

## **MULTISCREEN APPLICATIONS**

### **STRETCHING THE SCREEN**

- The first and easiest is simply to design beyond the normal boundaries of the device. This is done with two classes we haven't looked at yet: The ScrollView and the HorizontalScrollView.
- These two classes are typically used in an XML file for a screen layout, such as the main.xml file, and become the outermost elements. They work by allowing the user to use his finger to roll the screen up and down or left to right to gain access to controls beyond the physical screen.
- In the examples that follow we will simply load the screen with TextViews
- that far outnumber what would fit on a normal screen.
- The application serves no useful purpose; it just demonstrates the principle discussed here.
- First, let's look at the main.xml file:

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

        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:id="@+id/L1"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello"
    />
</LinearLayout>
</ScrollView>
```

- The most important thing to note is that the ScrollView encapsulates the outermost layout—in this case, the LinearLayout.
- In fact, you are restricted to placing only one object inside the ScrollView or HorizontalScrollView; not adhering to this rule will cause your application to crash with a run-time exception.
- That being said, you can still pack your single inner object with objects of its own as you see here. It's just that between the ScrollView's pair of tags, there can be only one pair of tags at the next level in. A minor but necessary point is the following line:

```
xmlns:android="http://schemas.android.com/apk/res/android"
```

It must be associated with the outermost tag pair in the XML file.

- Notice that I moved it from the LinearLayout tag to the ScrollView tag. This is an XML rule, and it has nothing to do with the Android objects used here.
- The ScrollView extends the screen vertically, so make sure the orientation for your LinearLayout is vertical. Otherwise, you are defeating the purpose of the scroll effect.
- Now let's look at the Java source code:

```
package com.sheusi.ScrollViews;
import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.content.Context;
import android.graphics.Color;
public class ScrollViewsActivity extends Activity {
    /** Called when the activity is first created. */
    TextView[] tvarray=new TextView[30];
    LinearLayout myLayout=null;
    Context myContext=null;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        myContext=this.getApplicationContext();
        myLayout=(LinearLayout)findViewById(R.id.L1);
        for(int ct=0;ct<=29;++ct){
            int colormult;
            tvarray[ct]=new TextView(myContext);
            tvarray[ct].setText(" TextView number "+String.valueOf(ct));
```

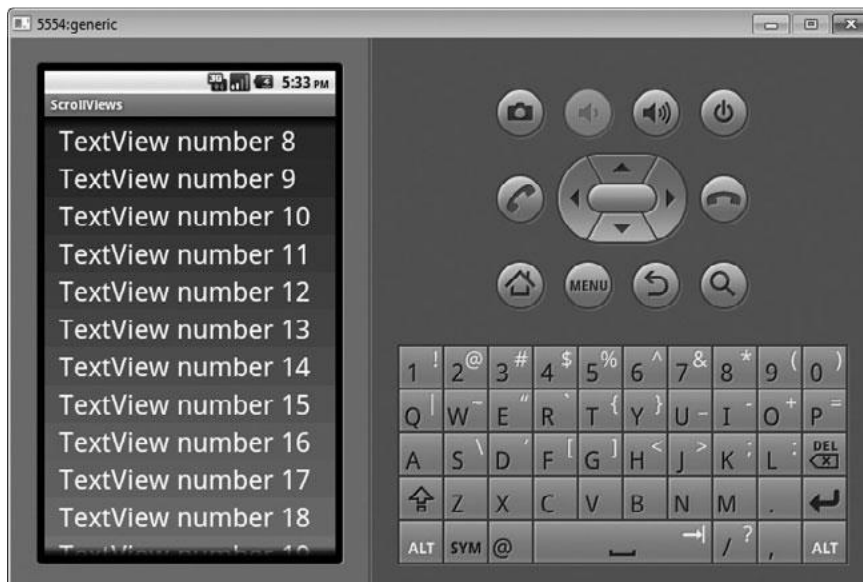
```

colormult=ct*8;
tvarray[ct].setBackgroundColor(Color.rgb(colormult,colormult,
colormult));
tvarray[ct].setTextSize(30);
myLayout.addView(tvarray[ct]);
}
}
}

```

- Again, realize that this is a junk application, only to demonstrate scrolling screens. This code will create 30 TextViews of various shades of gray on the screen, and the user will have to scroll the screen to see all of them.
- To make the code more efficient, I declare an array of TextViews and then, using a loop in the main code, I instantiate each of them, give them an individual shade of gray, and add text to number the control in the array. The coloring is just for effect. Remember that a 30-element array has index values 0 through 29; if you try to use index value 30, you will get a run-time exception, and the application will crash.
- **Note** Notice that the TextViews are added to the LinearLayout, not the ScrollView.
- The rule must be followed in XML and in Java code.

Figure 7.1 shows the application scrolled halfway through the TextViews.



**Figure 7.1**

Emulator showing an application using a ScrollView object.

- When the user actually scrolls the screen (you can do it on the emulator with a mouse drag), you will see a scrollbar on the right side. The scrollbar disappears as you release the screen, or the mouse in the case of the emulator
- As you might assume, the HorizontalScrollView is used and behaves in the same way, only left to right on the screen. You can modify the previous XML and Java code to see how it works.
- First, in your main.xml file, change the ScrollView tags to HorizontalScrollView. Be sure to change both the opening and closing tags. Next, change the orientation for the LinearLayout element to horizontal.
- In the Java code, we will change the size of the TextView from 30 to 8 and make changes to the loop according to the following code snippet:

