29. An Introduction to Android Fragments

As you progress through the chapters of this book it will become increasingly evident that many of the design concepts behind the Android system were conceived with the goal of promoting reuse of, and interaction between, the different elements that make up an application. One such area that will be explored in this chapter involves the use of Fragments.

This chapter will provide an overview of the basics of fragments in terms of what they are and how they can be created and used within applications. The next chapter will work through a tutorial designed to show fragments in action when developing applications in Android Studio, including the implementation of communication between fragments.

29.1 What is a Fragment?

A fragment is a self-contained, modular section of an application's user interface and corresponding behavior that can be embedded within an activity. Fragments can be assembled to create an activity during the application design phase, and added to or removed from an activity during application runtime to create a dynamically changing user interface.

Fragments may only be used as part of an activity and cannot be instantiated as standalone application elements. That being said, however, a fragment can be thought of as a functional "sub-activity" with its own lifecycle similar to that of a full activity.

Fragments are stored in the form of XML layout files and may be added to an activity either by placing appropriate <fragment> elements in the activity's layout file, or directly through code within the activity's class implementation.

Before starting to use Fragments in an Android application, it is important to be aware that Fragments were not introduced to Android until version 3.0 of the Android SDK. An application that uses Fragments must, therefore, make use of the *android-support-v4* Android Support Library in order to be compatible with older Android versions. The steps to achieve this will be covered in the next chapter, entitled *Using Fragments in Android Studio - A Worked Example*.

29.2 Creating a Fragment

The two components that make up a fragment are an XML layout file and a corresponding Java class. The XML layout file for a fragment takes the same format as a layout for any other activity layout and can contain any combination and complexity of layout managers and views. The following XML layout, for example, is for a fragment consisting simply of a RelativeLayout with a red background containing a single TextView:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/red" >

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="@string/fragone_label_text"
        android:textAppearance="?android:attr/textAppearanceLarge" />
</RelativeLayout>
```

The corresponding class to go with the layout must be a subclass of the Android *Fragment* class. If the application is to be compatible with devices running versions of Android predating version 3.0 then the class file must import *android.support.v4.app.Fragment*. The class should, at a minimum, override the *onCreateView()* method which is responsible for loading the fragment layout. For example:

```
package com.example.myfragmentdemo;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class FragmentOne extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
             ViewGroup container,
              Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_one_layout,
                      container, false);
    }
}
```

In addition to the *onCreateView()* method, the class may also override the standard lifecycle methods.

Note that in order to make the above fragment compatible with Android versions prior to version 3.0, the Fragment class from the v4 support library has been imported.

Once the fragment layout and class have been created, the fragment is ready to be used within application activities.

29.3 Adding a Fragment to an Activity using the Layout XML File

Fragments may be incorporated into an activity either by writing Java code or by embedding the fragment into the activity's XML layout file. Regardless of the approach used, a key point to be aware of is that when the support library is being used for compatibility with older Android releases, any activities using fragments must be implemented as a subclass of *FragmentActivity* instead of the *AppCompatActivity* class:

```
package com.example.myfragmentdemo;

import android.os.Bundle;
import android.support.v4.app.FragmentActivity;
import android.view.Menu;

public class FragmentDemoActivity extends FragmentActivity {
       @Override
       protected void onCreate(Bundle savedInstanceState) {
              super.onCreate(savedInstanceState);
              setContentView(R.layout.activity_fragment_demo);
       }

       @Override
       public boolean onCreateOptionsMenu(Menu menu) {
              getMenuInflater().inflate(R.menu.activity_fragment_demo,
                         menu);
              return true;
       }
}
```

Fragments are embedded into activity layout files using the <fragment> element. The following example layout embeds the fragment created in the previous section of this chapter into an activity layout:

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".FragmentDemoActivity" >

    <fragment
        android:id="@+id/fragment_one"
        android:name="com.example.myfragmentdemo.myfragmentdemo.FragmentOne"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_centerVertical="true"
        tools:layout="@layout/fragment_one_layout" />

</RelativeLayout>
```

The key properties within the <fragment> element are *android:name*, which must reference the class associated with the fragment, and *tools:layout*, which must reference the XML resource file containing the layout of the fragment.

Once added to the layout of an activity, fragments may be viewed and manipulated within the Android Studio Layout Editor tool. Figure 29-1, for example, shows the above layout with the embedded fragment within the Android Studio Layout Editor:

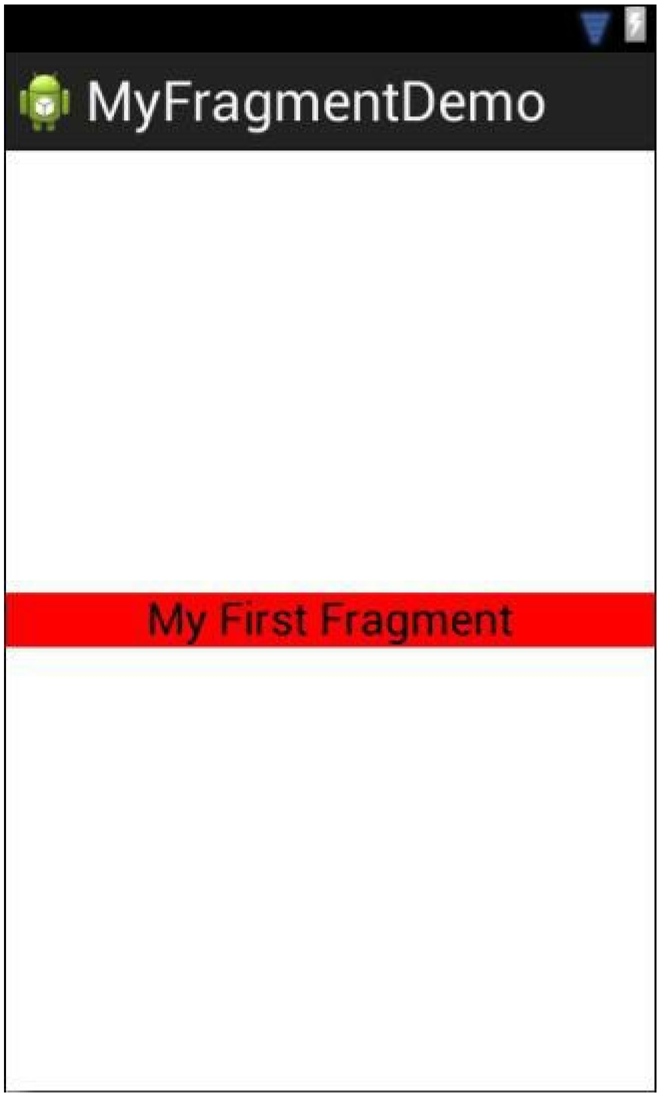


Figure 29-1

29.4 Adding and Managing Fragments in Code

The ease of adding a fragment to an activity via the activity's XML layout file comes at the cost of the activity not being able to remove the fragment at runtime. In order to achieve full dynamic control of fragments during runtime, those activities must be added via code. This has the advantage that the fragments can be added, removed and even made to replace one another dynamically while the application is running.

When using code to manage fragments, the fragment itself will still consist of an XML layout file and a corresponding class. The difference comes when working with the fragment within the hosting activity. There is a standard sequence of steps when adding a fragment to an activity using code:

1. Create an instance of the fragment's class.
2. Pass any additional intent arguments through to the class.
3. Obtain a reference to the *fragment manager* instance.
4. Call the *beginTransaction()* method on the fragment manager instance. This returns a *fragment transaction* instance.
5. Call the *add()* method of the fragment transaction instance, passing through as arguments the resource ID of the view that is to contain the fragment and the fragment class instance.
6. Call the *commit()* method of the fragment transaction.

The following code, for example, adds a fragment defined by the FragmentOne class so that it appears in the container view with an ID of LinearLayout1:

```
FragmentOne firstFragment = new FragmentOne();
firstFragment.setArguments(getIntent().getExtras());

FragmentManager fragManager = getSupportFragmentManager();
FragmentTransaction transaction = fragManager.beginTransaction();

transaction.add(R.id.LinearLayout1, firstFragment);
transaction.commit();
```

The above code breaks down each step into a separate statement for the purposes of clarity. The last four lines can, however, be abbreviated into a single line of code as follows:

```
getSupportFragmentManager().beginTransaction()
        .add(R.id.LinearLayout1, firstFragment).commit();
```

Once added to a container, a fragment may subsequently be removed via a call to the *remove()* method of the fragment transaction instance, passing through a reference to the fragment instance that is to be removed:

```
transaction.remove(firstFragment);
```

Similarly, one fragment may be replaced with another by a call to the *replace()* method of the fragment transaction instance. This takes as arguments the ID of the view containing the fragment and an instance of the new fragment. The replaced fragment may also be placed on what is referred to as the *back* stack so that it can be quickly restored in the event that the user navigates back to it. This is achieved by making a call to the *addToBackStack()* method of the fragment transaction object before making the *commit()* method call:

```
FragmentTwo secondFragment = new FragmentTwo();
transaction.replace(R.id.LinearLayout1, secondFragment);
transaction.addToBackStack(null);
transaction.commit();
```

29.5 **Handling Fragment Events**

As previously discussed, a fragment is very much like a sub-activity with its own layout, class and lifecycle. The view components (such as buttons and text views) within a fragment are able to generate events just like those in a regular activity. This raises the question as to which class receives an event from a view in a fragment; the fragment itself, or the activity in which the fragment is embedded. The answer to this question depends on how the event handler is declared.

