Creating Dynamic UIs with Android Fragments
Second Edition

This book looks at the impact fragments have on Android UI design and their role in both simplifying many common UI challenges and in providing best practices for incorporating rich UI behaviors. We look closely at the role of fragment transactions and how to work with the Android back stack. Leveraging this understanding, we explore several specialized fragment-related classes such as ListFragment and DialogFragment. We then go on to discuss how to implement rich navigation features such as swipe-based screen browsing, and the role of fragments when developing applications that take advantage of the latest aspects of material design. By the end of this book, we will be able to create a more interactive user experience and dynamically adaptive UIs.

Who this book is written for
This book is for developers who know the basics of Android programming and want to improve the appearance and usability of their applications by providing better support for tablets and smartphones in a single app and reducing the complexity of managing app UIs.

What you will learn from this book
- Learn the role and capabilities of fragments
- Use Android Studio's fragment-oriented features
- Create an app UI that works effectively on smartphones and tablets
- Manage the creation and life cycle of fragments
- Dynamically manage fragments using the FragmentTransaction class
- Learn application design techniques for communicating between fragments
- Leverage fragments when implementing applications that take advantage of the latest features of Material Design

Create engaging apps with fragments to provide a rich user interface that dynamically adapts to the individual characteristics of your customers' tablets and smartphones.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 3 'Fragment Life Cycle and Specialization'
- A synopsis of the book’s content
- More information on *Creating Dynamic UIs with Android Fragments Second Edition*
About the Author

Jim Wilson is the President of JW Hedgehog, Inc., a consulting firm specializing in solutions for the Android, iOS, and Microsoft platforms. He has over 30 years of software engineering experience, with the past 15 years heavily focused on creating mobile device and location-based solutions. Jim cofounded multiple software-related startups and has served in a consulting role at several more. After nearly a decade as a Microsoft Device Application Development MVP, he now focuses on developing Android and iOS device applications.

Jim's passion is mentoring software developers. He is a regular contributor of Android, iOS, and Xamarin training material at Pluralsight (http://training.jwhh.com), a leading provider of online developer training. Jim has authored more than 30 articles on device application development and has served as a contributing expert on mobile software development issues for a variety of media outlets.

Jim and his wife, along with several cats, split their time between Celebration, Florida (just three miles from Walt Disney World) and Weirs Beach, New Hampshire. You can take a look at his blog (http://blog.jwhh.com) where he talks about a variety of mobile software development issues as well as the adventures of a life split between the busy region of the "House of Mouse" and the quietness of NH's lakes and mountains.

You can reach Jim at androidtraining@jwhh.com.
Preface

Long gone are the days of mobile apps with a static UI squished on a tiny screen. Today's users expect mobile apps to be dynamic and highly interactive. They expect an app to look fantastic when they look at it on their medium resolution smartphone and just as fantastic when they switch over to using it on their high-resolution tablet. Apps need to provide rich navigation features, be adaptive, and be responsive.

Trying to meet these demands using Android's traditional activity-centric UI design model is difficult at best. As developers, we need more control than that afforded by activities. We need a new approach, and fragments give us this new approach.

In this book, you'll learn how to use fragments to meet the challenges of creating dynamic UIs in the modern world of mobile app development.

What this book covers

Chapter 1, Fragments and UI Modularization, introduces fragments, UI modularization, and the role that fragments play in developing a modularized UI. This chapter demonstrates creating simple fragments and using fragments statically within activities.

Chapter 2, Fragments and UI Flexibility, builds on the concepts introduced in the previous chapter to provide solutions to specific differences in device layouts. This chapter explains how to use adaptive activity layout definitions to provide support for a wide variety of device form factors with a small set of fragments that are automatically rearranged based on the current device's UI requirements.
Chapter 3, Fragment Life Cycle and Specialization, discusses the relationship of the life cycle of fragments with that of activities and demonstrates the appropriate programming actions at the various points in the life cycle. Leveraging this knowledge, the special purpose fragment classes, ListFragment and DialogFragment, are introduced to demonstrate their behavior and provide a deeper understanding of how their behavior in the activity life cycle differs from that of standard fragments.

Chapter 4, Working with Fragment Transactions, explains how to create multiple app screens within a single activity by dynamically adding and removing fragments using fragment transactions. Topics covered include thread handling, implementing back button behavior, and dynamically adapting multifragment UIs to differences in device characteristics.

