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 - An 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/and
    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 star

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);
       }
}
```

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/and
    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.Fragme
        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:

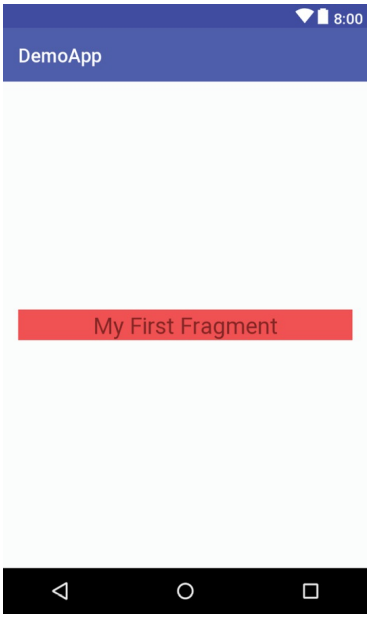

Figure 29-1

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.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. 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 fragments 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.