In the chapter entitled *An Overview and Example of Android Event Handling*, two approaches to event handling were discussed. The first method involved configuring an event listener and callback method within the code of the activity. For example:

```
Button button = (Button)findViewById(R.id.myButton);

button.setOnClickListener(
        new Button.OnClickListener() {
              public void onClick(View v) {
                        // Code to be performed when
                        // the button is clicked
                    }
              }
```

```
            );
```

In the case of intercepting click events, the second approach involved setting the *android:onClick* property within the XML layout file:

```
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onClick"
    android:text="Click me" />
```

The general rule for events generated by a view in a fragment is that if the event listener was declared in the fragment class using the event listener and callback method approach, then the event will be handled first by the fragment. If the *android:onClick* resource is used, however, the event will be passed directly to the activity containing the fragment.

29.6 **Implementing Fragment Communication**

Once one or more fragments are embedded within an activity, the chances are good that some form of communication will need to take place both between the fragments and the activity, and between one fragment and another. In fact, good practice dictates that fragments do not communicate directly with one another. All communication should take place via the encapsulating activity.

In order for an activity to communicate with a fragment, the activity must identify the fragment object via the ID assigned to it using the *findViewById()* method. Once this reference has been obtained, the activity can simply call the public methods of the fragment object.

Communicating in the other direction (from fragment to activity) is a little more complicated. In the first instance, the fragment must define a listener interface, which is then implemented within the activity class. For example, the following code declares an interface named ToolbarListener on a fragment class named ToolbarFragment. The code also declares a variable in which a reference to the activity will later be stored:

```
public class ToolbarFragment extends Fragment {

        ToolbarListener activityCallback;

        public interface ToolbarListener {
                public void onButtonClick(int position, String text);
        }
.
.
}
```

The above code dictates that any class that implements the ToolbarListener interface must also implement a callback method named *onButtonClick* which, in turn, accepts an integer and a String as arguments.

Next, the *onAttach()* method of the fragment class needs to be overridden and implemented. This method is called automatically by the Android system when the fragment has been initialized and associated with an activity. The method is passed a reference to the activity in which the fragment is contained. The method must store a local reference to this activity and verify that it implements the ToolbarListener interface:

```
    @Override
    public void onAttach(Context context) {
        super.onAttach(context);

        try {
           activityCallback = (ToolbarListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString()
                    + " must implement ToolbarListener");
        }
    }
```

Upon execution of this example, a reference to the activity will be stored in the local *activityCallback* variable, and an exception will be thrown if that activity does not implement the ToolbarListener interface.

The next step is to call the callback method of the activity from within the fragment. When and how this happens is entirely dependent on the circumstances under which the activity needs to be contacted by the fragment. The following code, for example, calls the callback method on the activity when a button is clicked:

```
    public void buttonClicked (View view) {
       activityCallback.onButtonClick(arg1, arg2);
    }
```

All that remains is to modify the activity class so that it implements the ToolbarListener interface. For example:

```
    public class FragmentExampleActivity extends FragmentActivity implements
    ToolbarFragment.ToolbarListener {
           public void onButtonClick(String arg1, int arg2) {
          // Implement code for callback method
           }
    .
    .
    }
```

As we can see from the above code, the activity declares that it implements the ToolbarListener interface of the ToolbarFragment class and then proceeds to implement the *onButtonClick()* method as required by the interface.

29.7 **Summary**

Fragments provide a powerful mechanism for creating re-usable modules of user interface layout and application behavior, which, once created, can be embedded in activities. A fragment consists of a user interface layout file and a class. Fragments may be utilized in an activity either by adding the fragment to the activity's layout file, or by writing code to manage the fragments at runtime. Fragments added to an activity in code can be removed and replaced dynamically at runtime. All communication between fragments should be performed via the activity within which the activities are embedded.

Having covered the basics of fragments in this chapter, the next chapter will work through a tutorial designed to reinforce the techniques outlined in this chapter.

30. Using Fragments in Android Studio - An Example

As outlined in the previous chapter, fragments provide a convenient mechanism for creating reusable modules of application functionality consisting of both sections of a user interface and the corresponding behavior. Once created, fragments can be embedded within activities.

Having explored the overall theory of fragments in the previous chapter, the objective of this chapter is to create an example Android application using Android Studio designed to demonstrate the actual steps involved in both creating and using fragments, and also implementing communication between one fragment and another within an activity.

30.1 About the Example Fragment Application

The application created in this chapter will consist of a single activity and two fragments. The user interface for the first fragment will contain a toolbar of sorts consisting of an EditText view, a SeekBar and a Button, all contained within a RelativeLayout view. The second fragment will consist solely of a TextView object, also contained within a RelativeLayout view.

The two fragments will be embedded within the main activity of the application and communication implemented such that when the button in the first fragment is pressed, the text entered into the EditText view will appear on the TextView of the second fragment using a font size dictated by the position of the SeekBar in the first fragment.

Since this application is intended to work on earlier versions of Android, it will also be necessary to make use of the appropriate Android support library.

30.2 Creating the Example Project

Create a new project in Android Studio, entering *FragmentExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *FragmentExampleActivity* with a corresponding layout resource file named *activity_fragment_example.*

Click the *Finish* button to begin the project creation process.

30.3 Creating the First Fragment Layout

The next step is to create the user interface for the first fragment that will be used within our activity.

This user interface will, of course, reside in an XML layout file so begin by navigating to the *layout* folder located under *app -> res* in the Project tool window. Once located, right-click on the *layout* entry and select the *New -> Layout resource file* menu option as illustrated in Figure 30-1:

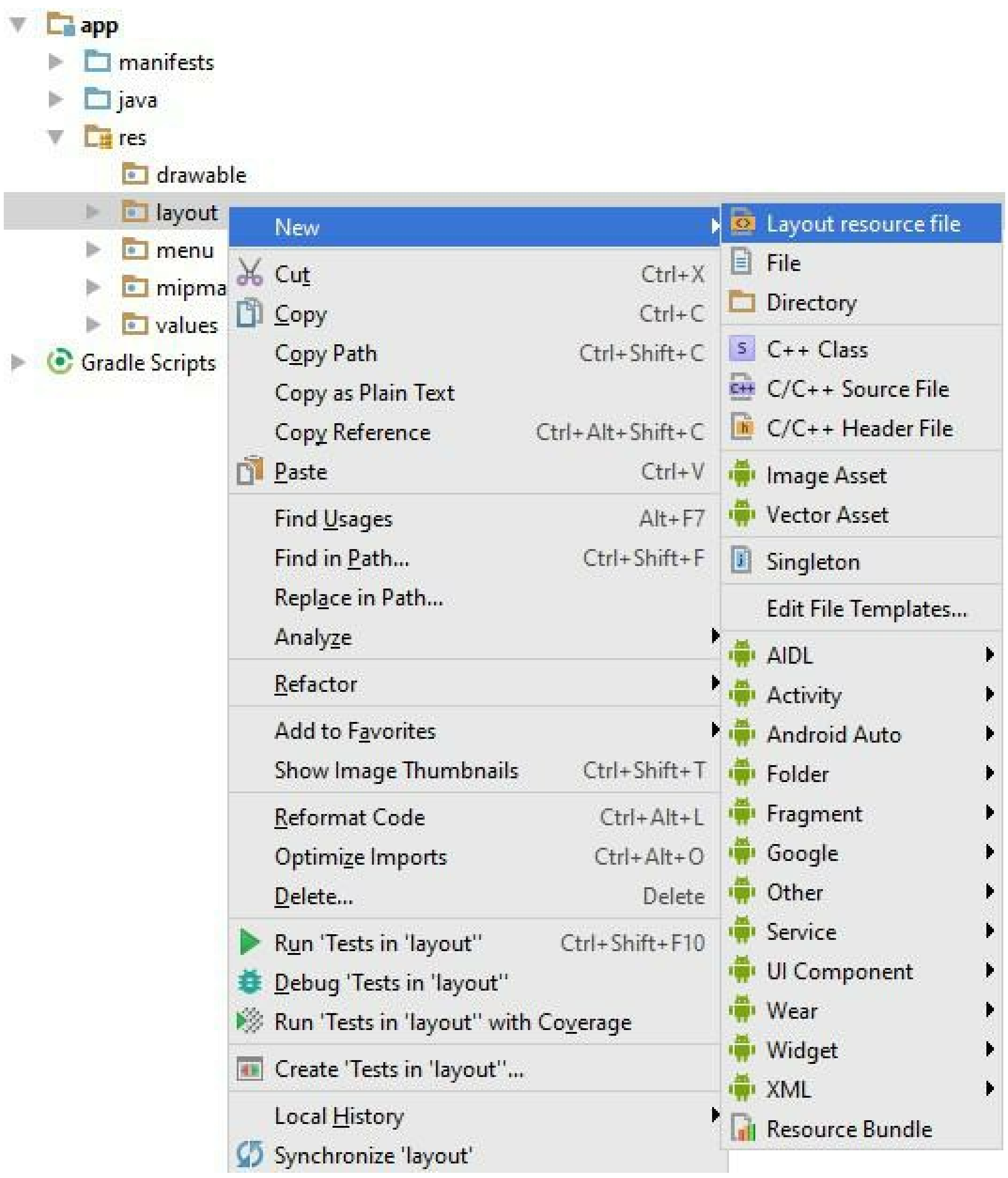


Figure 30-1

In the resulting dialog, name the layout *toolbar_fragment* and change the root element from LinearLayout to RelativeLayout before clicking on OK to create the new resource file.

