**Chapter 5 Fragments**

In this chapter, we will cover the following topics:

1. Creating and using a Fragment
2. Adding and removing Fragments during runtime
3. Passing data between Fragments
4. Handling the Fragment back stack

**Introduction**

With a firm understanding of layouts from Chapter 2, Layouts, we'll dig deeper into UI development with Fragments. **Fragments are a way to separate your UI into smaller sections that can easily be reused**.

Think of Fragments as **mini-activities**, **complete with their own classes, layouts, and life cycle**. Instead of designing your screen in one Activity Layout, possibly duplicating functionality across multiple layouts, **you can break the screen into smaller, logical sections and turn them into Fragments**. Your Activity Layout can then **reference one or multiple Fragments**, as needed.

**1 Creating and using a Fragment**

Android didn't always support Fragments. The early versions of Android were designed for phones when screens had relatively small displays.

It wasn't until Android started being used on tablets that there was a need **to split the screen into smaller sections**. Android 3.0 introduced the Fragments class and the **Fragment Manager**.

Along with a new class, also came the Fragment Lifecycle. The **Fragment Lifecycle is similar to the Activity Lifecycle** introduced in Chapter 1, Activities, as most events parallel the Activity Lifecycle.

Here's a brief overview of the **main callbacks**:

* **onAttach**(): It's called when the Fragment is associated with an Activity.
* **onCreate**(): It's called when the Fragment is first created.
* **onCreateView**(): It's called when the Fragment is about to be displayed for the first time.
* **onActivityCreated**(): It's called when the associated Activity is created.
* **onStart**(): It's called when the Fragment will become visible to the user.
* **onResume**(): It's called just before a Fragment is displayed.
* **onPause**(): It's called when the Fragment is first suspended. **The user may return to the Fragment, but this is where you should persist any user data**.
* **onStop**(): It's called when the Fragment is no longer visible to the user.
* **onDestroyView**(): It's called to allow final cleanup.
* **onDetach**(): It's called when the Fragment is no longer associated with the Activity.

For our first exercise**, we will create a new Fragment derived from the standard Fragment class**. But there are several **other Fragment classes we could derive from, including the following**:

* **DialogFragment**: It's used for creating a **floating dialog.**
* **ListFragment**: It **creates a ListView in a Fragment**, similar to ListActivity.
* **PreferenceFragment**: It creates a **list of Preference objects, commonly used for a Settings page.**

In this recipe, we will walk through creating a basic Fragment derived from the Fragment class **and include it in an Activity Layout**.

**Getting ready**

Create a new project in Android Studio and call it **CreateFragment**. Use the default Phone & Tablet option and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it…**

In this recipe, we will create a new Fragment class **with an accompanying layout file**. We will then **add the Fragment to the Activity Layout so it will be visible when the activity starts**.

Here are the **steps** to create and display a new Fragment:

1. **Create a new layout called fragment\_one.xml** using the following XML:

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

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

android:layout\_height="match\_parent"

android:layout\_width="match\_parent">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Fragment One"

android:id="@+id/textView"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

</RelativeLayout>

2. Create a **new Java class called FragmentOne.java** with the following code:

public class FragmentOne extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater, ViewGroup container,

Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_one, container, false);

}

}

3. Open the **activity\_main.xml file and replace the existing <TextView> element with the following <fragment> element**:

<fragment

android:name="com.packtpub.createfragment.FragmentOne"

android:id="@+id/fragment"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

4. **Run the program** on a device or emulator.

**How it works...**

We start by **creating a new class, the same as we do for an Activity**. In this recipe, we only create an **overwrite for the onCreateView() method to load our Fragment layout**. But, just like with the Activity events, we can override the other events as we need them.

Once the new Fragment is created, **we then add it to the Activity Layout**. Since the original Activity class was created before Fragments existed, **they do not support Fragments**.

**That's why, unless otherwise indicated, all the examples for this book extend from AppCompatActivity**. (If you used the Android Studio New Project Wizard, then by default **MainActivity extends AppCompatActivity**.)

