Chapter 4

Controls and the User Interface

Check Boxes

In the previous chapter, we looked at two common and easy-to-implement user interfaces: the `Button` and the `EditText`. Both of these are standard in the Android software development kit (SDK) and can, for the most part, be configured in the `main.xml` file. This section examines a few more useful ways the user can interact with the application and how to handle the events they produce.

The controls, or *widgets* as they are sometimes called, are descendents of the Android widget class, just like all the layouts we looked at. A relatively easy widget to begin with here is the `CheckBox`. The `CheckBox` can be activated with a finger touch and can be polled in the application's code for a checked or unchecked state. To use this widget, let's create an application that allows construction of a shopping list so the user can check off items as they are picked up. For simplicity sake, let's make the list hold just five items. There will be no way to save or retrieve the items; they will only be on the application while it is running. Again, this is just an example of the `CheckBox` in action. We will use a `TableLayout` with five rows, each holding an `EditText` and a `CheckBox`. At the bottom, we will add a digital clock so the shopper can keep track of time, and a `TextView` that will display the word `DONE!` when all the `CheckBoxes` are checked.

Here is the `main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
```

```
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
<TableRow
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
     >
     <EditText
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:minWidth="250px"
     android:id="@+id/item1"
     android:paddingBottom="5px"
     />
     <CheckBox
     android:id="@+id/check1"
     />
    </TableRow>
    <TableRow
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
     >
     <EditText
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:minWidth="250px"
     android:id="@+id/item2"
     android:paddingBottom="5px"
     />
     <CheckBox
     android:id="@+id/check2"
     />
    </TableRow>
    <TableRow
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
     >
     <EditText
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:minWidth="250px"
     android:id="@+id/item3"
     android:paddingBottom="5px"
     />
```

```
 <CheckBox
 android:id="@+id/check3"
 />
</TableRow>
<TableRow
android:layout_width="fill_parent"
android:layout_height="wrap_content"
 >
 <EditText
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:minWidth="250px"
 android:id="@+id/item4"
 android:paddingBottom="5px"
 />
 <CheckBox
 android:id="@+id/check4"
 />
</TableRow>
<TableRow
android:layout_width="fill_parent"
android:layout_height="wrap_content"
 >
 <EditText
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:minWidth="250px"
 android:id="@+id/item5"
 android:paddingBottom="5px"
 />
 <CheckBox
 android:id="@+id/check5"
 />
</TableRow>
<TableRow
 android:layout_width="fill_parent"
android:layout_height="wrap_content"
>
<DigitalClock
android:textSize="20pt"
 android:layout_width="fill_parent"
android:layout_height="wrap_content"
/>
</TableRow>
```

```
  <TableRow
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
   >
   <TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:minWidth="250px"
   android:id="@+id/done"
   android:textSize="20pt"
   android:paddingBottom="5px"
   android:text=""
   android:background="#ffffff"
    android:textColor="#ff0000"
   />
   </TableRow>
</TableLayout>
```

We have added a couple of extra parameters to the XML file that we didn't mention previously, such as `android:minWidth`, but they should be fairly self-evident. As you might guess, the "px" on some of the values stands for pixels, and "pt" stands for points, each point being 1/72 of an inch. Note that the "px" unit is fixed; as the screen density of the device changes, the actual size in inches of an image changes. If you are developing for a broad range of devices, this may not be appropriate. An alternative is the "dp" unit. Dp stands for Density-independent Pixels, (sometimes referred to as "dip," and the compiler will accept both "dp" and "dip"), and it is based on a 160-pixel-per-inch screen. In practicality, the use of 160dp instead of 160px will ensure that the item displayed will be the same size no matter what device it is displayed on. The "sp" unit, which stands for Scale-independent Pixels, works the same way but is used for font size specifications.

Here is the code for the Java file:

```
package com.sheusi.ShoppingList;

import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
import android.view.View.OnClickListener;
public class ShopList extends Activity implements OnClickListener{
  /** Called when the activity is first created. */
  EditText et1=null;
  CheckBox cb1=null;
```