The new resource file will appear within the Layout Editor tool ready to be designed. Switch the Layout Editor to Text mode and modify the XML so that it reads as outlined in the following listing to add three new view elements to the layout:

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_below="@+id/seekBar1"
        android:layout_centerHorizontal="true"
```

```
            android:layout_marginTop="17dp"
            android:text="Change Text" />

        <EditText
            android:id="@+id/editText1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentTop="true"
            android:layout_centerHorizontal="true"
            android:layout_marginTop="16dp"
            android:ems="10"
            android:inputType="text" >
            <requestFocus />
        </EditText>

        <SeekBar
            android:id="@+id/seekBar1"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_alignParentStart="true"
            android:layout_below="@+id/editText1"
            android:layout_marginTop="14dp"
            android:layout_alignParentLeft="true" />

    </RelativeLayout>
```

Once the changes have been made, switch the Layout Editor tool back to Design mode and click on the red warning button in the top right-hand corner of the design area. Select the I18N warning, click the *Extract string resource* link and assign the string to a resource named *change_text*.

Upon completion of these steps, the user interface layout should resemble that of Figure 30-2:

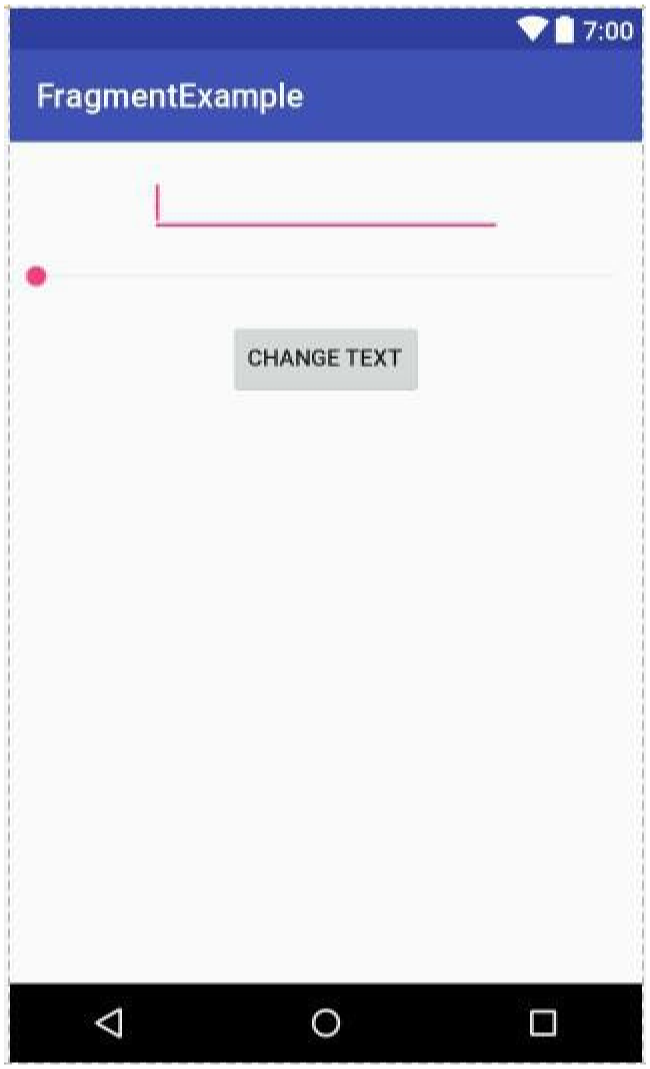


Figure 30-2

With the layout for the first fragment implemented, the next step is to create a class to go with it.

30.4 Creating the First Fragment Class

In addition to a user interface layout, a fragment also needs to have a class associated with it to do the actual work behind the scenes. Add a class for this purpose to the project by unfolding the *app* -> *java* folder under the FragmentExample project in the Project tool window and right-clicking on the package name given to the project when it was created (in this instance *com.ebookfrenzy.fragmentexample*). From the resulting menu, select the *New* -> *Java Class* option. In the resulting *Create New Class* dialog, name the class *ToolbarFragment* and click on *OK* to create the new class.

Once the class has been created, it should, by default, appear in the editing panel where it will read as follows:

```
package com.ebookfrenzy.fragmentexample;

/**
 * Created by <name> on <date>.
 */
public class ToolbarFragment {
}
```

For the time being, the only changes to this class are the addition some import directives and the overriding of the *onCreateView()* method to make sure the layout file is inflated and displayed when the fragment is used within an activity. The class declaration also needs to indicate that the class extends the Android Fragment class:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class ToolbarFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container, Bundle
                              savedInstanceState) {

        // Inflate the layout for this fragment
        View view =  inflater.inflate(R.layout.toolbar_fragment,
                container, false);
        return view;
    }
}
```

Later in this chapter, more functionality will be added to this class. Before that, however, we need to create the second fragment.

30.5 Creating the Second Fragment Layout

Add a second new Android XML layout resource file to the project, once again selecting a RelativeLayout as the root element. Name the layout *text_fragment* and click *OK*. When the layout loads into the Layout Editor tool, change to Text mode and modify the XML to add a TextView to the fragment layout as follows:

```
<?xml version="1.0" encoding="utf-8"?>

<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_centerVertical="true"
        android:text="Fragment Two"
        android:textAppearance="?android:attr/textAppearanceLarge" />

</RelativeLayout>
```

Once the XML changes have been made, switch back to Design mode, extract the string to a resource named *fragment_two*. Upon completion of these steps, the user interface layout for this second fragment should resemble that of Figure 30-3:

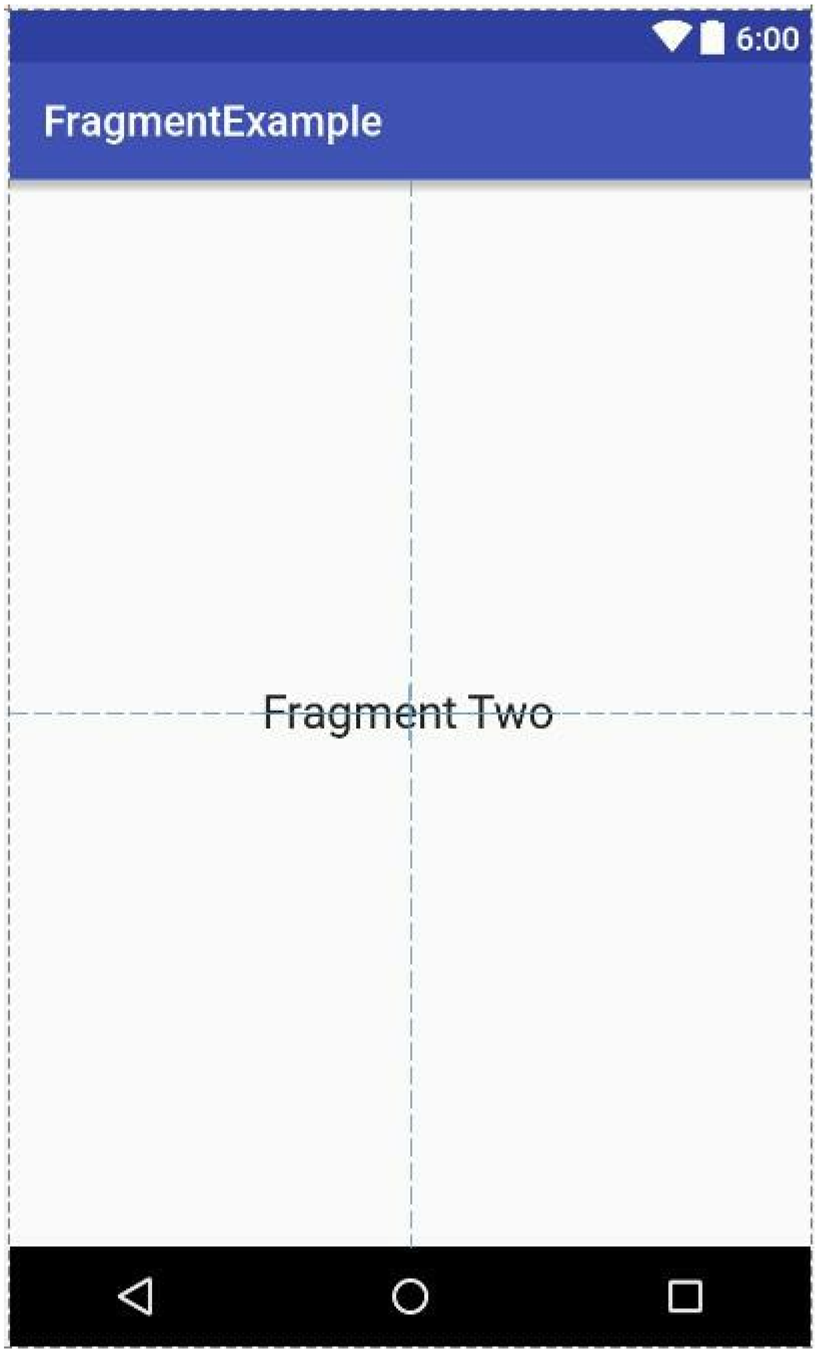


Figure 30-3

As with the first fragment, this one will also need to have a class associated with it. Right-click on *app -> java -> com.ebookfrenzy.fragmentexample* in the Project tool window. From the resulting menu, select the *New -> Java Class* option. Name the fragment *TextFragment* and click *OK* to create the class.