**There's more...**

We're only creating a single, simple Fragment in this recipe to teach the fundamentals of Fragments. But this is a good time to point out the power of Fragments.

If we are creating **multiple Fragments** (and usually we are, as that's the point of using Fragments), when creating the Activity Layouts as we did in step 4, **we could create different layout configurations using the Android Resource Folders**.

The **portrait layout may have only a single Fragment** while the **landscape may have two or more.** The **Master/Detail layout typically uses Fragments**, thus **only requiring each screen section to be designed and coded once**, then included in the layout as appropriate.

**See also**

For more information on the Master/Detail pattern, **see the Passing data between Fragments recipe** later in this chapter.

**2 Adding and removing Fragments during runtime**

Defining a Fragment in the layout, as we did in the previous recipe, is **known as a static Fragment**, which **doesn't allow the fragment to be changed during runtime**.

Rather than using the <fragment> element, **we will create a container to hold the Fragment, then create the Fragment dynamically in the Activity's onCreate() method**.

The **FragmentManager provides the APIs for adding, removing, and changing Fragments during runtime using a FragmentTransaction**. A Fragment transaction consists of the following:

1. **Starting a transaction**

2. **Performing one or multiple actions**

3. **Committing the transaction**

This recipe will demonstrate the Fragment Manager **by adding and removing Fragments during runtime**.

**Getting ready**

Create a new project in Android Studio and call it: **RuntimeFragments**. Use the default Phone & Tablet option and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it…**

To demonstrate adding and removing Fragments, **we first need to create the Fragments**, which we will do by **extending the Fragment** **class**.

After creating the new Fragments, **we need to alter the layout for the Main Activity to include the Fragment container**. From there, we just **add the code to handle the Fragment transactions**. Here are the steps:

1. Create a **new layout file called fragment\_one.xml** and include the following XML:

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

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

android:layout\_height="match\_parent"

android:layout\_width="match\_parent">

<TextView

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Fragment One"

android:id="@+id/textView"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

</RelativeLayout>

2. The **second layout file called fragment\_two.xml** is almost identical, with the only difference being the text:

**android:text="Fragment Two"**

3. Create a new Java class called **FragmentOne.java** with the following code:

public class FragmentOne extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater,

ViewGroup container, Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_one,

container, false);

}

}

**Import from the support library as follows**:

**import android.support.v4.app.Fragment;**

4. Create the second Java class called **FragmentTwo** with the following code:

public class FragmentTwo extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater,

ViewGroup container, Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_two,

container, false);

}

}

**As before, import from the support library:**

import android.support.v4.app.Fragment;

5. Now we need to **add a container and a button to the Main Activity layout. Change activity\_main.xml** 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">

<FrameLayout

android:id="@+id/frameLayout"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_above="@+id/buttonSwitch"

android:layout\_alignParentTop="true">

</FrameLayout>

<Button

android:id="@+id/buttonSwitch"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Switch"

android:layout\_alignParentBottom="true"

android:layout\_centerInParent="true"

android:onClick="switchFragment"/>

</RelativeLayout>

6. With the Fragments created **and the container added to the layout**, we are now ready **to write the code to manipulate the Fragments**. Open **MainActivity.java** and add the following code below the class constructor:

FragmentOne mFragmentOne;

FragmentTwo mFragmentTwo;

int showingFragment=0;

7. **Add the following code to the existing onCreate()** method, **below setContentView()**:

mFragmentOne = new FragmentOne();

mFragmentTwo = new FragmentTwo();

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction =

fragmentManager.beginTransaction();

fragmentTransaction.add(R.id.frameLayout, mFragmentOne);

fragmentTransaction.commit();

showingFragment=1;

**Import from the support libraries:**

**import android.support.v4.app.FragmentManager;**

**import android.support.v4.app.FragmentTransaction;**

8. The last code we need to add **handles the Fragment switching, called by the button**:

public void switchFragment(View view) {

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();

if (showingFragment==1) {

fragmentTransaction.replace(R.id.frameLayout, mFragmentTwo);

showingFragment = 2;

} else {

fragmentTransaction.replace(R.id.frameLayout, mFragmentOne);

showingFragment=1;

}

fragmentTransaction.commit();

}

9. **Run the program** on a device or emulator.

