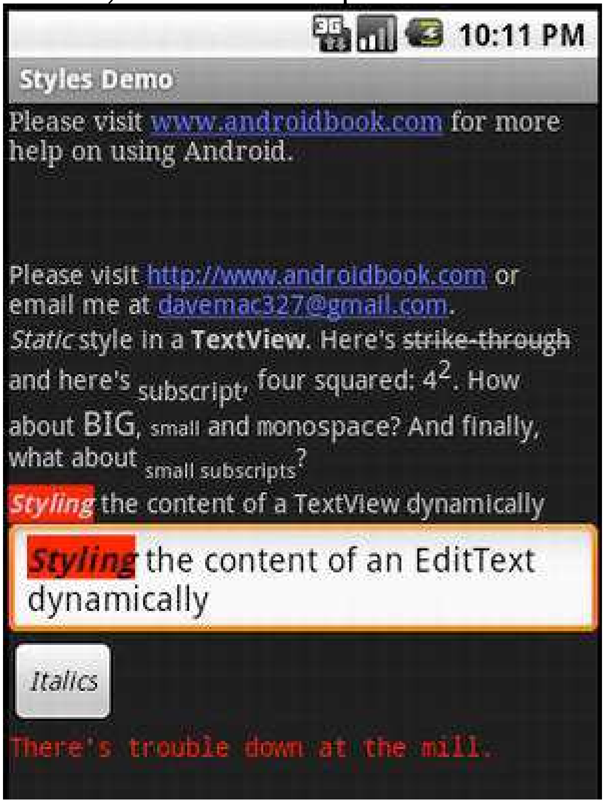


Using Styles:-

Sometimes, we want to highlight or style a portion of the View's content. We can do this statically or dynamically. Statically, we can apply markup directly to the strings in our string resources, as shown here:

<string name="styledText"><i>Static</i> style in a <b>TextView</b>.</string>

We can then reference it in our XML or from code. Note that we can use the following HTML tags with string resources: <i>, <b>, and <u> for italics, bold, and underlined, respectively, as well as <sup> (superscript), <sub> (subscript), <strike> (strikethrough), <big>, <small>, and <monospace>. We can even nest these to get, for example, small superscripts. This works not just in TextViews but also in other views, like buttons. The figure shows what styled and themed text looks like, using many of the examples in this section.



Please visit www.androidbook.com for more help on using Android.

Please visit http://www.androidbook.com or email me at davemac327@gmail.com.

Static style in a **TextView**. Here's ~~strike-through~~ and here's $_{\text{subscript}}$, four squared: 4^2. How about BIG, small and monospace? And finally, what about $_{\text{small subscripts}}$?

Styling the content of a TextView dynamically

Styling the content of an EditText dynamically

There's trouble down at the mill.

Styling a TextView control's content programmatically requires a little additional work but allows for much more flexibility (see Listing 6–35), because we can style it at runtime. This flexibility can only be applied to a spannable, though, which is how EditText normally manages the internal text, whereas TextView does not normally use Spannable. Spannable is basically a String that we can apply styles to. To get a TextView to store text as a spannable, we can call setText() this way:

tv.setText("This text is stored in a Spannable",

```
TextView.BufferType.SPANNABLE);
```

Then, when we call tv.getText(), we 'll get a spannable. We can get the content of the EditText (as a Spannable object) and then set styles for portions of the text. The code in the listing sets the text styling to bold and italics and sets the background to red. We can use all the styling options as we have with the HTML tags as described previously, and then some.