Edit the new *TextFragment.java* class file and modify it to implement the *onCreateView()* method and designate the class as extending the Android *Fragment* class:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
```

```
public class TextFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.text_fragment,
                container, false);

        return view;
    }
}
```

Now that the basic structure of the two fragments has been implemented, they are ready to be embedded in the application's main activity.

30.6 Adding the Fragments to the Activity

The main activity for the application has associated with it an XML layout file named *activity_fragment_example.xml*. For the purposes of this example, the fragments will be added to the activity using the <fragment> element within this file. Using the Project tool window, navigate to the *app -> res -> layout* section of the *FragmentExample* project and double-click on the *activity_fragment_example.xml* file to load it into the Android Studio Layout Editor tool.

With the Layout Editor tool in Design mode, select and delete the default TextView object from the layout and select the *Layouts* category in the palette. Drag the *<fragment>* component from the list of layouts and drop it onto the layout so that it is centered horizontally and positioned such that the dashed line appears indicating the top layout margin:

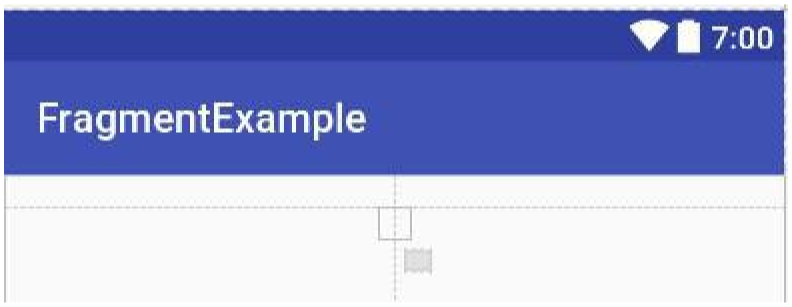
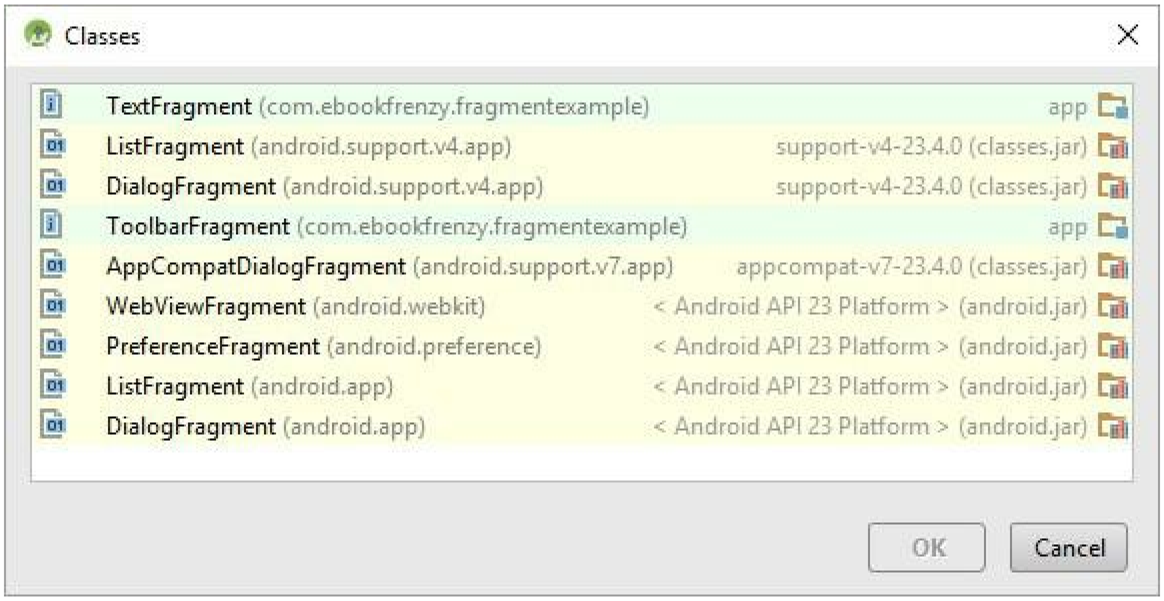


Figure 30-4

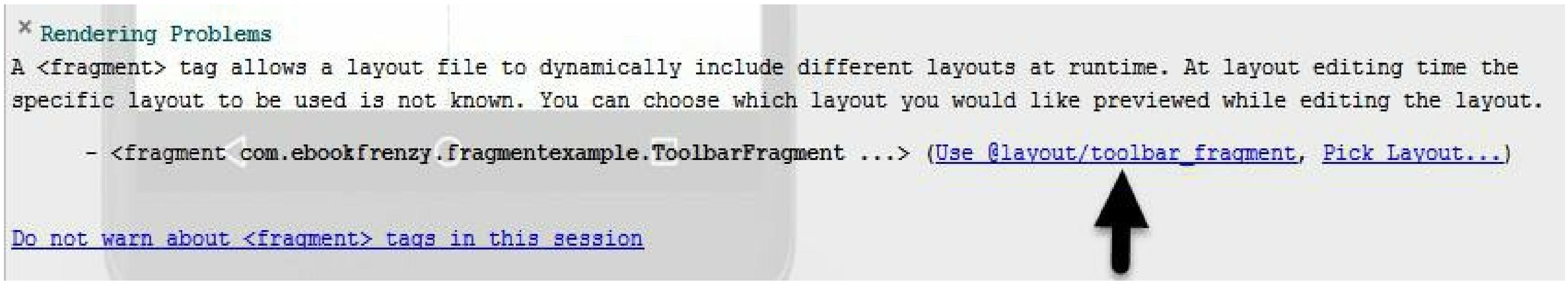
On dropping the fragment onto the layout, a dialog will appear displaying a list of Fragments available within the current project as illustrated in Figure 30-5:


Classes

TextFragment (com.ebookfrenzy.fragmentexample)	app
ListFragment (android.support.v4.app)	support-v4-23.4.0 (classes.jar)
DialogFragment (android.support.v4.app)	support-v4-23.4.0 (classes.jar)
ToolbarFragment (com.ebookfrenzy.fragmentexample)	app
AppCompatDialogFragment (android.support.v7.app)	appcompat-v7-23.4.0 (classes.jar)
WebViewFragment (android.webkit)	< Android API 23 Platform > (android.jar)
PreferenceFragment (android.preference)	< Android API 23 Platform > (android.jar)
ListFragment (android.app)	< Android API 23 Platform > (android.jar)
DialogFragment (android.app)	< Android API 23 Platform > (android.jar)

Figure 30-5

Select the ToolbarFragment entry from the list and click on the OK button to dismiss the Fragments dialog. Once added, a message panel will appear (Figure 30-6) indicating that the Layout Editor tool needs to know which fragment to display during the preview session. Display the ToolbarFragment fragment by clicking on the *Use @layout/toolbar_fragment* link within the message:


```
Rendering Problems
A <fragment> tag allows a layout file to dynamically include different layouts at runtime. At layout editing time the
specific layout to be used is not known. You can choose which layout you would like previewed while editing the layout.

    - <fragment com.ebookfrenzy.fragmentexample.ToolbarFragment ...> (Use @layout/toolbar_fragment, Pick Layout...)

Do not warn about <fragment> tags in this session
```

Figure 30-6

Click and drag another <fragment> entry from the panel and positioning it so that it is centered horizontally and positioned beneath the bottom edge of the first fragment. When prompted, select the *TextFragment* entry from the fragment dialog before clicking on the OK button. When the rendering message appears, click on the *Use @layout/text_fragment* option. Establish a constraint connection between the top edge of the TextFragment and the bottom edge of the ToolbarFragment.

Note that the fragments are now visible in the layout as demonstrated in Figure 30-7:

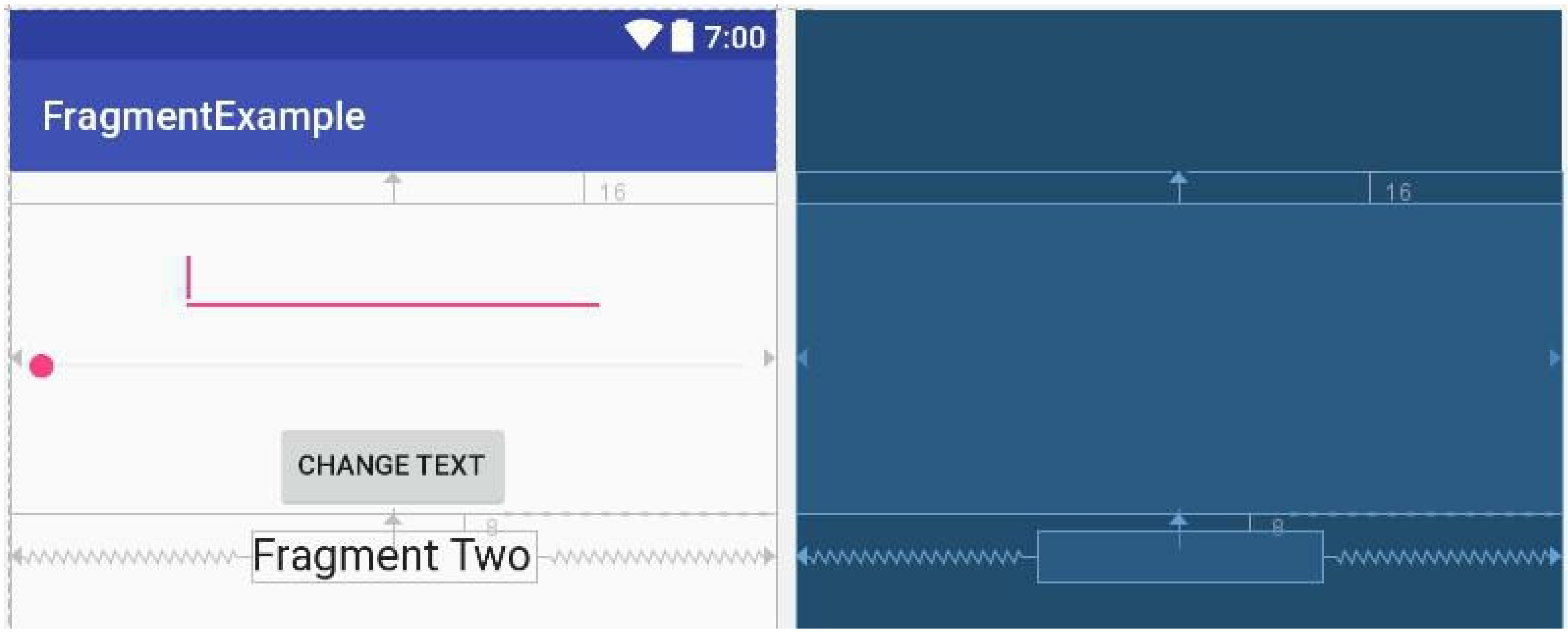


Figure 30-7

Before proceeding to the next step, select the TextFragment instance in the layout and, within the Properties tool window, change the ID of the fragment to *text_fragment*.

30.7 Making the Toolbar Fragment Talk to the Activity

When the user touches the button in the toolbar fragment, the fragment class is going to need to get the text from the EditText view and the current value of the SeekBar and send them to the text fragment. As outlined in *An Introduction to Android Fragments*, fragments should not communicate with each other directly, instead using the activity in which they are embedded as an intermediary.

The first step in this process is to make sure that the toolbar fragment responds to the button being clicked. We also need to implement some code to keep track of the value of the SeekBar view. For the purposes of this example, we will implement these listeners within the ToolbarFragment class. Select the *ToolbarFragment.java* file and modify it so that it reads as shown in the following listing:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.content.Context;
import android.widget.Button;
import android.widget.EditText;
import android.widget.SeekBar;
import android.widget.SeekBar.OnSeekBarChangeListener;

public class ToolbarFragment extends Fragment implements
OnSeekBarChangeListener {

    private static int seekvalue = 10;
    private static EditText edittext;

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container, Bundle
                              savedInstanceState) {

        // Inflate the layout for this fragment
        View view =  inflater.inflate(R.layout.toolbar_fragment,
                container, false);

        edittext = (EditText) view.findViewById(R.id.editText1);
        final SeekBar seekbar =
                (SeekBar) view.findViewById(R.id.seekBar1);

        seekbar.setOnSeekBarChangeListener(this);

        final Button button =
                (Button) view.findViewById(R.id.button1);
        button.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                buttonClicked(v);
            }
        });

        return view;
    }

    public void buttonClicked (View view) {

    }

    @Override
    public void onProgressChanged(SeekBar seekBar, int progress,
                                  boolean fromUser) {
        seekvalue = progress;
    }

    @Override
    public void onStartTrackingTouch(SeekBar arg0) {

    }

    @Override
    public void onStopTrackingTouch(SeekBar arg0) {

    }
}
```