Chapter 5, Creating Rich Navigation, brings everything together by building on the previous chapters to show how to use fragments to enhance the user's experience through rich navigation features. This chapter demonstrates how to implement a number of navigation features, including screen browsing with swipe-based paging, direct screen access with drop-down list navigation, and random screen viewing with tabs.

Chapter 6, Fragments and Material Design, introduces the next generation of application development using material design. This chapter demonstrates how to implement fragments that incorporate a rich visual appearance and animated transitions using the latest features of Android's material design capabilities.
This chapter discusses the relationship of the life cycle of fragments with that of activities and demonstrates the appropriate programming actions at various points in the life cycle. The special purpose fragment classes, ListFragment and DialogFragment, are introduced covering their use and how their behavior in the activity life cycle differs from that of standard fragments.

The following topics are covered in this chapter:

- Fragment setup/display event sequence
- Fragment teardown/hide event sequence
- Working with the ListFragment class
- Working with the DialogFragment class
- Interacting with a DialogFragment class as a traditional Dialog class
- Wrapping an existing Dialog class in a DialogFragment class

By the end of this chapter, we will be able to coordinate the setup and teardown of fragments within their host activities and effectively utilize the ListFragment and DialogFragment classes.
Understanding the fragment life cycle

One of the challenges of developing Android applications is to ensure that our applications effectively handle the life cycle of the application's activities. During the lifetime of an application, a given activity may be created, destroyed, and recreated many times. A simple action, such as a user rotating a device from the portrait to landscape orientation or vice-versa, normally causes the visible activity to be completely destroyed and recreated using the appropriate resources for the new orientation. Applications that do not cooperate effectively with this natural life cycle often crash or behave in some other undesirable manner.

Each fragment instance exists within a single activity; therefore, this fragment must cooperate in some way with the activity life cycle. In fact, not only do fragments cooperate with the activity life cycle, but also they are intimately connected.

In the setup and display phases, as well as in the hide and teardown phases, fragments provide many of the same life cycle-related callback methods as activities. In addition, fragments provide additional life cycle-related callback methods that relate to the fragment's relationship with the containing activity.

As our applications become more sophisticated and we work with more specialized implementations of the fragment class, understanding the fragment class' life cycle and the relationship with the activity life cycle is essential.

If you are unfamiliar with the basics of Android's activity life cycle callback methods, refer to the Activity Lifecycle section of the Android Activity documentation at http://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle.

Understanding fragment setup and display

Fragment setup and display is a multiphase process involving the fragment's association with an activity, its creation, and the standard life cycle events of moving the activity into the running state (also known as the resumed or active state). Understanding the behavior of the life cycle events and the associated callback methods is essential to use fragments effectively. Once we have an understanding of the life cycle events and callback methods, we'll look at just how the event callback methods are used.
The following figure shows the sequence of life cycle-related callback method calls that occur on fragments and activities during setup and display:

As you might expect in most cases, the first step in the setup and display of a fragment occurs in the activity's `onCreate` method. In most cases, the activity calls the `setContentView` method from within the activity's `onCreate` callback method, which then loads the layout resource and triggers the activity's association with the contained fragments.

Note what happens next. Before the fragment is even created, it is attached to the activity. The fragment is first notified of the attachment and receives a reference to the activity through the `onAttach` callback method. The activity is then notified and receives a reference to the fragment through the `onAttachFragment` callback method.

Although attaching the fragment to the activity prior to creating the fragment may seem unexpected, doing so is useful. In many cases, the fragment needs access to the activity during the creation process because the activity often contains information that the fragment will display or that is otherwise important to the fragment's creation process.

Attached to the activity, the fragment then performs general creation work in the `onCreate` method and then constructs the contained view hierarchy in the `onCreateView` method. We'll talk more about which actions are appropriate to perform in each method in the Maximizing the available resources section later in this chapter.
When an activity contains multiple fragments, Android calls the four methods `Fragment.onAttach`, `Activity.onAttachFragment`, `Fragment.onCreate`, and `Fragment.onCreateView` in succession for one fragment before making any calls to these methods for the next fragment. This allows each fragment to complete the process of attachment and creation before the next fragment begins this process.

Once the sequence of calling these four methods is complete for all the fragments, the remaining setup and display callback methods are called individually in succession for each fragment.