**How it works...**

Most of the steps for this recipe involve setting up the Fragments. Once the Fragments are declared, **we create them in the onCreate()** method. Though the code can be condensed to a single line, it's shown in the long form as it makes it easier to read and understand.

First, **we get FragmentManager so we can begin FragmentTransaction**. Once we have FragmentTransaction, **we start the transaction with beginTransaction**(). Multiple actions can occur within the transaction, but all we need here is **to add() our initial Fragment. We call the commit() method to finalize the transaction**.

Now that you understand the Fragment transaction, here is the **succinct version for onCreate():**

**getSupportFragmentManager().beginTransaction().add(R.id.frameLayout, mFragmentOne).commit();**

Our **switchFragment()** method does basically the same type of Fragment transaction**. Instead of calling the add() method, we call the replace() method with the existing Fragment**.

We **keep track of the current Fragment with the showingFragment variable** so we **know which Fragment to show next**. We are not limited to switching between two Fragments either. **If we needed additional Fragments, we just need to create them.**

**There's more...**

In the Switching between activities recipe from Chapter 1, Activities, we discussed the back stack. Most users would expect the back key to move backward through the "screens" and they don't know or care if those screens are activities or Fragments. Fortunately, **Android makes it very easy to add Fragments to the back stack just by adding a call to addToBackStack() before calling commit()**.

When a Fragment is removed or replaced **without adding it to the back stack, it is immediately destroyed**. If it is added to the back stack, it is **stopped** and, if the user returns to the Fragment, it is **restarted, instead of recreated**.

**See also**

For more information on managing the Fragment back stack, see the Handling the Fragment back stack recipe later in this chapter.

**3 Passing data between Fragments**

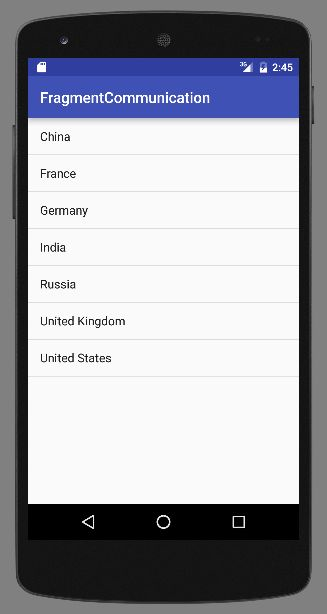
Often, the need arises **to pass information between Fragments**. An email application serves as a classic example. It's common to have **the list of emails in one Fragment** and **show the email details in another Fragment (this is commonly referred to as a Master/Detail pattern)**.

Fragments make creating this pattern easier **because we only have to code each Fragment once**, **then include them in different layouts**.

We can easily **have a single Fragment in a portrait layout with the ability to swap out the master Fragment with the detail Fragment when an email is selected**.

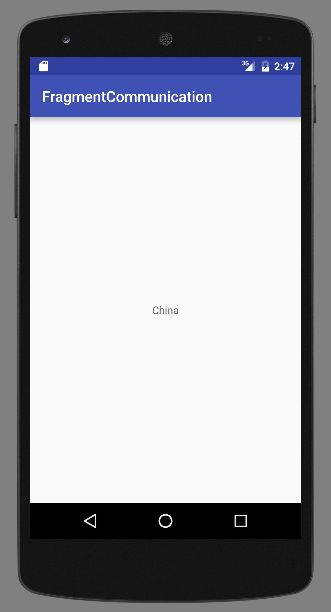
We can also **create a two-panel layout where both the list and detail Fragments are side by side**. Either way, **when the user clicks the email in the list, the email opens up in the detail panel**. This is when we need to communicate between two Fragments.