Before moving on, we need to take some time to explain the above code changes. First, the class is declared as implementing the OnSeekBarChangeListener interface. This is because the user interface contains a SeekBar instance and the fragment needs to receive notifications when the user slides the bar to change the font size. Implementation of the OnSeekBarChangeListener interface requires that the *onProgressChanged()*, *onStartTrackingTouch()* and *onStopTrackingTouch()* methods be implemented. These methods have been implemented but only the *onProgressChanged()* method is actually required to perform a task, in this case storing the new value in a variable named seekvalue which has been declared at the start of the class. Also declared is a variable in which to store a reference to the EditText object.

The *onCreateView()* method has been modified to obtain references to the EditText, SeekBar and Button views in the layout. Once a reference to the button has been obtained it is used to set up an onClickListener on the button which is configured to call a method named *buttonClicked()* when a click event is detected. This method is also then implemented, though at this point it does not do anything.

The next phase of this process is to set up the listener that will allow the fragment to call the activity when the button is clicked. This follows the mechanism outlined in the previous chapter:

```
public class ToolbarFragment extends Fragment implements
OnSeekBarChangeListener {

         private static int seekvalue = 10;
         private static EditText edittext;

         ToolbarListener activityCallback;

         public interface ToolbarListener {
               public void onButtonClick(int position, String text);
         }

         @Override
         public void onAttach(Context context) {
               super.onAttach(context);
               try {
                   activityCallback = (ToolbarListener) context;
               } catch (ClassCastException e) {
                   throw new ClassCastException(context.toString()
                       + " must implement ToolbarListener");
               }
          }

          @Override
           public View onCreateView(LayoutInflater inflater,
              ViewGroup container, Bundle savedInstanceState) {
               // Inflate the layout for this fragment

               View view =
                      inflater.inflate(R.layout.toolbar_fragment,
                              container, false);

               edittext = (EditText)
                      view.findViewById(R.id.editText1);
               final SeekBar seekbar =
                      (SeekBar) view.findViewById(R.id.seekBar1);

               seekbar.setOnSeekBarChangeListener(this);

               final Button button =
                 (Button) view.findViewById(R.id.button1);
               button.setOnClickListener(new View.OnClickListener() {
                   public void onClick(View v) {
                       buttonClicked(v);
                   }
               });

               return view;
```

```
            }

            public void buttonClicked (View view) {
                    activityCallback.onButtonClick(seekvalue,
                        edittext.getText().toString());
            }
    .
    .
    .
    }
```

The above implementation will result in a method named *onButtonClick()* belonging to the activity class being called when the button is clicked by the user. All that remains, therefore, is to declare that the activity class implements the newly created ToolbarListener interface and to implement the *onButtonClick()* method.

Since the Android Support Library is being used for fragment support in earlier Android versions, the activity also needs to be changed to subclass from *FragmentActivity* instead of *AppCompatActivity*. Bringing these requirements together results in the following modified *FragmentExampleActivity.java* file:

```
    package com.ebookfrenzy.fragmentexample;

    ~~import android.support.v7.app.AppCompatActivity;~~
    import android.support.v4.app.FragmentActivity;
    import android.os.Bundle;

    public class FragmentExampleActivity extends FragmentActivity implements
    ToolbarFragment.ToolbarListener {

        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_fragment_example);
        }

        public void onButtonClick(int fontsize, String text) {

        }
    .
    .
    .
    }
```

With the code changes as they currently stand, the toolbar fragment will detect when the button is clicked by the user and call a method on the activity passing through the content of the EditText field and the current setting of the SeekBar view. It is now the job of the activity to communicate with the Text Fragment and to pass along these values so that the fragment can update the TextView object accordingly.

30.8 **Making the Activity Talk to the Text Fragment**

As outlined in *An Introduction to Android Fragments*, an activity can communicate with a fragment by obtaining a reference to the fragment class instance and then calling public methods on the object.

As such, within the TextFragment class we will now implement a public method named *changeTextProperties()* which takes as arguments an integer for the font size and a string for the new text to be displayed. The method will then use these values to modify the TextView object. Within the Android Studio editing panel, locate and modify the *TextFragment.java* file to add this new method and to add code to the *onCreateView()* method to obtain the ID of the TextView object:

```
package com.ebookfrenzy.fragmentexample;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

public class TextFragment extends Fragment {

    private static TextView textview;

    @Override
    public View onCreateView(LayoutInflater inflater,
                             ViewGroup container,
                             Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.text_fragment,
                container, false);

        textview = (TextView) view.findViewById(R.id.textView1);

        return view;
    }

    public void changeTextProperties(int fontsize, String text)
    {
        textview.setTextSize(fontsize);
        textview.setText(text);
    }
}
```

When the TextFragment fragment was placed in the layout of the activity, it was given an ID of *text_fragment*. Using this ID, it is now possible for the activity to obtain a reference to the fragment instance and call the *changeTextProperties()* method on the object. Edit the *FragmentExampleActivity.java* file and modify the *onButtonClick()* method as follows:

```
public void onButtonClick(int fontsize, String text) {

   TextFragment textFragment =
    (TextFragment)
      getSupportFragmentManager().findFragmentById(R.id.text_fragment);

   textFragment.changeTextProperties(fontsize, text);
}
```

30.9 Testing the Application

With the coding for this project now complete, the last remaining task is to run the application. When the application is launched, the main activity will start and will, in turn, create and display the two fragments. When the user touches the button in the toolbar fragment, the *onButtonClick()* method of the activity will be called by the toolbar fragment and passed the text from the EditText view and the current value of the SeekBar. The activity will then call the *changeTextProperties()* method of the second fragment, which will modify the TextView to reflect the new text and font size:

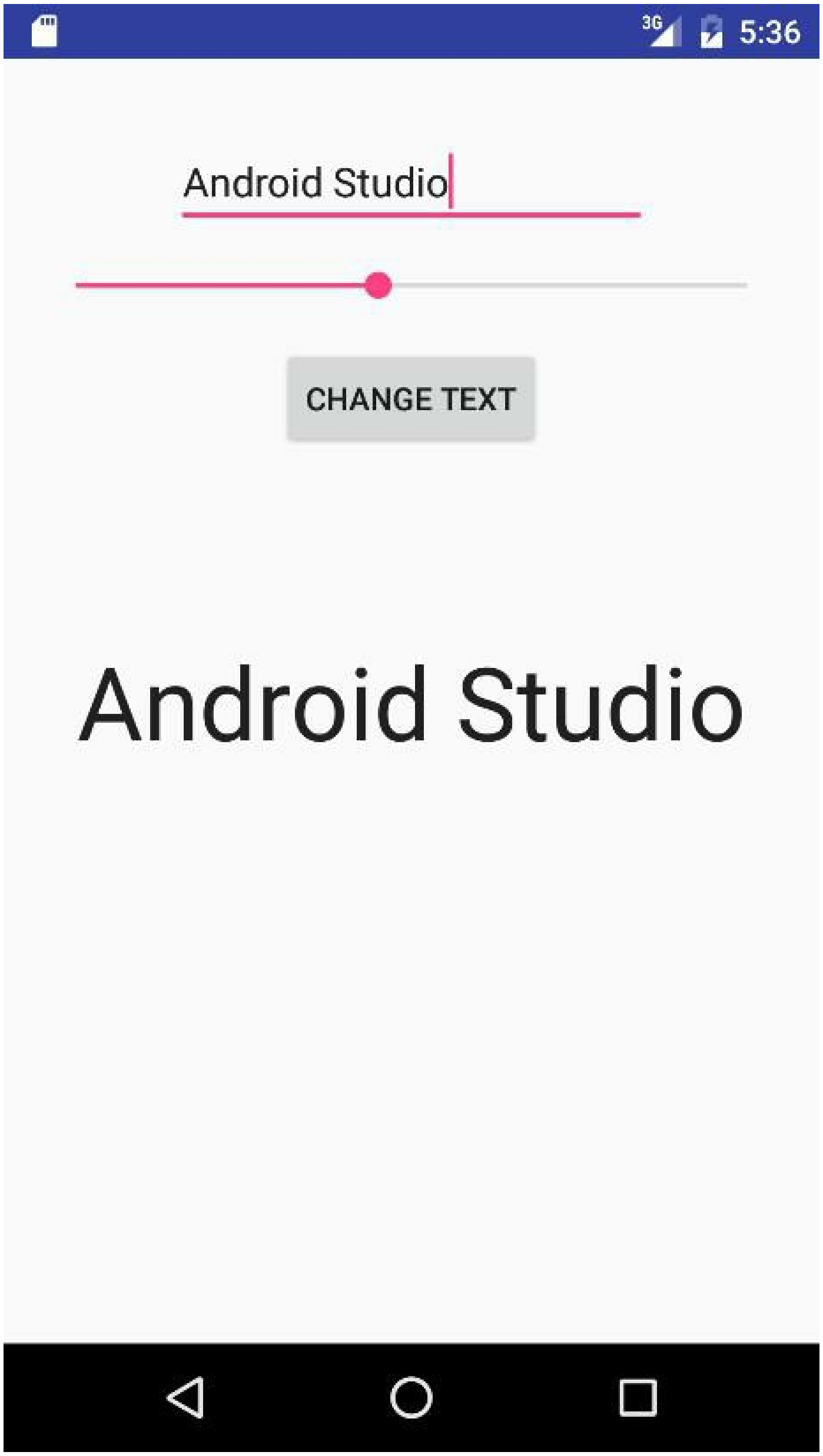


Figure 30-8

30.10 **Summary**

The goal of this chapter was to work through the creation of an example project intended specifically to demonstrate the steps involved in using fragments within an Android application. Topics covered included the use of the Android Support Library for compatibility with Android versions predating the introduction of fragments, the inclusion of fragments within an activity layout and the implementation of inter-fragment communication.

31. Creating and Managing Overflow Menus on Android

An area of user interface design that has not yet been covered in this book relates to the concept of menus within an Android application. Menus provide a mechanism for offering additional choices to the user beyond the view components that are present in the user interface layout. While there are a number of different menu systems available to the Android application developer, this chapter will focus on the more commonly used Overflow menu. The chapter will cover the creation of menus both manually via XML and visually using the Android Studio Layout Editor tool.

31.1 The Overflow Menu

The overflow menu (also referred to as the options menu) is a menu that is accessible to the user from the device display and allows the developer to include other application options beyond those included in the user interface of the application. The location of the overflow menu is dependent upon the version of Android that is running on the device. On a device running Android 2.3.3, for example, the overflow menu is represented by the menu icon located in the center (between the back and search buttons) of the bottom soft key toolbar as illustrated in Figure 31-1:

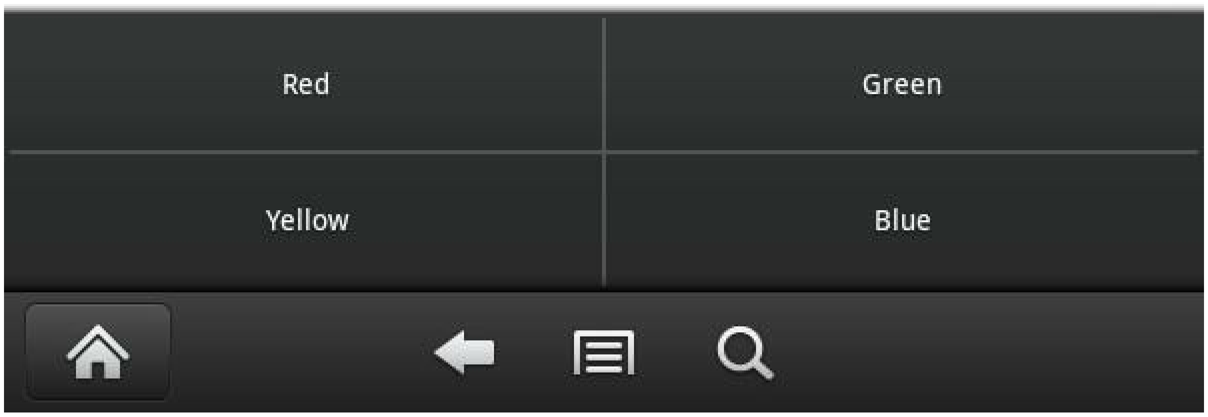


Figure 31-1

With the Android 4.0 release and later, on the other hand, the overflow menu button is located in the top right-hand corner (Figure 31-2) in the action toolbar represented by the stack of three squares:

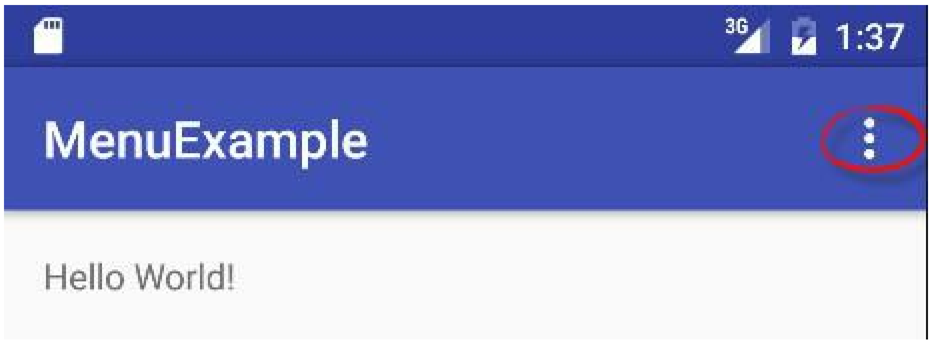


Figure 31-2

31.2 Creating an Overflow Menu

The items in a menu can be declared within an XML file, which is then inflated and displayed to the user on demand. This involves the use of the <menu> element, containing an <item> sub-element for each menu item. The following XML, for example, defines a menu consisting of two menu items relating to color choices:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=
           ".MenuExampleActivity" >
    <item
        android:id="@+id/menu_red"
        android:orderInCategory="1"
        app:showAsAction="never"
        android:title="@string/red_string"/>
       <item
        android:id="@+id/menu_green"
        android:orderInCategory="2"
        app:showAsAction="never"
        android:title="@string/green_string"/>
</menu>
```

In the above XML, the *android:orderInCategory* property dictates the order in which the menu items will appear within the menu when it is displayed. The *app:showAsAction* property, on the other hand, controls the conditions under which the corresponding item appears as an item within the action bar itself. If set to *ifRoom*, for example, the item will appear in the action bar if there is enough room. Figure 31-3 shows the effect of setting this property to *ifRoom* for both menu items:

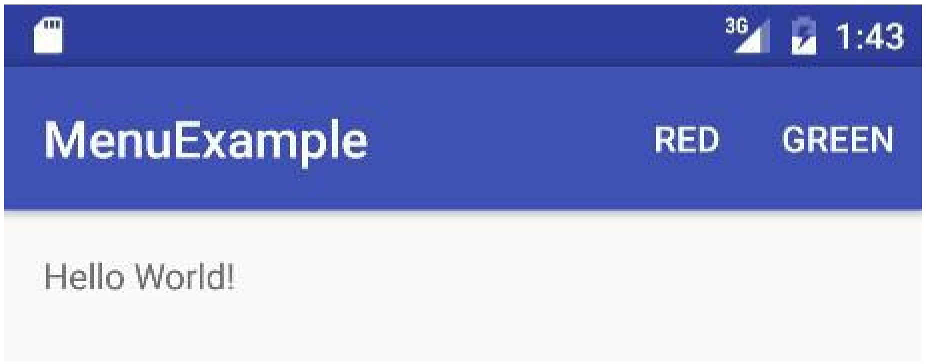


Figure 31-3

This property should be used sparingly to avoid over cluttering the action bar.

By default, a menu XML file is created by Android Studio when a new Android application project is created. This file is located in the *app -> res -> menu* project folder and contains a single menu item entitled “Settings”:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context=".MainActivity">
       <item android:id="@+id/action_settings"
            android:title="@string/action_settings"
            android:orderInCategory="100"
            app:showAsAction="never" />
</menu>
```

This menu is already configured to be displayed when the user selects the overflow menu on the user interface when the app is running, so simply modify this one to meet your needs.