Using Themes:-

One problem with styles is that we need to add an attribute specification of style="@style/..." to every view definition that we want it to apply to. If we have some style elements we want applied across an entire activity, or across the whole application, we should use a theme instead. A *theme* is really just a style applied broadly; but in terms of defining a theme, it's exactly like a style. In fact, themes and styles are fairly interchangeable: we can extend a theme into a style or refer to a style as a theme. Typically, only the names give a hint as to whether a style is intended to be used as a style or a theme. To specify a theme for an activity or an application, we would add an attribute to the <activity> or <application> tag in the AndroidManifest.xml file for our project. The code might look like one of these:

```
<activity android:theme="@style/MyActivityTheme">
<application android:theme="@style/MyApplicationTheme">
<application android:theme="@android:style/Theme.NoTitleBar">
```

We can find the Android-provided themes in the same folder as the Android-provided styles, with the themes in a file called themes.xml. When we look inside the themes file, we will see a large set of styles defined, with names that start with Theme. We will also notice that within the Android-provided themes and styles, there is a lot of extending going on, which is why we end up with styles called Theme.Dialog.AppError", This concludes our discussion of the Android control set. As we mentioned in the beginning of the chapter, building UIs in Android requires we to master two things: the control set and the layout managers. In the next section, we are going to discuss the Android layout managers.

UNDERSTANDING LAYOUT MANAGERS

Android offers a collection of view classes that act as containers for views. These container classes are called *layouts* (or *layout managers*), and each implements a specific strategy to manage the size and position of its children. For example, the LinearLayout class lays out its children either horizontally or vertically, one after the other. All layout managers derive from the View class; therefore we can nest layout managers inside of one another. The layout managers that ship with the Android SDK are defined in Table 6–2.

Table 6–2. *Android Layout Managers*

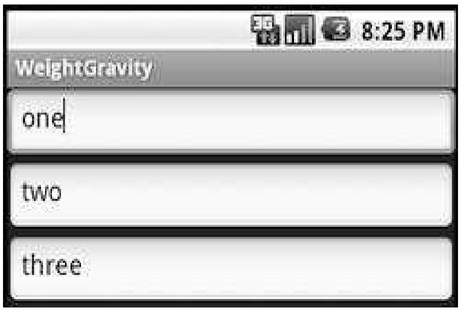
Layout Manager	Description
LinearLayout	Organizes its children either horizontally or vertically
TableLayout	Organizes its children in tabular form
RelativeLayout	Organizes its children relative to one another or to the parent
FrameLayout	Allows us to dynamically change the control(s) in the layout
GridLayout	Organizes its children in a grid arrangement

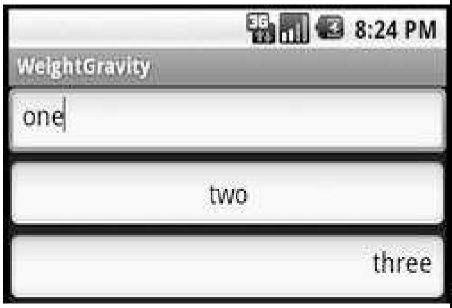
LINEAR LAYOUT MANAGER

The LinearLayout layout manager is the most basic. This layout manager organizes its children either horizontally or vertically based on the value of the orientation property. We've used LinearLayout in several of our examples so far. We can create a vertically oriented LinearLayout by setting the value of orientation to vertical. Because layout managers can be nested, we could, for example, construct a vertical layout manager that contained horizontal layout managers to create a fill-in form, where each row had a label next to an EditText control. Each row would be its own horizontal layout, but the rows as a collection would be organized vertically.

Layout managers extend android.widget.ViewGroup, as do many control-based container classes such as ListView. Although the layout managers and control-based containers extend the same class, the layout manager classes strictly deal with the sizing and position of controls and not user interaction with child controls. For example, compare LinearLayout to the ListView control. On the screen, they look similar in that both can organize children vertically. But the ListView control provides APIs for the user to make selections, whereas LinearLayout does not. In other words, the control-based container (ListView) supports user interaction with the items in the container, whereas the layout manager (LinearLayout) addresses sizing and positioning only.