Since one of the primary goals of Fragments is that they be completely self-contained, **direct communication between Fragments is discouraged**, and for good reason.

If Fragments had to rely on other Fragments, your code would likely break when the layouts changed and only one Fragment was available.

Fortunately, direct communication is not required for this scenario either. **All Fragment communication should pass through the host activity**. The host activity **is responsible for managing the Fragments and can properly route the messages**.

Now the question becomes: **How do Fragments communicate with the activity?** **The answer is with an interface**. You're probably already familiar with an interface, **as that's how a view communicates an event back to an activity**. One of the most common examples is the button **onClick() interface**.



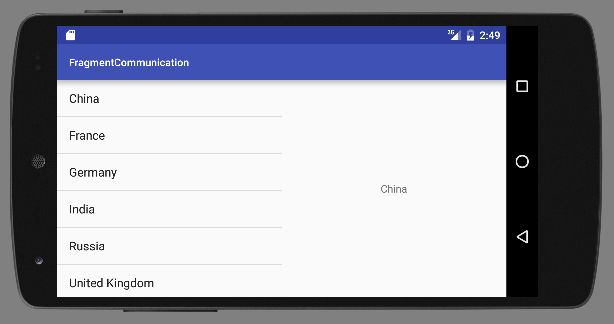
In this recipe, **we will create two Fragments to demonstrate passing data from one Fragment to another via the host activity**.

We'll also build on what we learned from the previous recipe by **including two different Activity Layouts**-one for **portrait** and one for **landscape**.

When in portrait mode, **the activity will swap the Fragments as needed**. Here is a screenshot of when the application first runs in **portrait** mode:

This is the screen showing the **detail Fragment when you click on a country name**:

When in **landscape**, both Fragments **will be side by side, as shown** in the landscape screenshot:



Since the Master/Detail pattern **generally involves a list for the master, we'll take advantage of ListFragment** (mentioned in the Creating and using a Fragment section). When an **item in the list is selected, the item text** (country name in our example) will be **sent to the detail Fragment via the host activity**.

**Getting ready**

Create a new project in Android Studio and call it **FragmentCommunication**. Use the default Phone & Tablet option and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

To fully demonstrate working Fragments, **we'll need to create two Fragments**. **The first Fragment will extend from ListFragment** so it **will not need a layout**.

We're going to go one step further **by creating both portrait and landscape layouts for our Activity**.

For portrait mode, we'll swap Fragments and for landscape mode, we'll show both Fragments side by side.

When typing this code, **Android Studio will offer two different library import options**. Since the New Project Wizard automatically **references the AppCompat library, we need to use the support library APIs instead of the framework APIs**. Though very similar, the following code **uses the support Fragment APIs**.

Here are the steps, **starting with the first Fragment**:

1. Create a new Java class called **MasterFragment** and change it so it extends **ListFragment** as shown:

public class MasterFragment extends ListFragment

Import from the following library:

**android.support.v4.app.ListFragment**

2. Create the **following interface inside the MasterFragment** class:

public interface OnMasterSelectedListener {

public void onItemSelected(String countryName);

}

3. **Set up the interface callback listener** with the following code:

private OnMasterSelectedListener mOnMasterSelectedListener=null;

public void setOnMasterSelectedListener(OnMasterSelectedListener listener) {

mOnMasterSelectedListener=listener;

}

4. The last step for the MasterFragment **is to create ListAdapter to populate ListView**, which we do in the **onViewCreated() method**. When a country name is selected, **we'll use setOnItemClickListener() to call our OnMasterSelectedListener interface** with the following code:

public void onViewCreated(View view, Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

String[] countries = new String[]{"China", "France",

"Germany", "India", "Russia", "United Kingdom",

"United States"};

ListAdapter countryAdapter = new ArrayAdapter<String>(

getActivity(), android.R.layout.simple\_list\_item\_1,

countries);

setListAdapter(countryAdapter);

getListView().setChoiceMode(ListView.CHOICE\_MODE\_SINGLE);

getListView().setOnItemClickListener(new AdapterView.OnItemClickListener() {

@Override

public void onItemClick(AdapterView<?> parent, View

view, int position, long id) {

if (mOnMasterSelectedListener != null) {

mOnMasterSelectedListener.onItemSelected(((

TextView) view).getText().toString());