31.3 **Displaying an Overflow Menu**

An overflow menu is created by overriding the *onCreateOptionsMenu()* method of the corresponding activity and then inflating the menu's XML file. For example, the following code creates the menu contained within a menu XML file named *menu_menu_example*:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
       getMenuInflater().inflate(R.menu.menu_menu_example, menu);
       return true;
}
```

As with the menu XML file, Android Studio will already have overridden this method in the main activity of a newly created Android application project. In the event that an overflow menu is not required in your activity, either remove or comment out this method.

31.4 **Responding to Menu Item Selections**

Once a menu has been implemented, the question arises as to how the application receives notification when the user makes menu item selections. All that an activity needs to do to receive menu selection notifications is to override the *onOptionsItemSelected()* method. Passed as an argument to this method is a reference to the selected menu item. The *getItemId()* method may then be called on the item to obtain the ID which may, in turn, be used to identify which item was selected. For example:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
       switch (item.getItemId()) {
        case R.id.menu_red:
            // Red item was selected
            return true;
        case R.id.menu_green:
            // Green item was selected
            return true;
        default:
            return super.onOptionsItemSelected(item);
       }
}
```

31.5 **Creating Checkable Item Groups**

In addition to configuring independent menu items, it is also possible to create groups of menu items. This is of particular use when creating checkable menu items whereby only one out of a number of choices can be selected at any one time. Menu items can be assigned to a group by wrapping them in the *<group>* tag. The group is declared as checkable using the *android:checkableBehavior* property, setting the value to either *single, all or none*. The following XML declares that two menu items make up a group wherein only one item may be selected at any given time:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/menu_red"
            android:title="@string/red_string"/>
        <item
```

```
                android:id="@+id/menu_green"
                android:title="@string/green_string"/>
        </group>
    </menu>
```

When a menu group is configured to be checkable, a small circle appears next to the item in the menu as illustrated in Figure 31-4. It is important to be aware that the setting and unsetting of this indicator does not take place automatically. It is, therefore, the responsibility of the application to check and uncheck the menu item.

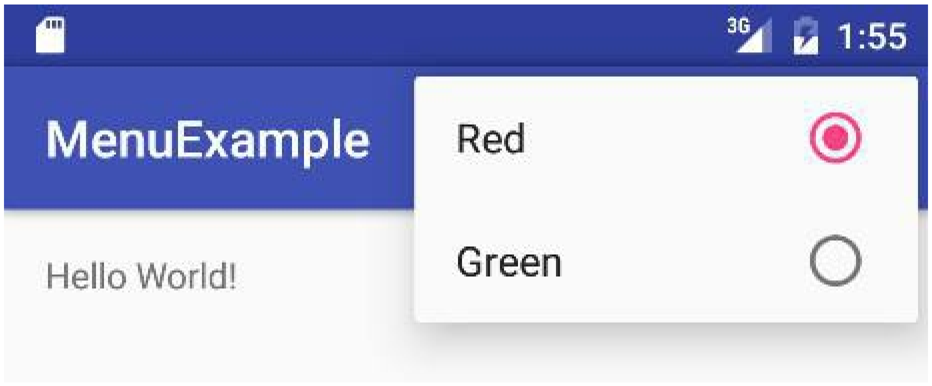


Figure 31-4

Continuing the color example used previously in this chapter, this would be implemented as follows:

```
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
            switch (item.getItemId()) {
                case R.id.menu_red:
                    if (item.isChecked()) item.setChecked(false);
                    else item.setChecked(true);
                    return true;
                case R.id.menu_green:
                    if (item.isChecked()) item.setChecked(false);
                    else item.setChecked(true);
                    return true;
                default:
                    return super.onOptionsItemSelected(item);
            }
    }
```

31.6 Menus and the Android Studio Menu Editor

Prior to the introduction of Android Studio 2.2, the only way to construct a menu was to manually edit the XML content of the menu resource file as outlined above. Android Studio now allows menus to be designed visually simply by loading the menu resource file into the Menu Editor tool, dragging and dropping menu elements from a palette and setting properties. This considerably eases the menu design process, though it is important to be aware that it is still necessary to write the code in the *onOptionsItemSelected()* method to implement the menu behavior.

To visually design a menu, locate the menu resource file and double-click on it to load it into the Menu Editor tool. Figure 31-5, for example, shows the default menu resource file for a basic activity loaded into the Menu Editor:

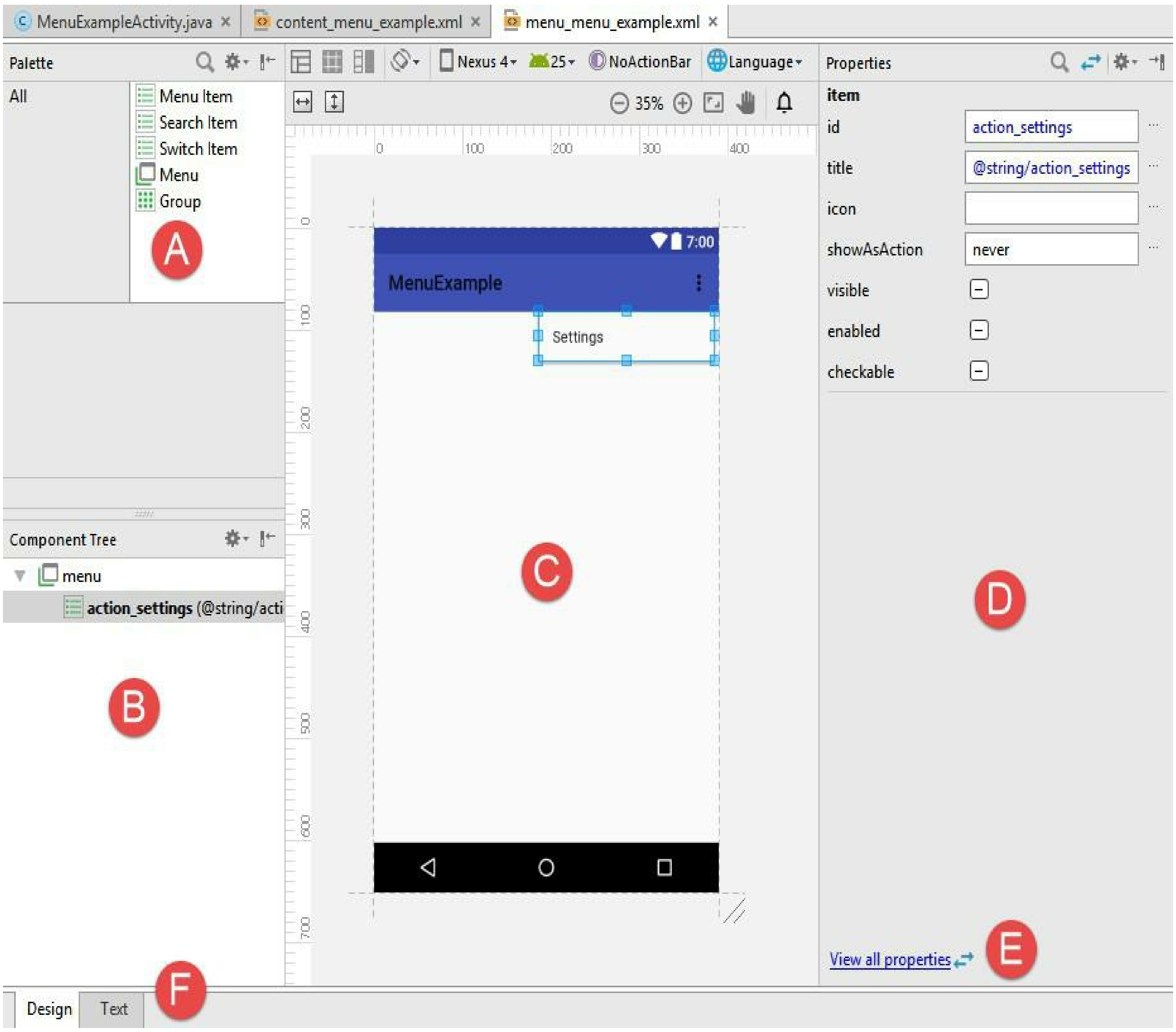


Figure 31-5

The palette (A) contains items that can be added to the menu contained in the design area (C). The Component Tree (B) is a useful tool for identifying the hierarchical structure of the menu. The Properties panel (D) contains a subset of common properties for the currently selected item. The view all properties link (E) may be used to access the full list of properties.

New elements may be added to the menu by dragging and dropping objects either onto the layout canvas or the Component Tree. When working with menus in the Layout Editor tool, it will sometimes be easier to drop the items onto the Component Tree since this provides greater control over where the item is placed within the tree. This is of particular use, for example, when adding items to a group.

Although the Menu Editor provides a visual approach to constructing menus, the underlying menu is still stored in XML format which may be viewed and edited manually by switching from Design to Text mode using the tab marked F in the above figure.

31.7 **Creating the Example Project**

To see the overflow menu in action, create a new project in Android Studio, entering *MenuExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of a basic activity named *MenuExampleActivity* with corresponding layout and menu resource files named *activity_menu_example* and *menu_menu_example.*

When the project has been created, navigate to the *app -> res -> layout* folder in the Project tool window and double-click on the *content_menu_example.xml* file to load it into the Android Studio Menu Editor tool. Switch the tool to Design mode, select the ConstraintLayout from the Component Tree panel and enter *layoutView* into the ID field of the Properties panel.

31.8 **Designing the Menu**

Within the Project tool window, locate the project's *app -> res -> menu -> menu_menu_example.xml* file and double-click on it to load it into the Layout Editor tool. Select and delete the default Settings menu item added by Android Studio so that the menu currently has no items.

From the palette, click and drag a menu *group* object onto the title bar of the layout canvas as highlighted in Figure 31-6:

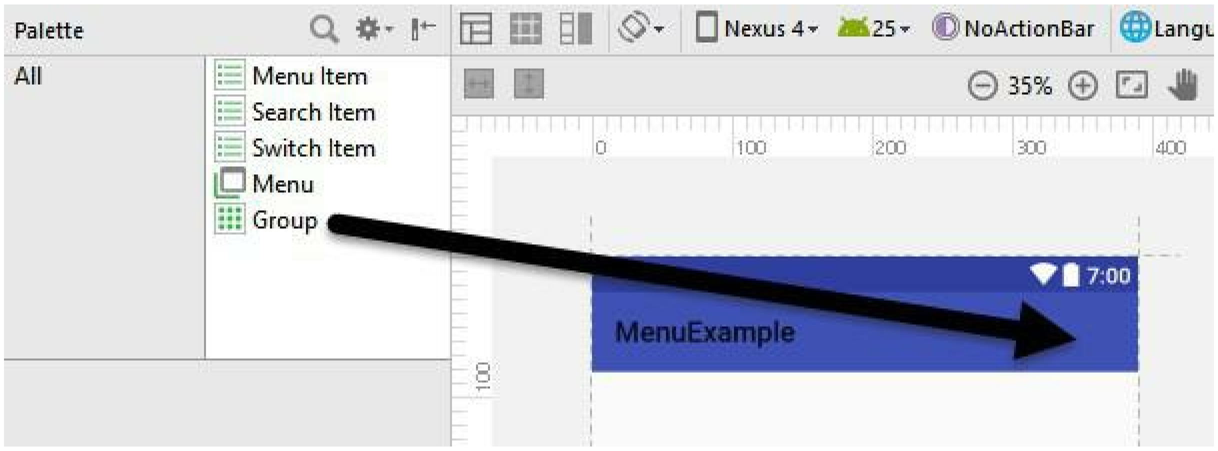


Figure 31-6

Although the group item has been added, it will be invisible within the layout. To verify the presence of the element, refer to the Component Tree panel where the group will be listed as a child of the menu:

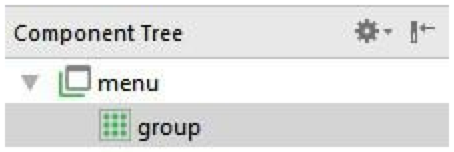
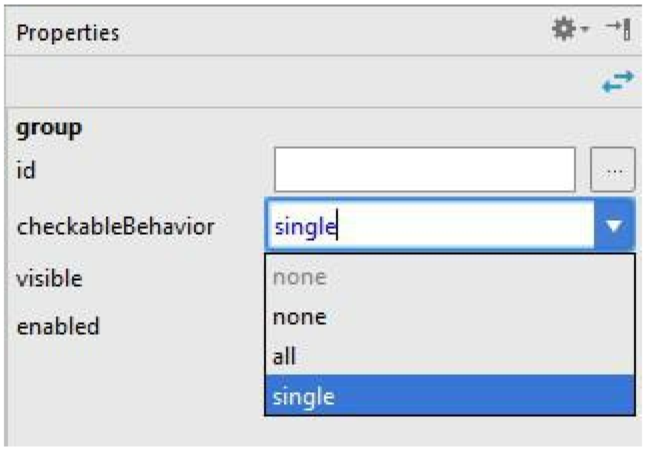


Figure 31-7

Select the *group* entry in the Component Tree and, referring to the Properties panel, set the *checkableBehavior* property to *single* so that only one group menu item can be selected at any one time:


Properties

group

id

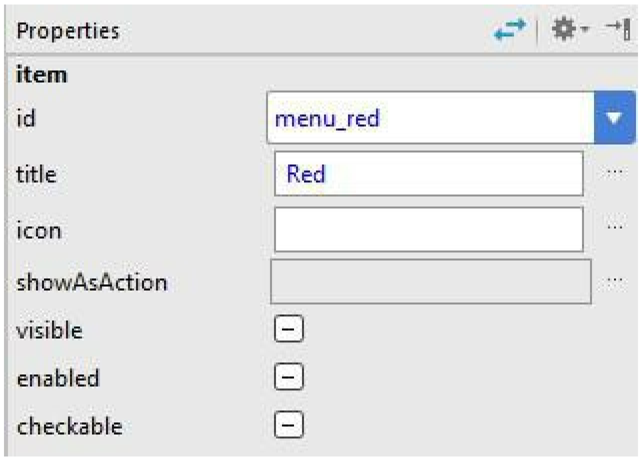
checkableBehavior single

visible

enabled

Figure 31-8

Next, drag and drop four *item* elements from the palette and drop them onto the *group* element in the Component Tree. Select the first item and use the Properties panel to change the title to "Red" and the ID to *menu_red*:


Properties

item

id menu_red

title Red

icon

showAsAction

visible

enabled

checkable

Figure 31-9

Repeat these steps for the remaining three menu items setting the titles to "Green", "Yellow" and "Blue" with matching IDs of *menu_green*, *menu_yellow* and *menu_blue*. Use the red warning button in the top right-hand corner of editor panel to extract the strings to resources.

On completion of these steps, the menu layout should match that shown in Figure 31-10 below:

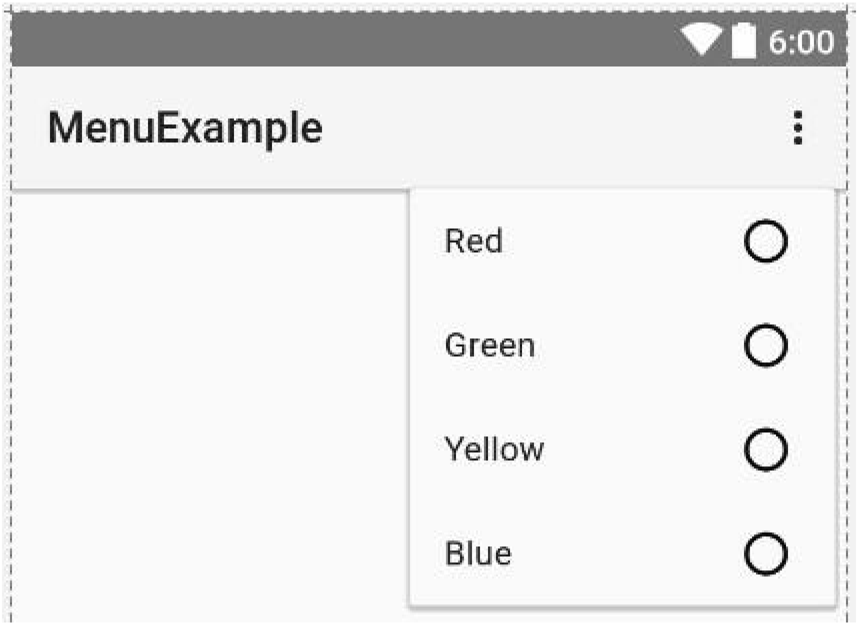


Figure 31-10

Switch the Layout Editor tool to Text mode and review the XML representation of the menu which should match the following listing:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.ebookfrenzy.menuexample.MenuExampleActivity">

    <group android:checkableBehavior="single">
        <item android:title="@string/red_string"
            android:id="@+id/menu_red" />
        <item android:title="@string/green_string"
            android:id="@+id/menu_green" />
        <item android:title="@string/yellow_string"
            android:id="@+id/menu_yellow" />
        <item android:title="@string/blue_string"
            android:id="@+id/menu_blue" />
    </group>
</menu>
```