```
    EditText et2=null;
    CheckBox cb2=null;
    EditText et3=null;
    CheckBox cb3=null;
    EditText et4=null;
    CheckBox cb4=null;
    EditText et5=null;
    CheckBox cb5=null;
    TextView tv=null;
   @Override
    public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.main);
       et1=(EditText)findViewById(R.id.item1);
       cb1=(CheckBox)findViewById(R.id.check1);
       et2=(EditText)findViewById(R.id.item1);
       cb2=(CheckBox)findViewById(R.id.check2);
       et3=(EditText)findViewById(R.id.item1);
       cb3=(CheckBox)findViewById(R.id.check3);
       et4=(EditText)findViewById(R.id.item1);
       cb4=(CheckBox)findViewById(R.id.check4);
       et5=(EditText)findViewById(R.id.item1);
       cb5=(CheckBox)findViewById(R.id.check5);
       tv=(TextView)findViewById(R.id.done);
       cb1.setOnClickListener(this);
       cb2.setOnClickListener(this);
       cb3.setOnClickListener(this);
       cb4.setOnClickListener(this);
       cb5.setOnClickListener(this);
       cb1.setChecked(false);
       cb2.setChecked(false);
       cb3.setChecked(false);
       cb4.setChecked(false);
       cb5.setChecked(false);
     }
   public void onClick(View v){
     if(cb1.isChecked() & cb2.isChecked() & cb3.isChecked() & cb4.isChecked() &
 cb5.isChecked())

       tv.setText("DONE!");
     else
       tv.setText("");
   }
 }
```

Your running application should look like Figure 4.1.

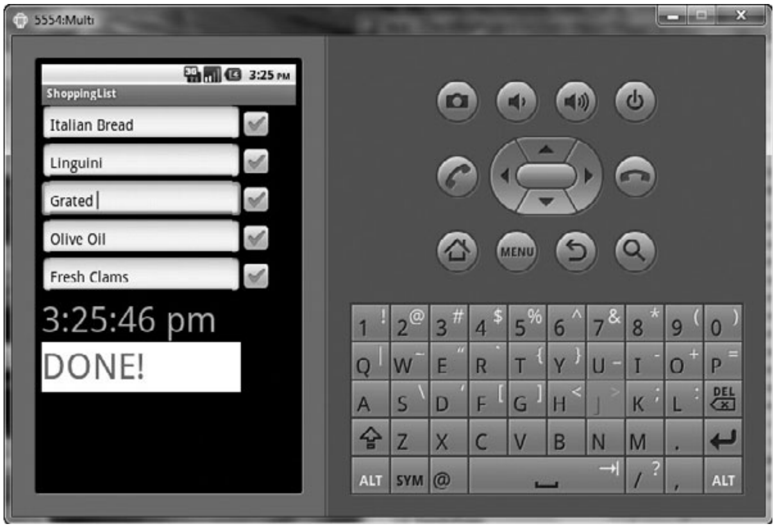

Figure 4.1
Emulator image showing shopping list application.

Radio Buttons

Java programmers are familiar with check boxes and how combining them into a `CheckBox` group turns them into radio buttons. What makes radio buttons special is that they are mutually exclusive; that is if one is selected, all the others are automatically deselected. They are called radio buttons because they work like the station selectors on a car radio. An easy demonstration for radio buttons is a tip calculator. We will supply an `EditText` box in which the user can supply the dinner bill amount and select one of three radio buttons to determine a tip for 10, 15, or 20 percent. The tip is calculated upon selection of one of the buttons. The following code is the `main.xml` file for the tip calculator:

```
<?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"
  >
<TableLayout
android:layout_width="fill_parent"
  android:layout_height="wrap_content"
>
```

```
<TableRow
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
    >
    <TextView
     android:layout_width="wrap_content"
    android:layout_height="wrap_content"
      android:textSize="9pt"
    android:text="Enter Bill Amount: $"

    />

    <EditText
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
      android:textSize="9pt"
    android:id="@+id/bill_amount"
   android:minWidth="100px"
    />

</TableRow>
</TableLayout>
 <RadioGroup
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:id="@+id/tip_choices"
   >

<RadioButton
   android:id="@+id/ten"
    android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="10%"
   android:textSize="20pt"
   />
   <RadioButton
   android:id="@+id/fifteen"
    android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="15%"
   android:textSize="20pt"
   />
   <RadioButton
   android:id="@+id/twenty"
    android:layout_width="wrap_content"
   android:layout_height="wrap_content"
```

```
    android:text="20%"
    android:textSize="20pt"
    />
    </RadioGroup>
<TableLayout
 android:layout_width="fill_parent"
    android:layout_height="wrap_content" >
<TableRow
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
     >
     <TextView
      android:layout_width="wrap_content"
     android:layout_height="wrap_content"
       android:textSize="9pt"
     android:text="The tip should be: "