}

}

});

}

5. Next, we need to **create DetailFragment**, starting with the layout. Create a **new layout file called fragment\_detail.xml** with the following XML:

<?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/textViewCountryName"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

</RelativeLayout>

6. Create a **new Java class called DetailFragment extending from Fragment** as follows:

public class DetailFragment extends Fragment

Import from the following library:

**android.support.v4.app.Fragment**

7. Add the **following constant** to the class:

**public static String KEY\_COUNTRY\_NAME="KEY\_COUNTRY\_NAME";**

8. **Override onCreateView()** as follows:

@Override

public View onCreateView(LayoutInflater inflater,

ViewGroup container,

Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_detail, container, false);

}

9. Code **onViewCreated()** as follows:

@Override

public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {

super.onViewCreated(view, savedInstanceState);

Bundle bundle = getArguments();

if (bundle != null && bundle.containsKey(KEY\_COUNTRY\_NAME)) {

showSelectedCountry(bundle.getString(KEY\_COUNTRY\_NAME));

}

}

10. The last step for this Fragment **is to update TextView when we receive the selected country name**. Add the following method to the class:

public void showSelectedCountry(String countryName) {

((TextView)getView().findViewById(R.id.textViewCountryName)).setText(countryName);

}

11. The existing **activity\_main.xml layout will handle the portrait mode layout**. Remove the existing <TextView> and replace with the **following <FrameLayout>:**

<FrameLayout

android:id="@+id/frameLayout"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:layout\_marginTop="8dp"

app:layout\_constraintBottom\_toBottomOf="parent"

app:layout\_constraintLeft\_toLeftOf="parent"

app:layout\_constraintRight\_toRightOf="parent"

app:layout\_constraintTop\_toTopOf="parent" />

12. For the **landscape layout, create a new directory called layout-land in the res folder**. The final result will be res/layout-land. If you do not see the new res/layout-land directory, **change from Android view to project view**.

13. Create a **new activity\_main.xml layout** in res/layout-land as follows:

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

<LinearLayout 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"

android:orientation="horizontal">

<FrameLayout

android:id="@+id/frameLayoutMaster"

android:layout\_width="0dp"

android:layout\_weight="1"

android:layout\_height="match\_parent"/>

<FrameLayout

android:id="@+id/frameLayoutDetail"

android:layout\_width="0dp"

android:layout\_weight="1"

android:layout\_height="match\_parent"/>

</LinearLayout>

14. The final steps are **to set up MainActivity to handle the Fragments**. Open the MainActivity.java file and **add the following class variable to track single/dual pane**:

**boolean mDualPane;**

15. Next, **change onCreate()** as follows:

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

MasterFragment masterFragment = null;

FrameLayout frameLayout = findViewById(R.id.frameLayout);

if (frameLayout != null) {

mDualPane = false;

FragmentTransaction fragmentTransaction = getSupportFragmentManager().beginTransaction();

masterFragment = (MasterFragment) getSupportFragmentManager()

.findFragmentByTag("MASTER");

if (masterFragment == null) {

masterFragment = new MasterFragment();

fragmentTransaction.add(R.id.frameLayout, masterFragment, "MASTER");

}

DetailFragment detailFragment = (DetailFragment)

getSupportFragmentManager().findFragmentById(R.id.frameLayoutDetail);

if (detailFragment != null) {

fragmentTransaction.remove(detailFragment);

}

fragmentTransaction.commit();

} else {

mDualPane = true;

FragmentTransaction fragmentTransaction = getSupportFragmentManager().beginTransaction();

masterFragment = (MasterFragment) getSupportFragmentManager()

.findFragmentById(R.id.frameLayoutMaster);

if (masterFragment == null) {

masterFragment = new MasterFragment();

fragmentTransaction.add(R.id.frameLayoutMaster, masterFragment);

