

CHECKBOXES, RADIOBUTTONS AND SPINNER HAVE BEEN DISCUSSED IN PRACTICAL SESSIONS TOO. GO THROUGH THE FOLLOWING EXAMPLES. ARRAY ADAPTERS HAVE ALSO BEEN COVERED HERE WITH EXPLANATION OF EACH.

CONTROLS AND THE USER INTERFACE

CHECK BOXES

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:



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;

```

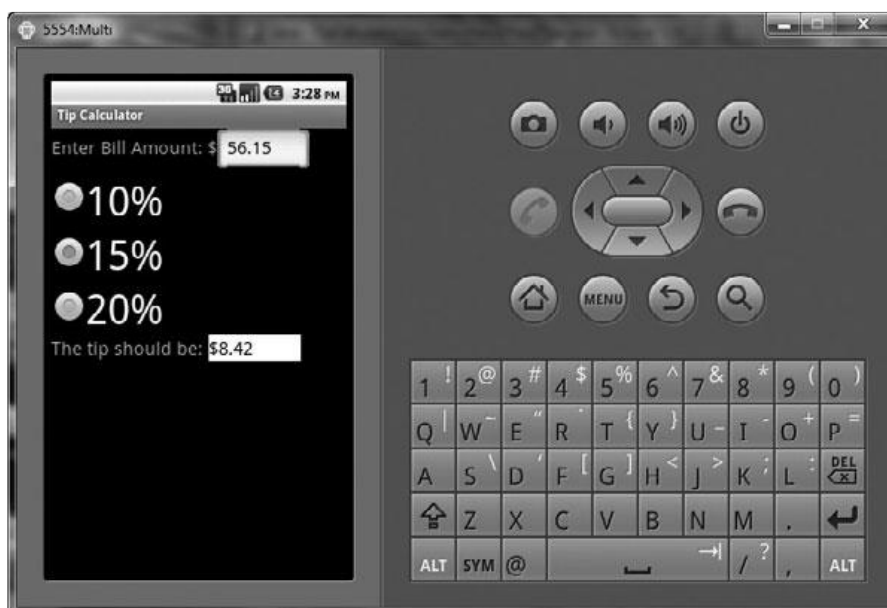
```

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 OnCheckedChangeListener. 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 :



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_dropdown_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 :



Summary of functions used above

Listener Class	Method	Description
<code>View.OnClickListener</code>	<code>onClick()</code>	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.
<code>View.OnLongClickListener</code>	<code>onLongClick()</code>	Called when the user touches and stays on an item or holds down the Enter key or the trackball.
<code>View.OnFocusChangeListener</code>	<code>onFocusChange()</code>	Called when the user navigates to or away from the item using navigation keys or the trackball.
<code>View.OnKeyListener</code>	<code>onKey()</code>	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.
<code>View.OnTouchListener</code>	<code>onTouch()</code>	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.

For any doubt contact 9873961590.