     />

     <TextView
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
      android:textSize="9pt"
       android:minWidth="100px"
     android:id="@+id/tip_amount"
      android:background="#ffffff"
     android:textColor="#000000"
     />
    </TableRow>
</TableLayout>

</LinearLayout>
```

The `main.xml` files are getting longer, but they aren't getting more complicated. There are just more elements with essentially the same attributes that we have been using all along. The important addition here is the use of the `RadioButtons` and the `RadioGroup`. Notice how all the `RadioButtons` are set inside the `RadioGroup` tags. Also notice that we gave the `RadioGroup` an ID value. This is important because in the code the `RadioGroup` will be an object we have to modify. Study the Java source code for the tip calculator:

```
package com.sheusi.tips;

import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
import android.widget.RadioGroup.*;
import java.text.DecimalFormat;

public class Tips extends Activity implements RadioGroup.OnCheckedChangeListener{
   /** Called when the activity is first created. */
  EditText ba=null;
  TextView ta=null;
  RadioButton t10=null;
  RadioButton t15=null;
  RadioButton t20=null;
  RadioGroup rg=null;
  DecimalFormat df=new DecimalFormat("$####.00");
   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      ba=(EditText)findViewById(R.id.bill_amount);
      ta=(TextView)findViewById(R.id.tip_amount);
      t10=(RadioButton)findViewById(R.id.ten);
      t15=(RadioButton)findViewById(R.id.fifteen);
      rg=(RadioGroup)findViewById(R.id.tip_choices);
      rg.setOnCheckedChangeListener(this);
   }
   public void onCheckedChanged(RadioGroup rg,int i){
     if(i==t10.getId())
       ta.setText(df.format(Double.parseDouble(ba.getText().toString())*.10));
     if(i==t15.getId())
       ta.setText(df.format(Double.parseDouble(ba.getText().toString())*.15));
     if(i==t20.getId())
       ta.setText(df.format(Double.parseDouble(ba.getText().toString())*.20));
     }
}
```

It's interesting that, by simple line count, the `main.xml` exceeds the Java code. This goes to show how much work the `main.xml` file actually does. The first thing you might notice about the Java code is that we introduced a new listener, the `OnCheckedChange-Listener`. What is worth noting about this listener is that it is connected to the `RadioGroup` and not the individual `RadioButtons`. The `onCheckedChanged( )` method of the listener takes two arguments: a `RadioGroup` and an integer value. We could share the method among several `RadioGroups`, so we need to know which one we are dealing with when an event occurs. The integer parameter identifies the `RadioButton` that was selected.

What are the integer values? Well, we really don't need to know them while we are coding because we can use the `.getId( )` method of each button to retrieve the actual value, but if you are really curious look at the `R.java` file. You will notice there is a hexadecimal value associated with each object listed. (It starts with "0x" to indicate it is a hexadecimal value.) If you take any of these numbers and convert it to decimal, you will have the integer value returned by the corresponding object's `.getId( )` method. Technically, the computer doesn't care if it's hex, decimal, or binary, but if you write code to print the value to a `TextView` for curiosity sake, the value will print as decimal by default. Once again, we see how the project files are tied to each other.

The `if( )` statements take the bill amount that exists as text, convert it to a double data type, multiply by the appropriate factor, format the product according to the specification in the `DecimalFormat` object, and place it in the `TextView` at the bottom of the screen. The running application should look like Figure 4.2.

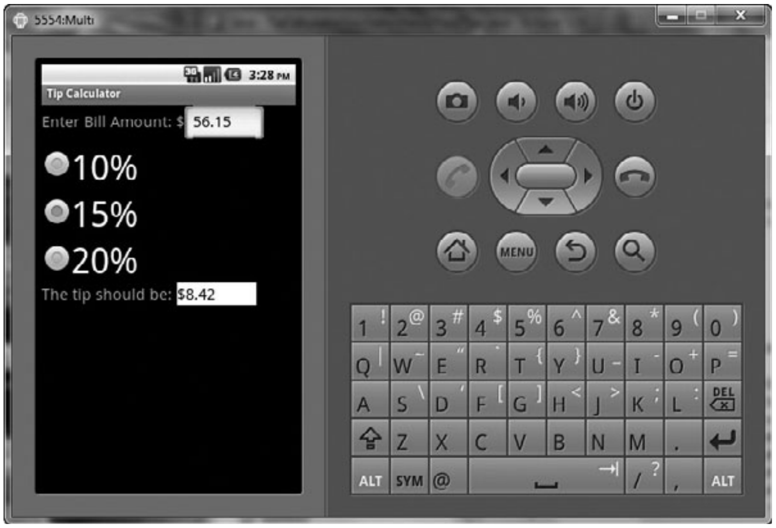

Figure 4.2
Emulator image showing Tip Calculator application.

The Spinner

You will notice in the `RadioButton` that the series of only three buttons takes up almost half the screen space. The programmer may need to contend with many items that must be displayed on the screen and many choices for a given item. A `Spinner` widget is ideal in this situation. The `Spinner` is similar to the `java.awt.Choice` class or the Visual Basic .NET ComboBox. The `Spinner` not only conserves screen space, it restricts the user to a list of valid, correctly spelled entries.

Because the entries are likely to be known at design time (for example, a list of states, a list of zip codes, and so on), they can be included in the XML files before Java code is written. The example we will look at displays the New England state names, allows the user to pick a state, and then displays the chosen entry in a TextView at the bottom of the list for confirmation's sake. Notice how little screen space the example actually occupies. The first item is the strings.xml file. This demonstrates how to enter an array of strings. We assume you understand the principle of an array in any programming language at this point. The string.xml file is found in the values directory under the res directory and is created by Eclipse when the project is created.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <string name="hello">Hello World, MySpinner!</string>
  <string name="app_name">Spinner Example</string>
  <string-array name="NewEnglandStates">
    <item>Maine</item>
    <item>New Hampshire</item>
    <item>Vermont</item>
    <item>Massachusetts</item>
    <item>Rhode Island</item>
    <item>Connecticut</item>
    </string-array>
   <string name="spinner_prompt">Pick A State</string>
</resources>
```