}

DetailFragment detailFragment = (DetailFragment) getSupportFragmentManager()

.findFragmentById(R.id.frameLayoutDetail);

if (detailFragment == null) {

detailFragment = new DetailFragment();

fragmentTransaction.add(R.id.frameLayoutDetail, detailFragment);

}

fragmentTransaction.commit();

}

masterFragment.setOnMasterSelectedListener(new MasterFragment.OnMasterSelectedListener() {

@Override

public void onItemSelected(String countryName) {

sendCountryName(countryName);

}

});

}

16. The last code to **add is the sendCountryName() method**, which **handles sending the country name to DetailFragment**:

private void sendCountryName(String countryName) {

DetailFragment detailFragment;

if (mDualPane) {

//Two pane layout

detailFragment = (DetailFragment) getSupportFragmentManager().findFragmentById(R.id.frameLayoutDetail);

detailFragment.showSelectedCountry(countryName);

} else {

// Single pane layout

detailFragment = new DetailFragment();

Bundle bundle = new Bundle();

bundle.putString(DetailFragment.KEY\_COUNTRY\_NAME, countryName);

detailFragment.setArguments(bundle);

FragmentTransaction fragmentTransaction = getSupportFragmentManager().beginTransaction();

fragmentTransaction.replace(R.id.frameLayout, detailFragment);

fragmentTransaction.addToBackStack(null);

fragmentTransaction.commit();

}

}

17. **Run the program** on a device or emulator.

**How it works...**

We start by creating **MasterFragment**. In the Master/Detail pattern we are using, **this usually represents a list**, so we create a list by extending ListFragment. **ListFragment is the Fragment equivalent of ListActivity**. Other than extending from a Fragment, it's basically the same.

As stated in the recipe introduction, we shouldn't attempt to communicate directly with other Fragments. To provide a means **to communicate the list item selection, we expose the interface: OnMasterSelectedListener. We call onItemSelected()** every time an item is selected in the list.

Most of the work for passing data between Fragments **is done in the host activity but, ultimately, the receiving Fragment needs a way to receive the data**.

**DetailFragment supports this** in two ways:

* **Passing the country name in the argument bundle**, **available at creation time**
* **A public method for the activity to call directly**.

When the **activity creates the Fragment, it also creates a bundle to hold the data we want to send**. Here **we add the country name using KEY\_COUNTRY\_NAME defined in step 7**. **We retrieve this bundle with getArguments() in onViewCreated()**.

If the key is found in the bundle, **it is extracted and displayed using the showSelectedCountry() method**. This is **the same method the activity will call directly if the Fragment is already visible (in the two-panel layout)**.

Most of the work for this recipe is in the activity. We created two layouts: one for portrait and one for landscape. When in **landscape orientation, Android will choose the landscape layout from the res/layout-land directory** created in step 12. Both layouts use a **<FrameLayout> placeholder**, similar to the previous exercise. We **manage the Fragments in both onCreate() and sendCountryName()**.

In onCreate(), **we set the mDualPane flag** **by checking whether the current layout includes the frameLayout view**. If frameLayout is found (meaning it's not null), **then we have only a single panel because frameLayout is only defined in the portrait layout**. If frameLayout is **not found, then we have two <FrameLayout>** elements instead: one for **MasterFragment** and another for **DetailFragment**.

The last thing we do in onCreate() is to **set up the MasterFragment listener by creating an anonymous callback**, which **passes the country name to the sendCountryName() method**.

The **sendCountryName() method is where the data is actually passed to DetailFragment**. If we are in portrait (or single-pane) mode, **we need to create DetailFragment and replace the existing MasterFragment**.

This is where **we create the bundle with the country name and call setArguments()**. Notice how **we call addToBackStack() before committing the transaction?** This allows **the back key to bring the user back to the list (MasterFragment)**.

If we are in landscape mode, **DetailFragment is already visible so we call the howSelectedCountry() public method directly**.