31.9 **Modifying the *onOptionsItemSelected()* Method**

When items are selected from the menu, the overridden *onOptionsItemsSelected()* method of the application's activity will be called. The role of this method will be to identify which item was selected and change the background color of the layout view to the corresponding color. Locate and double-click on the *app -> java -> com.ebookfrenzy.menuexample -> MenuExampleActivity* file and modify the method as follows:

```
package com.ebookfrenzy.menuexample;

import android.os.Bundle;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.View;
import android.view.Menu;
import android.view.MenuItem;
import android.support.constraint.ConstraintLayout;

public class MenuExampleActivity extends AppCompatActivity {
.
.
.
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {

        ConstraintLayout mainLayout =
                (ConstraintLayout) findViewById(R.id.layoutView);

        switch (item.getItemId()) {
            case R.id.menu_red:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.RED);
                return true;
            case R.id.menu_green:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.GREEN);
                return true;
            case R.id.menu_yellow:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);

mainLayout.setBackgroundColor(android.graphics.Color.YELLOW);
                return true;
            case R.id.menu_blue:
                if (item.isChecked()) item.setChecked(false);
                else item.setChecked(true);
                mainLayout.setBackgroundColor(android.graphics.Color.BLUE);
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }
.
.
}
```

31.10 Testing the Application

Build and run the application on either an emulator or physical Android device. Using the overflow menu, select menu items and verify that the layout background color changes appropriately. Note that the currently selected color is displayed as the checked item in the menu.

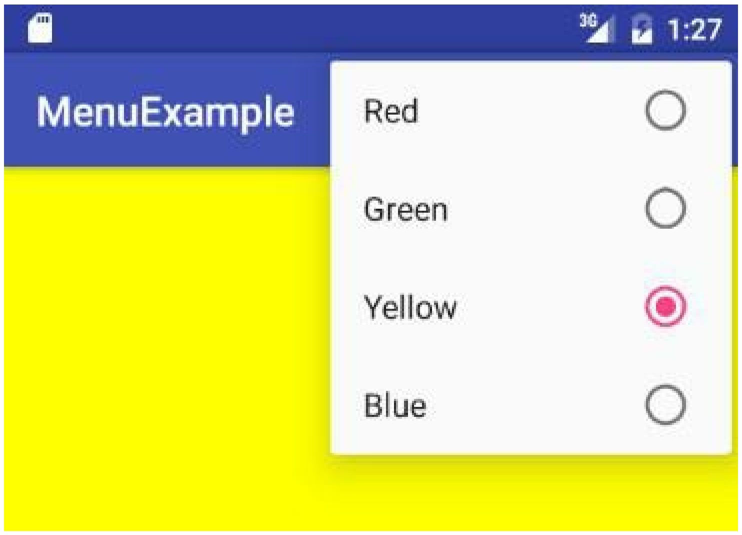


Figure 31-11

31.11 Summary

On earlier versions of Android, the overflow menu is accessible from the soft key toolbar at the bottom of the screen. On Android 4.0 and later, the menu is accessed from the far right of the actions toolbar at the top of the display. This menu provides a location for applications to provide additional options to the user.

The structure of the menu is most easily defined within an XML file and the application activity receives notifications of menu item selections by overriding and implementing the *onOptionsItemSelected()* method.