Next, let's look at the main.xml file where we make an entry for the Spinner widget.

```
<?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="A Spinner Example"
  />
  <Spinner
  android:id="@+id/NEStates"
   android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:entries="@array/NewEnglandStates"
  android:prompt="@string/spinner_prompt"
```

```
   ></Spinner>
   <TextView
   android:id="@+id/tv1"
   android:layout_width="fill_parent"
   android:layout_height="wrap_content"
   />
</LinearLayout>
```

In the Spinner element, the android:prompt places a string at the top of the list when it is opened. It is optional, and without it the first choice in the list simply appears at the top of the list. The TextView entry at the bottom of the LinearLayout demonstrates that the Spinner works and is unnecessary to a functional application. However, the programmer should be aware that the first entry in the Spinner's list (the array defined in the strings.xml file) will be the default entry when the application is started, and any variable that uses the results of the user's choice for its value will have the first choice as its value when the application starts.

Finally, let's look at the Java code for the application.

```
package com.sheusi.SpinnerExample;
import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
public class MySpinner extends Activity implements AdapterView.OnItemSelectedListener{
   /** Called when the activity is first created. */
   Spinner statespinner=null;
   TextView tv1=null;
  @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      statespinner=(Spinner)findViewById(R.id.NEStates);
      tv1=(TextView)findViewById(R.id.tv1);
      statespinner.setOnItemSelectedListener(this);
  }
    public void onItemSelected(AdapterView<?> parent, View view, int pos, long id) {

      tv1.setText(((TextView)view).getText());
      }
    public void onNothingSelected(AdapterView<?> parent) {
     // do nothing
  }
}
```

Notice the introduction of a new `Listener` interface required to support the `Spinner`: the `AdapterView.OnItemSelectedListener`. Use of this interface requires definition of two methods: `onItemSelected( )` and `onNothingSelected( )`. As a Java programmer, you should know that Java allows multiple adapters to be used as long as the required methods for each adapter are defined in the class implementing the adapters. You can also create inline classes to implement the adapters, which is common in examples you might find on the Internet. Other than that, the code should look familiar and be pretty straightforward by now.