Using the LinearLayout layout manager





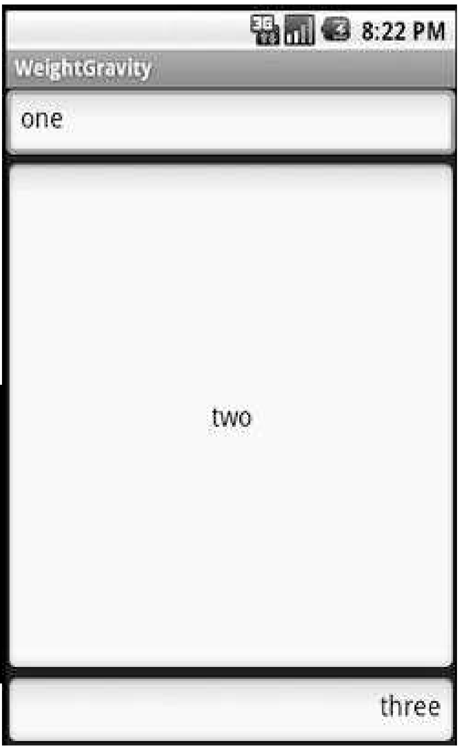
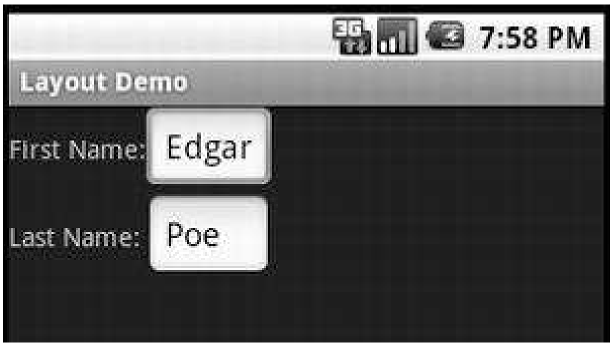


TABLE LAYOUT MANAGER

The TableLayout layout manager is an extension of LinearLayout. This layout manager structures its child controls into rows and columns. To use this layout manager, we create an instance of TableLayout and place TableRow elements within it. These TableRow elements contain the controls of the table. Because the contents of a TableLayout are defined by rows as opposed to columns, Android determines the number of columns in the table by finding the row with the most cells. Android creates a table with two rows and three columns. The last column of the first row is an empty cell.