```

for(int ct=0;ct<=7;++ct){
int colormult;
tvarray[ct]=new TextView(myContext);
tvarray[ct].setText(" TextView number "+String.valueOf(ct));
colormult=ct*30;
tvarray[ct].setBackgroundColor(Color.rgb(colormult, colormult, colormult));
tvarray[ct].setTextSize(30);
myLayout.addView(tvarray[ct]);
}

```

- We will reduce the loop size to reflect the size of the array and increase the grayscale multiplier to 30.
- **Note** Color values cannot exceed 255 for any of the red, green, and blue settings, so the variable colormult must stay below 255.
- The running application should now look like Figure 7.2.



- The other ways to expand your usable screen space are a little more difficult and need a lot more planning.
- For instance, instead of making one screen hold more information, we can use multiple screens if it is more appropriate.
- The next we will look at is using separate instances of the Activity class to produce a multi-screen application.
- Each screen is treated as a separate activity, so it is written into its own Java class file and has its own XML file for the screen layout.
- The manifest file also has an entry for each activity class used in the application.
- Activities start other activities by using an instance of Android's Intent class. An Intent is created by taking the Activity class as an argument to its constructor.
- Data can be passed back and forth among activities using a method built into the Intent class called `putExtra( )`.
- Standard services that the device provides, such as dialing the phone, are also built into the Intent class as integer values for the Intent's action field represented by symbolic constants.
- A symbolic constant is an English word or phrase that represents a numeric value, used to make source code more readable. A programmer can declare her own symbolic constants in applications, but the ones used here build into the Intent class.
- The following are some of the actions built into the Intent class:

`ACTION_DIAL`

`ACTION_CALL`

`ACTION_SENTTO`

`ACTION_ANSWER`

`ACTION_INSERT`

`ACTION_DELETE`

`ACTION_MAIN`

`ACTION_EDIT`

`ACTION_VIEW`

- For instance, to dial a phone number from within an application, you could use the following:

```
Uri theNumber = Uri.parse("tel:8005551212");
```

```
Intent dialMe= new Intent(Intent.ACTION_DIAL,theNumber);
```

```
startActivity(dialMe);
```

- Another possibility is to use an `onClick( )` or `onTouch( )` method belonging to a screen component to view a web page using the built-in action `ACTION_VIEW`. The target URL of the browser can be delivered along with the call based on some condition of the original screen. The code could look like this: `public void onClick(View v){`

```
String myUrl=someEditText.getText( ).toString( );
Intent myIntent=new Intent(Intent.ACTION_VIEW);
myIntent.setData(Uri.parse(myUrl));
startActivity(myIntent);
}
```

- Of course, the appropriate import statements to include the necessary classes are required, and the necessary permissions such as access to the Internet must be entered into the manifest file.
- To demonstrate a multiscreen application, we will make two screens, each containing a `Button` object that will be used to switch back and forth between screens, and a `TextView` to let us know which screen we are viewing at any given time.
- The first step will be to create two XML files for the screen design: the `main.xml` file and an additional file called `second.xml`. To create an additional screen XML file, right-click on the layout folder under the `res` folder in the Package Explorer, and choose `New, File` from the menu. Be sure to add `.xml` to the filename; the `.xml` file extension is required. You can leave the file blank for the time being.
- The `main.xml` file should be edited to look like this:

```
<?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:textSize="20px"
android:text="Primary Screen"
/>
<Button
android:id="@+id/SwitchToSecond"
android:layout_width="fill_parent"
android:layout_height="wrap_content"
```

```

android:textSize="20px"
android:text="Switch to Second Screen"
/>
</LinearLayout>

```

- For convenience sake, you can copy the entire contents of the main.xml file and paste it on your new, blank second.xml file. Make some edits to the second.xml file so that it looks like the following:

```

<?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:textSize="20px"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Secondary Screen"
    />
    <Button
        android:id="@+id/SwitchToMain"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20px"
        android:text="Back to Main Screen"
    />
</LinearLayout>

```

- Be particularly careful to change the id attribute in the Button tag so that you won't be confused when writing the Java code later. The XML files all tend to look the same, so it is a good idea to create id attributes that indicate which screen they belong to.
- The next step is to add the following lines to the manifest file to provide for the additional Activity class we will create for the second screen:

```

<activity android:name=".SecondScreen">
</activity>

```

- Be careful to place this inside the application tags, but outside any other activity tag pair, as shown in the following snippet:

```

<application
    android:icon="@drawable/icon"android:label="@string/app_name">
    <activity android:name=".MultiScreen"

```

```

android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<activity android:name=".SecondScreen">
</activity>
</application>

```

- Finally, we can work on our Java code. First, let's create a second Java class file called SecondScreen (remember: it is important that this class filename matches the activity name we just put in the manifest file) by right-clicking the package name under the src directory in the Package Explorer, as we did when we created the second XML file in the res\layout directory. This time choose New-Class. It will automatically have the .java extension added, so simply enter SecondScreen for the filename.

Initially, the file should look like the following:

```

package com.sheusi.MultiScreen;
import android.app.Activity;
import android.os.Bundle;
public class SecondScreen extends Activity{
    @Override
    public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.second);
    }
}

```

- Now let's turn our attention back to the main Java class, the MultiScreen.java file. We have added some additional import statements, a Button declaration and assignment, and most importantly, the onClick( ) listener that switches to the second screen. The listener is done as an "in-line" class declaration, which essentially creates a new OnClickListener class and defines its onClick( ) method all in one statement:

```

screen2Button.setOnClickListener(new View.OnClickListener(){
    public void onClick(View v){
    Intent myIntent=new Intent(v.getContext(),SecondScreen.class);
    startActivity(myIntent);
}
}

```

```
}  
});
```

- This is a common convenience in Java code as long as the `onClick( )` method we define never has to be used by any other control. In other words, no other Button object will need to share it. The complete MultiScreen.java file should look like this:

```
package com.sheusi.MultiScreen;  
import android.app.Activity;  
import android.os.Bundle;  
import android.widget.*;  
import android.view.*;  
import android.content.Intent;  
public class MultiScreen extends Activity {  
    /** Called when the activity is first created. */  
    Button screen2Button=null;  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
        screen2Button=(Button)findViewById(R.id.SwitchToSecond);  
        screen2Button.setOnClickListener(new View.OnClickListener(){  
            public void onClick(View v){  
                Intent myIntent=new Intent(v.getContext(),SecondScreen.class);  
                startActivity(myIntent);  
            }  
        });  
    }  
}
```

- If you run the application, you will see that clicking the button gets us to the second screen; we just cannot get back yet. To do this, we have to add more code to the SecondScreen.java file. We need to add the import statements that we added to the other Java file, the Button declaration and assignment, and finally the in-line View.OnClickListener class definition. You will see that the simple method call, `finish( )`, will get us back to the main screen.
- The SecondScreen.java file should now look like this:

```
package com.sheusi.MultiScreen;  
import android.app.Activity;  
import android.os.Bundle;  
import android.widget.*;
```

```

import android.view.*;
public class SecondScreen extends Activity{
    Button returnButton=null;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.second);
        returnButton=(Button)findViewById(R.id.SwitchToMain);
        returnButton.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v){
                finish();
            }
        });
    }
}

```

- You should be able to switch back and forth between screens. Again, this is an oversimplified example, but it lays a foundation for multiscreen applications.
- Applications with multiple activities can pass data back and forth in an Android class called the Bundle.
- The Bundle is a collection of key-value pairs associated with the Intent class; it is loaded and unloaded using the Intent's putExtra() and getExtras() methods.
- Remember that an application's main Activity can branch off into more than one secondary Activity, and they can either return to the main Activity or branch somewhere else; either pass data back or not; and so on.
- For the application programmer, this, of course, means more code, but it also means another level of design consideration because he must determine which data items are going where and whether they need to come back. To do this, we have to carefully plot data paths, name the bundles, and name the key-value pairs. It is a good idea to sketch this out before sitting down to code.
- The example presented next is based on the previous two-screen example, but it is done as a different project. You are certainly welcome to modify the code you wrote for the previous example for convenience. It will include an EditText field on the opening screen, which will allow the user to enter any text. When the user touches the button to go on to the secondary activity (screen), the text entered on the first screen will appear in the EditText field on the second screen. The user can edit the

text on the second screen, and the new text will be returned to the first screen when the button labeled Return is pressed.

- Following is a breakdown of the objects and labels used in the demonstration application:

### **Screen 1 (Activity 1)**

Key/value pair -> payload/contents of the EditText field

Bundle name when received at Screen 2 -> incoming

### **Screen 2 (Activity 2)**

Key/value pair -> returnpayload/contents of the EditText field

Bundle name when received back at Screen 1 -> response

- First let's take a look at the two XML files for the main and secondary screens. **main.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"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20px"
        android:text="Primary Screen"
    />
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Switch to Second Screen"

        android:textSize="20px"
        android:id="@+id/SwitchToSecond"
    />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=""
        android:textSize="20px"
        android:id="@+id/payloadValue"
    />
</LinearLayout>
```