The programmer may have occasion to create the array of choices dynamically in code, such as assigning a list of cities to a `Spinner` object based on the state chosen by another `Spinner` object, or entered in an `EditText`; or even the results of a database query. The modification of the original Java code to achieve this is shown here. The `Spinner` here is loaded from an array of states entered by assignment statements for convenience and clarity.

```
package com.sheusi.SpinnerExample;
import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
public class MySpinner extends Activity implements AdapterView.OnItemSelectedListener{
   /** Called when the activity is first created. */
   Spinner statespinner=null;
   TextView tv1=null;
  @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      statespinner=(Spinner)findViewById(R.id.NEStates);

      // Begin dynamically loaded Spinner object
      String[] southernstates=new String[5];
      southernstates[0]="Florida";
      southernstates[1]="Louisiana";
      southernstates[2]="Texas";
      southernstates[3]="California";
      southernstates[4]="Arizona";
      ArrayAdapter<String> adapter = new ArrayAdapter<String>( this,
         android.R.layout.simple_spinner_item,southernstates);
     adapter.setDropDownViewResource(
      android.R.layout.simple_spinner_dropdown_item);
     statespinner.setAdapter(adapter);
```

```
        // End dynamically loaded Spinner object

        tv1=(TextView)findViewById(R.id.tv1);
        statespinner.setOnItemSelectedListener(this);

         }
      public void onItemSelected(AdapterView<?> parent,
  View view, int pos, long id) {
        tv1.setText(((TextView)view).getText());
       }
       public void onNothingSelected(AdapterView<?> parent) {

       }
}
```

The lines that follow create an instance of an `ArrayAdapter` for strings and load it with the contents of the Southern States array as indicated by the third parameter. The second parameter, `android.R.layout.simple_spinner_item`, is part of the Android SDK, so leave it as it is.

```
 ArrayAdapter<String> adapter = new ArrayAdapter<String>( this,
        android.R.layout.simple_spinner_item,southernstates);
```

The following runs the `setDropDownViewResource( )` method for our adapter instance. As above, the parameter `android.R.layout.simple_spinner_drop down_item`, one of several constants of the `R.layout` class, is part of the SDK, so leave that as is.

```
 adapter.setDropDownViewResource(
      android.R.layout.simple_spinner_dropdown_item);
```

Finally, the next line simply connects our newly created Adapter instance to the `Spinner`:

```
statespinner.setAdapter(adapter);
```

Recall that the `android:entries` parameter in the `main.xml` file assigned the list of values to the `Spinner`. The Java code replaces these with the new array. If you intend to assign the values in your Java code, you could eliminate this parameter in the `main.xml` file.

Your running application should look like Figure 4.3.

DatePicker

Two more attractive and useful widgets in the Android SDK are the `DatePicker` and the `TimePicker`. They, like the `Spinner`, are ideal because the programmer doesn't have to worry about the application user choosing invalid date values, such

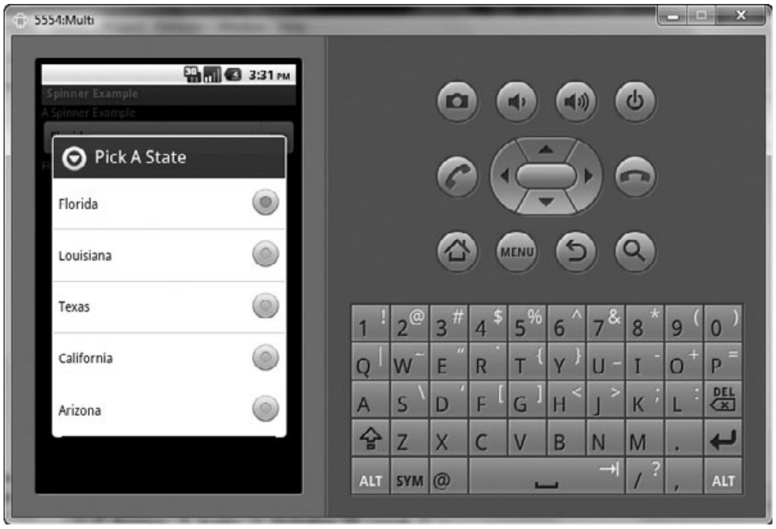

Figure 4.3
Emulator image showing an open Spinner object.

as the 33rd of the month, spelling the month incorrectly due to the tiny keyboard on a mobile device, or using an improper date format. The `DatePicker` and the `TimePicker` have their own listeners that we can implement by putting the appropriate methods in our main class, just as we did with the `Spinner`. What is different about these classes is that, to be useful, we need to look at two other Java classes: the `Calendar` class and the `Date` class. One or both of these classes typically store the date chosen by the user. Although the results of the widgets' fields are technically treated as integers, as we will see in the methods, the whole point of these widgets is to store and perform calculations with dates and times, not primitive integer values. Therefore, we must look at these Java classes. A little research on these classes will reveal what they can and cannot do and how they are typically used. There is some interchangeability between them, and each can be instantiated as an object using the other to supply its initial values.