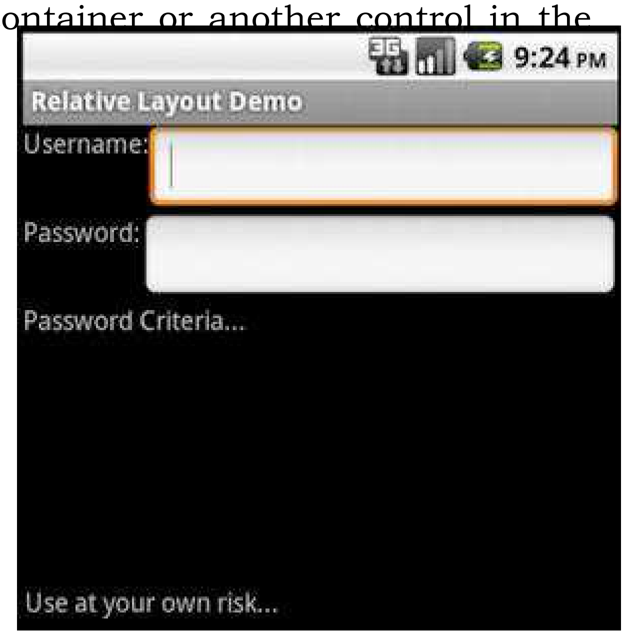


First Name: Edgar

Last Name: Poe

RELATIVE LAYOUT MANAGER

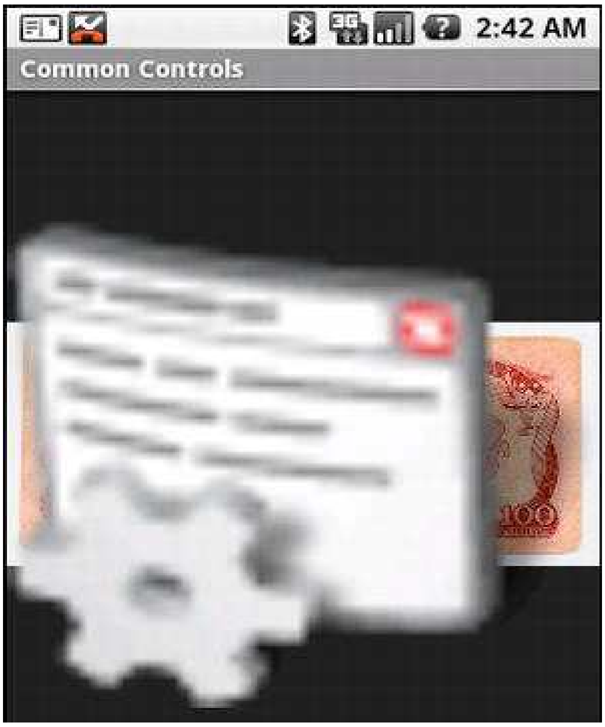
Another interesting layout manager is RelativeLayout. As the name suggests, this layout manager implements a policy where the controls in the container are laid out relative to either the container or another control in the container. As shown, the UI looks like a simple login form. The username label is pinned to the top of the container, because we set android:layout_alignParentTop to true. Similarly, the Username input field is positioned below the Username label because we set android:layout_below. The Password label appears below the Username label, and the Password input field appears below the Password label. The disclaimer label is pinned to the bottom of the container set android:layout_alignParentBottom to true.



Besides these three layout attributes, we can also specify layout_above, layout_toRightOf, layout_toLeftOf, layout_centerInParent, and several more. Working with RelativeLayout is fun due to its simplicity. In fact, once we start using it, it'll become our favorite layout manager we'll find our self going back to it over and over again.

FRAME LAYOUT MANAGER

The layout managers that we've discussed so far implement various layout strategies. In other words, each one has a specific way that it positions and orients its children on the screen. With these layout managers, we can have many controls on the screen at one time, each taking up a portion of the screen. Android also offers a layout manager that is mainly used to display a single item: FrameLayout. We mainly use this utility layout class to dynamically display a single view, but we can populate it with many items, setting one to visible while the others are invisible.



As we said earlier, we generally use FrameLayout when we need to dynamically set the content of a view to a single control. Although this is the general practice, the control will accept many children, as we demonstrated. Adds two controls to the layout but has one of the controls visible at a time. FrameLayout, however, does not force us to have only one control visible at a time. If we add many controls to the layout, FrameLayout will simply stack the controls, one on top of the other, with the last one on top. This can create an interesting UI. For example, Figure 6–25 shows a FrameLayout control with two ImageView objects that are visible. We can see that the controls are stacked, and that the top one is partially covering the image behind it.

Another interesting aspect of the FrameLayout is that if we add more than one control to the layout, the size of the layout is computed as the size of the largest item in the container, the top image is actually much smaller than the image behind it, but because the size of the layout is computed based on the largest control, the image on top is stretched. Also note that if we put many controls inside a FrameLayout with one or more of them invisible to start, we might want to consider using setMeasureAllChildren(true) on our FrameLayout. Because the largest child dictates the layout size, you'll have a problem if the largest child is invisible to begin with: when it becomes visible, it is only partially visible. To ensure that all items are rendered properly, call setMeasureAllChildren() and pass it a value of true. The equivalent XML attribute for FrameLayout is android:measureAllChildren="true".

GRID LAYOUT MANAGER

Android 4.0 brought with it a new layout manager called GridLayout. As we might expect, it lays out views in a grid pattern of rows and columns, somewhat like TableLayout. However, it's easier to use than TableLayout. With a GridLayout, we can specify a row and column value for a view, and that's where it goes in the grid. This means we don't need to specify a view for every cell, just those that we want to hold a view. Views can span multiple grid cells. We can even put more than one view into the same grid cell.

When laying out views, we must not use the weight attribute, because it does not work in child views of a GridLayout. We can use the layout_gravity attribute instead. Other interesting attributes we can use with

GridLayout child views include layout_column and layout_columnSpan to specify the left-most column and the number of columns the view takes up, respectively. Similarly, there are layout_row and layout_rowSpan attributes. Interestingly, we do not need to specify layout_height and layout_width for GridLayout child views; they default to WRAP_CONTENT.