**There's more...**

**In MasterFragment, before sending the onItemSelected() event**, **we check to make sure the listener is not null with the following code:**

**if (mOnMasterSelectedListener != null)**

Though it's the job of the activity to set up the callback to receive the events, **we don't want this code to crash if there's no listener**. An alternative approach would be **to verify the activity extends our interface in the Fragment's onAttach() callback.**

The objective for this recipe was to demonstrate the **proper pattern for communicating between fragments (by using an interface) and how to pass data**. We used the ListView fragment because it made typing this example easier, **but for real-world applications, it's probably better to use RecyclerView**.

**RecyclerView does not have a pre-made Fragment class** (or Activity class) so you need **to roll your own but it's no different than the examples shown in earlier chapters**.

**See also**

For RecyclerView examples, **refer to the RecyclerView replaces ListView section in Chapter 2**, Layouts and the **Using Contextual Batch Mode with RecyclerView section in Chapter 4**, Menus and Action Mode.

For more information on resource directories, see the Selecting themes based on the Android version section in Chapter 3, Views, Widgets, and Styles.

**4 Handling the Fragment back stack**

In several of the previous recipes, it was mentioned that you **should call the addToBackStack() method** in the Fragment transaction **to enable Android to maintain a Fragment back stack**.

This is the first step, but may not be enough to provide a rich user experience. **In this recipe, we'll explore two other callbacks: onBackPressed() and onBackStackChanged()**.

As you'll see, by implementing these callbacks, **your application can provide specific behavior for the Fragment back stack**. The **onBackPressed() callback allows the app to check the back stack state and provide custom behavior**, such as **closing the app when appropriate**.

The **onBackStackChanged() callback is called whenever the actual back stack changes** - such as **when a Fragment is popped from the back stack**.

By overriding this callback, **your app can check the current Fragment and update the UI (such as the Home key back arrow) as appropriate**.

**Getting ready**

Create a new project in Android Studio and call it **FragmentBackStack**. Use the default Phone & Tablet option and select Empty Activity on the Add an Activity to Mobile dialog.

**How to do it...**

To demonstrate handling the Fragment back stack, **we'll create two fragments with a Next button to create a back stack**. With that setup, **we'll implement the onBackPressed() callback to exit the app when the user reaches the top Fragment**.

We'll be using the **Fragment Manager from the support library**, so be sure to choose the support library version when prompted for the import library. We'll **need two layout files - one for each fragment - along with two fragment classes**. Here are the steps in detail:

1. Create a new layout file called **fragment\_one.xml** with the following XML:

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

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

android:layout\_height="match\_parent"

android:layout\_width="match\_parent">

<TextView

android:id="@+id/textView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Fragment One"

android:layout\_centerVertical="true"

android:layout\_centerHorizontal="true" />

</RelativeLayout>

2. Create the **second fragment layout file called fragment\_two.xml** with the same XML as above, changing the following text property:

**android:text="Fragment Two"**

3. With the layout files created, it's time to **create the classes for the fragments**. Create a new Java class called **FragmentOne.java** with the following code:

public class FragmentOne extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater,

ViewGroup container, Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_one,

container, false);

}

}

4. Create the second Java class called **FragmentTwo** with the following code:

public class FragmentTwo extends Fragment {

@Override

public View onCreateView(LayoutInflater inflater,

ViewGroup container, Bundle savedInstanceState) {

return inflater.inflate(R.layout.fragment\_two,

container, false);

}

}

5. Now we **need to add a container and a button to the Main Activity** **layout**. Change **activity\_main.xml** 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">

<FrameLayout

android:id="@+id/frameLayout"

android:layout\_width="match\_parent"

android:layout\_height="wrap\_content"

android:layout\_above="@+id/buttonNext"

android:layout\_alignParentTop="true">

</FrameLayout>

<Button

android:id="@+id/buttonNext"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

android:text="Next"

android:layout\_alignParentBottom="true"

android:layout\_centerInParent="true"/>

</RelativeLayout>

6. With the Fragments created and the container added to the layout, we are now ready to **write the code to manipulate the Fragments**. **Open MainActivity.java** and add the following code below the class constructor:

**Button mButtonNext;**

7. Add the following code to the **existing onCreate() method**, **below setContentView()**:

mButtonNext = findViewById(R.id.buttonNext);

mButtonNext.setOnClickListener(new View.OnClickListener() {

@Override

public void onClick(View view) {

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction =

fragmentManager.beginTransaction();

fragmentTransaction.replace(R.id.frameLayout, new FragmentTwo());

fragmentTransaction.addToBackStack(null);

fragmentTransaction.commit();

mButtonNext.setVisibility(View.INVISIBLE);

}

});

FragmentManager fragmentManager = getSupportFragmentManager();

FragmentTransaction fragmentTransaction =

fragmentManager.beginTransaction();

fragmentTransaction.add(R.id.frameLayout, new FragmentOne());

fragmentTransaction.addToBackStack(null);

fragmentTransaction.commit();

8. The last method to **implement is the onBackPressed() callback**:

@Override

public void onBackPressed() {

if(getSupportFragmentManager().getBackStackEntryCount() == 2 ) {

super.onBackPressed();

mButtonNext.setVisibility(View.VISIBLE);

} else {

finish();

}

}

9. **Run the program** on a device or emulator.

**How it works...**

Most of the steps are similar to the Adding and removing Fragments **during runtime recipe discussed previously**, until step 8. The first seven steps just set up the app to create the fragments for our demonstration.

In step 8, **we implement the onBackPressed() callback**. This is where we code for **our specific situation**. For this sample, all we need to do is make the Next button visible again.

**There's more...**

With the basics covered for handling the back stack, it's time to discuss the other callback: **onBackStackChanged()**. This is where **you can implement custom behavior when the stack changes**.

One common example **is changing the Home icon to a back arrow**. We get **this behavior automatically with an Activity when we set the parent property (in AndroidManifest)**, but **Android doesn't do this for fragments**.

What if we wanted to **have a back arrow on FragmentTwo?** **Add this line of code to the NextButton onClick()**:

**getSupportActionBar().setDisplayHomeAsUpEnabled(true);**

If you run the app now, **you'll see the back arrow when you go to FragmentTwo**. The problem is, the back arrow **doesn't actually do anything.** The next problem you may notice **is that if you use the back key, you still see the back arrow when you return to FragmentOne**.

To make the **back arrow work, add the following code to MainActivity:**

@Override

public boolean onOptionsItemSelected(MenuItem menuItem) {

if (menuItem.getItemId() == android.R.id.home) {

onBackPressed();

return true;

} else {

return super.onOptionsItemSelected(menuItem);

}

}

Now the app **will respond to the back arrow and treat it the same as the back key**. What about the second issue? **The Home icon still shows the back arrow**.

This is **where we can use the onBackStackChanged() callback**. Instead of modifying the NextButton onClick() as we did earlier, **we can put all our code in onBackStackChanged().**

To do this, **we need to implement the OnBackStackChangedListener interface in the class definition**. **Change the MainActivity** declaration as follows:

public class MainActivity extends AppCompatActivity

implements FragmentManager.OnBackStackChangedListener {

Then add this line to the onCreate() method (below setContentView()) to add the listener:

getSupportFragmentManager().addOnBackStackChangedListener(this);

Now we can implement the onBackStackChanged() callback:

@Override

public void onBackStackChanged() {

Fragment fragment = getSupportFragmentManager().findFragmentById(R.id.frameLayout);

if (fragment instanceof FragmentOne) {

getSupportActionBar().setDisplayHomeAsUpEnabled(false);

} else if (fragment instanceof FragmentTwo) {

getSupportActionBar().setDisplayHomeAsUpEnabled(true);

}

}

Now when you run the app, **you'll see the back arrow when you go to FragmentTwo**. **You can press the back arrow icon or use the back key to return to the first screen**. Thanks to the onBackStackChanged() callback, **you won't see the back arrow when you're on FragmentOne**.