Most documentation you might find on the `DatePicker` (which we concentrate on in this section) will show how to construct a `Date` object using the values from the `DatePicker`. However, if you look at documentation of Java's `Date` classes, you will see that the constructor often used by examples is actually deprecated, or obsolete. What's more, there are actually two `Date` classes: the `java.util.Date` and the `java.sql.Date`. This requires the programmer to take extra care when using a `Date` class. Some Java documentation recommends replacing the use of the `Date` class(es) with an instance of the `Calendar` class, which is what we will do in our example.

Our application will be a simple implementation of the `DatePicker` and a `TextView` field below the `DatePicker` to display the chosen date each time we change the `Date Picker`. You will notice in this application, just as we pointed out in the `Spinner` application, how much room on the screen is actually consumed by the `DatePicker`. A little later we will learn how to solve this problem by placing the `DatePicker` on a pop-up so it will appear over the rest of the screen until we dismiss it.

First, look at our `main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="fill_parent"
  android:layout_height="fill_parent"
  >
  <DatePicker
    android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:id="@+id/mydatepicker"/>
  <TextView
  android:layout_width="fill_parent"
  android:layout_height="wrap_content"
  android:textSize="12pt"
  android:background="#ffffff"
  android:textColor="#000000"
  android:id="@+id/myChosenDate"
  />
</LinearLayout>
```

There is nothing too special here, except we need to be sure to put `android:id` attributes in our elements. Look at the Java code here.

```
package com.sheusi.DatePickerDemo;

import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import java.util.*;

public class DatePickerDemo extends Activity implements DatePicker.OnDateChangedListener{
  /** Called when the activity is first created. */
  DatePicker dp=null;
  Calendar cal=null;
  TextView mcdate=null;
 @Override
```

```
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      dp=(DatePicker)findViewById(R.id.mydatepicker);
      mcdate=(TextView)findViewById(R.id.myChosenDate);
      cal=Calendar.getInstance();
      dp.init(dp.getYear(),dp.getMonth(),dp.getDayOfMonth(),this);
  }
  public void onDateChanged(DatePicker dpview,int year, int monthOfYear,
 int dayOfMonth){
    cal.set(year,monthOfYear, dayOfMonth);
    java.util.Date d=cal.getTime();
    mcdate.setText(String.valueOf(d.getMonth()+1)+"/"+String.valueOf(d.getDate())+"/
"+String.valueOf(d.getYear()+1900));

  }
}
```

As you have seen before, we implement a listener class and add the necessary method(s) to our code. Here we implement the `DatePicker.OnDateChangedListener` and add the necessary method, `onDateChanged( )`. You can see that we declared a `Calendar` variable at the top of the code.

In the `onCreate( )` method, we assign the `Calendar` variable with the factory method, `Calendar.getInstance( )`. This places the system's date's values into the instance variable. Likewise, when we assign the `DatePicker` object variable, "dp," it will be initialized with the system date. The `.init( )` method takes four parameters. The first three should be self-evident, and the fourth is set to the listener class instance to be associated with this particular `DatePicker`. In our case we use the word "this" because the listener is implemented in the surrounding class.

In the `onDateChanged( )` method, we first assign new date, month, and year values to our already-existing `Calendar` class variable. As we mentioned before, there are some things a `Date` object does better than a `Calendar` object, and vice versa. One of the things the `Date` class does better is allow access to its fields for printing. Therefore, we declare a local `Date` object and assign it with the values from the `Calendar` object. This is done in this line:

```
java.util.Date d=cal.getTime();
```

Finally, we fill our text field with values from the `Date` object formatted to look like the common form mm/dd/yyyy. There are many ways to do this; this is just one way.

You will see when you run the application that the date in the TextView field changes every time we change the DatePicker. As stated before concerning the Spinner, it is not necessary to display the results each time you change the date, but the programmer should retain the DatePicker's values to be used where necessary.

You may notice some arithmetic statements where the date display is assigned to the TextView object. First, we add one to the month because the months are stored as zero-based values in the DatePicker and subsequently in the Date object; in other words, January is month zero, not one. Again, when we display the year, we must add 1900 for essentially the same reason.

The TimePicker implementation is similar to the DatePicker's, so we will not give an example here. The programmer should just note that the listener is the TimePicker.OnTimeChangedListener, and the method that must be implemented is the onTimeChanged( ) method.

Your running application should look like Figure 4.4.

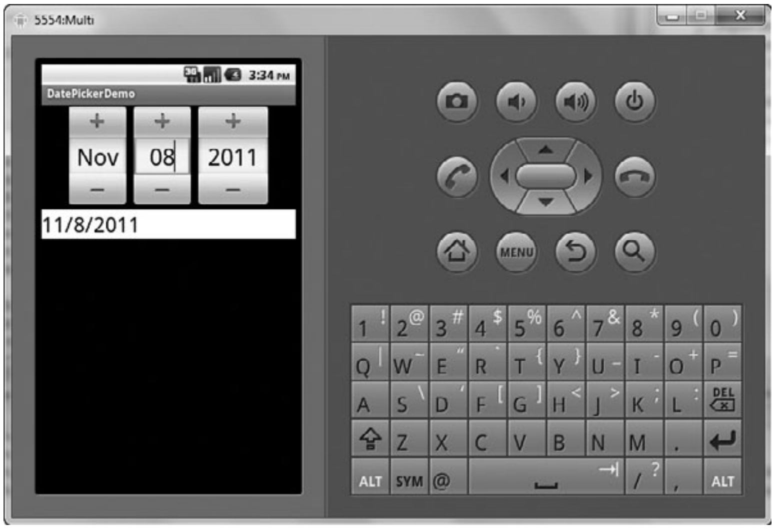

Figure 4.4
Emulator image showing a DatePicker object.

Follow-Up

1. Experiment with the following listener classes and methods that can be associated with the View class and its subclasses.

Listener Class	**Method**	**Description**
`View.OnClickListener`	`onClick( )`	This method is called when the user touches the item or focuses on the item with navigation keys or a trackball and then presses the Enter key or presses the trackball.
`View.OnLongClickListener`	`onLongClick( )`	Called when the user touches and stays on an item or holds down the Enter key or the trackball.
`View.OnFocusChange Listener`	`onFocusChange( )`	Called when the user navigates to or away from the item using navigation keys or the trackball.
`View.OnKeyListener`	`onKey( )`	Called when the user is focused on the item and presses or releases a key on the device. Typically, the developer wants to detect a specific key.
`View.OnTouchListener`	`onTouch( )`	Called when the user performs an action qualified as a touch event, such as a press, release, or movement within the bounds of the item associated with the listener.

2. Experiment with the `TimePicker` class by substituting it for the `DatePicker` in the last exercise.
3. Experiment with `onKey( )` listeners by responding to specific characters entered in `EditText` objects.

Key Classes Used in This Chapter

CheckBox

Class	`CheckBox`
Package	`android.widget`
Extends	`android.widget.CompoundButton`
Overview	The `CheckBox` is similar to a `Button` but has only two states: checked and unchecked.

Methods used in this chapter:

`void setChecked(boolean chked)`	Inherited from `CompoundButton`. This method changes the state of the `CheckBox`.
`boolean isChecked()`	Inherited from `CompoundButton`. This method returns the state of the `CheckBox`.
`void setOnClickListener(View.OnClickListener listener)`	Inherited from `View`. Registers a listener and process to be invoked when the control is clicked.

Other commonly used methods:

`void setOnCheckChangedListener(CompoundButton.OnCheckChangedListener listener)`	Registers a listener and process to be invoked when the control is changed. Inherited from `CompoundButton`.
`boolean performClick()`	Calls the view's `onClickListener`, if it is defined. Inherited from `CompoundButton`.
`void toggle()`	Changes the state of the check box to its inverse. Inherited from `CompoundButton`.

RadioButton

Class	`RadioButton`
Package	`android.widget`
Extends	`android.widget.CompoundButton`
Overview	The radio button is a two-state widget similar to the `CheckBox`; however, once it is checked, the user cannot uncheck it.

Methods used in this chapter:

`void setOnCheckChangedListener (CompoundButton.OnCheck-ChangedListener occl)`	Inherited from `CompoundButton`

Other commonly used methods:

`toggle( )`	Forces the state to be changed from the current state to the opposite state.
`boolean isChecked( )`	This method returns the state of the `CheckBox`. Inherited from `CompoundButton`.
`boolean performClick()`	Calls the view's `onClickListener`, if it is defined. Inherited from `CompoundButton`.

RadioGroup

Class	`RadioGroup`
Package	`android.widget`
Extends	`android.widget.LinearLayout`