## second.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"
    >
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:textSize="20px"
        android:text="Secondary Screen"
    />
    <Button
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Back to Main Screen"
        android:textSize="20px"
        android:id="@+id/SwitchToMain"
    />
    <EditText
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text=""
        android:textSize="20px"
        android:id="@+id/payloadValue"
    />
</LinearLayout>
```

- As you can see, the two files are nearly identical except for the text that appears on the buttons and some of the android:id fields. As in the previous example and with any multiactivity application, you need to make an entry in the manifest file for each Activity class file.
- Here is the manifest used for this project:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.sheusi.ActivitiesNBundles"
    android:versionCode="1"
    android:versionName="1.0">
    <uses-sdk android:minSdkVersion="8" />
```

```

<application android:icon="@drawable/icon"
android:label="@string/app_name">
<activity android:name=".ActivitiesNBundlesActivity"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<activity android:name=".SecondActivity">
</activity>
</application>
</manifest>

```

- Now let's take a look at the Java code for the first screen:

ActivitiesNBundlesActivity.java.

```

package com.sheusi.ActivitiesNBundles;
import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
import android.content.Intent;
public class ActivitiesNBundlesActivity extends Activity {
/** Called when the activity is first created. */
Button screen2Button=null;
EditText et=null;
private final int SECOND_ACTIVITY=2;
@Override
public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
screen2Button=(Button)findViewById(R.id.SwitchToSecond);
et=(EditText)findViewById(R.id.payloadValue);
screen2Button.setOnClickListener(new View.OnClickListener(){

public void onClick(View v){
Intent myIntent=new Intent(v.getContext(),SecondActivity.class);
myIntent.putExtra("payload",et.getText().toString());
//key-value pair
startActivityForResult(myIntent,SECOND_ACTIVITY);
}
});

```

```

}
@Override
//This method is necessary if you expect data to be returned
//when the second activity ends
public void onActivityResult(int requestCode, int resultCode, Intent i){
super.onActivityResult(requestCode, resultCode, i);
Bundle response=i.getExtras();
Et.setText(response.getString("returnpayload"));
}
}

```

- First, be sure to import the appropriate packages.
- Next, notice that we use a symbolic constant to represent the second screen.
- Remember that data in bundles is associated with specific Intent objects, so we want to associate the right Intent with the right target Activity.
- In other words, we want to deliver the right package(s) to the right address. You will see where this symbolic constant is used near the end of the code.
- As in the previous example, clicking the button creates a new Intent object. What is different this time is that we add a piece of data in a name-value pair to the Intent using the `putExtra( )` method.
- Finally, because we expect to get something back, we use the `startActivityForResult( )` method instead of `startActivity( )`; and we override the `onActivityResult( )` method to pull off the data and do what we want with it.

- Here is the Java code for the second screen:

```

package com.sheusi.ActivitiesNBundles;
import android.app.Activity;
import android.os.Bundle;
import android.widget.*;
import android.view.*;
import android.content.Intent;
public class SecondActivity extends Activity {
  /** Called when the activity is first created. */
  Button returnButton=null;
  EditText et2s=null;
  @Override

```

```

public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.second);
    returnButton=(Button)findViewById(R.id.SwitchToMain);
    et2s=(EditText)findViewById(R.id.payloadValue);
    Bundle incoming=this.getIntent().getExtras();
    et2s.setText(incoming.getString("payload"));
    returnButton.setOnClickListener(new View.OnClickListener(){
    public void onClick(View v){
        Intent goBack=new Intent();
        goBack.putExtra("returnpayload",et2s.getText().toString());
        setResult(RESULT_OK, goBack);
        finish();
    }
    }
    );
}

```

- The big difference here is that you can see the declaration of the Bundle object and its assignment using the Intent's getExtras( ) method. Remember that more than one key-value pair can be inserted into a bundle, so we use the key name to identify the data item we want to use at a given time.
- Notice that it is used as the parameter in the getString( ) method of the bundle's instance when it is inserted into the EditText field.
- Finally, the same process is used again to send the edited text value back to the main screen when the button is clicked.
- A new Intent object is instantiated, and a key-value pair is created. This time the key is named returnpayload and it is attached to the Intent; then the second screen is finished. The parameter, RESULT\_OK, is a value for a field of the Activity class, which signals that the Activity was completed correctly.