Overview	This class creates a multiple exclusion group of radio buttons. In other words, if one radio button is checked, all others will become unchecked. Only one radio button in a group can be checked at a time. Initially, all buttons are unchecked. `RadioButtons` are added to a `RadioGroup` in a layout XML file similar to the way `TableRows` are added to a `TableLayout`.

Methods used in this chapter:

`setOnCheckChangedListener (CompoundButton.OnCheck-ChangedListener occl)`	Inherited from `CompoundButton`. Assigns a listener to the `RadioGroup`.

Other commonly used methods:

`void clearCheck()`	Clears all the `RadioButtons` in the group
`void check( int ID )`	Checks the `RadioButton` indicated by the ID
`void addView(View child, int index, ViewGroup.LayoutParams params)`	Adds a child view with the specified layout parameters
`int getCheckedRadioButtonId()`	Returns the identifier of the currently selected radio button in this group
`void clearCheck()`	Clears the current selection

Spinner

Class	`Spinner`
Package	`android.widget`
Extends	`android.widget.AbsSpinner`
Overview	A view that displays one child at a time and lets the user pick among them

Methods used in this chapter:

`setOnItemSelectedListener (AdapterView.OnItemSelected Listener listener)`	Assigns a listener to the `Spinner`. Inherited from `AdapterView`.
`setAdapter(SpinnerAdapter adapter)`	Associates an adapter to the `Spinner`. The adapter is the source of the items in the `Spinner`.

Other commonly used methods:

`setPrompt(CharSequence prompt)`	Sets the prompt to display when the `Spinner` is shown.
`int getCount()`	Returns the number of items in the `Spinner`. Inherited from `AbsSpinner`.
`CharSequence getPrompt()`	Retrieves the prompt from the `Spinner`.
`void setPromptId(int id)`	Sets the prompt to display when the dialog is shown.
`void setGravity()`	Describes how the selected item view is positioned.
`Object getSelectedItem()`	Returns selected item. Inherited from `AdapterView`.
`long getSelectedItemId()`	Returns selected item `id`. Inherited from `AdapterView`.

ArrayAdapter

Class	`ArrayAdapter`
Package	`android.widget`
Extends	`android.widget.BaseAdapter`
Overview	A `BaseAdapter` that is backed by an array of arbitrary objects. If the `ArrayAdapter` is used for anything other than a `TextView`, care must be taken to choose the correct constructor.

Methods used in this chapter:

`void setDropDownViewResource (int resource)`	Sets the layout resource to create the drop-down views

Other commonly used methods:

`void add(Object object)`	Adds an object at the end of the array
`void remove(Object object)`	Removes the specified object from the array
`void insert(Object object, int index)`	Inserts the specified object at the specified index position
`Object getItem(int position)`	Returns the object at the specified position
`int getPosition(Object item)`	Returns the position of the specified item in the array

DatePicker

Class	`DatePicker`
Package	`android.widget`
Extends	`android.widget.FrameLayout`
Overview	This class is a widget for selecting a date. The date is set by a series of `Spinners`. The date can be selected by a year, month, and day `Spinners` or a `CalendarView` object.

Methods used in this chapter:

`init(int year, int monthOfYear, int dayOfMonth, DatePicker.OnDateChangedListener onDateChangedListener)`	This method sets the initial date for the `DatePicker` object and assigns `OnDateChangedListener`.

Other commonly used methods:

`CalendarView getCalendarView( )`	Returns a `CalendarView`
`int getDayOfMonth( )`	Returns the day of the month set on this object
`int getMonth( )`	Returns the month set on this object
`int getYear( )`	Returns the year set on this object
`boolean getSpinnersShown()`	Gets whether the `Spinners` are shown
`boolean isEnabled()`	Gets whether the `DatePicker` is enabled
`void setEnabled(boolean enabled)`	Sets the enabled state of this view
`void updateDate(int year, int month, int dayOfMonth)`	Updates the current date

DatePicker.OnDateChangedListener

Class	`DatePicker.OnDateChangedListener` (interface)
Package	`android.widget`
Extends	n/a
Overview	Signals that the user has changed the date on the `DatePicker` associated with this listener

Methods used in this chapter: (required)

```
public abstract void onDateChanged(DatePicker view, int year, int
monthOfYear, int dayOfMonth)
```

Other commonly used methods:

n/a

Reader's Notes