After the activity completes the execution of its `onCreate` method, Android then calls each fragment's `onActivityCreated` method. The `onActivityCreated` method indicates that all views and fragments created by the activity's layout resource are now fully constructed and can be safely accessed.

At this point, the fragment receives the standard life cycle callbacks on the `onStart` and `onResume` methods just after each of the activity methods of the same name is called. Any work performed in the fragment's `onStart` and `onResume` methods is very much like the work performed in the corresponding methods within an activity.

For many fragments, the only methods in this part of their life cycle that are overridden are the `onCreate` and `onCreateView` methods, as we noted in the examples in the previous chapters.

**Avoiding method name confusion**

The activity and fragment classes have a number of commonly named callback methods, and most of these commonly named methods have a common purpose. One important exception is the `onCreateView` method. The purpose of this method is very different for each class.

As mentioned previously, Android calls the fragment class' `onCreateView` method to give the fragment an opportunity to create and return the fragment's contained view hierarchy. This method is commonly overridden within a fragment.

The method of the same name in the activity class is called repeatedly by the `LayoutInflater` class during the process of inflating a layout resource. Most activity implementations do not override this method.
Understanding fragment hide and teardown

Just as fragments behave in a similar way to activities during setup and display, they also behave in a similar way during hide and teardown, as shown in the following figure:

Initially, during hide and teardown, fragments behave just as activities. When the user switches to another activity, each fragment's `onPause`, `onSaveInstanceState`, and `onStop` methods are called. For each method, the fragment implementation is called first, followed by the activity implementation.

After the `onStop` method is called, fragments begin to behave a little differently than activities. Consistent with the separation of fragment creation from fragment view hierarchy creation, fragment view hierarchy destruction is separate from fragment destruction. Following the call to the activity's `onStop` method, the fragment's `onDestroyView` method is called, indicating that the view hierarchy returned by the fragment's `onCreateView` method is being destroyed. The fragment's `onDestroy` method is then called, followed by the fragment's `onDetach` method. At this point, the fragment has no association with an activity and any calls to the `getActivity` method will return null.
For activities containing multiple fragments, Android calls the sequence of the three methods onDestroyView, onDestroy, and onDetach for an individual fragment before beginning the sequence of calling these three methods for the next fragment. This groups the process of destroying and detaching each fragment similar to the way Android groups the process of attaching and creating each fragment. Once this sequence is completed for all fragments, Android calls the activity's onDestroy method.

**Maximizing the available resources**

For the most part, life cycle management for a fragment is very much like that of an activity. There is, however, one important exception: the two-phase nature of fragment creation and destruction. Fragments separate their creation and destruction from their contained view hierarchy. This is because fragments have the ability to exist and be associated with an activity in the absence of their view hierarchy.

There are many scenarios where an activity contains multiple fragments but has only a subset of these fragments visible at any point in time. In such a case, the contained fragments can have their onAttach and onCreate methods called. But the call to each fragment's onCreateView method is delayed until the time comes for the app to make the contents of this fragment visible. Similarly, when the time comes to hide the contents of a fragment, only the fragment's onDestroyView method is called, not the onDestroy and onDetach methods.

This behavior comes into play when fragments are dynamically managed within an activity. This behavior allows the overhead of associating a fragment with an activity and initializing the fragment's state to occur only once while being able to easily change the visibility of the fragment's view hierarchy. This is important when we explicitly manage the visibility of fragments using the FragmentTransaction class and certain action bar features that manage fragments. We'll talk about these issues in the next two chapters.

**Managing a fragment state**

For many fragment implementations, the most important callback method in the life cycle sequence is the onSaveInstanceState method. Just as with an activity, this callback method provides the fragment with an opportunity to persist any state before the fragment is destroyed, such as when the user moves to another activity or when the user rotates the device to a different orientation. In both these cases, the activity and contained fragments may be completely torn down and recreated. By persisting the fragment state in the onSaveInstanceState method, this state is later passed back to the fragment in both the onCreate and onCreateView methods.
When managing the state of a fragment, you want to be sure to separate work that is general to the fragment's overall existence from being work-specific to setting up the view hierarchy. Any expensive initialization work that's general to the fragment's existence, such as connecting to a data source, complex calculations, or resource allocations, should occur in the `onCreate` method rather than the `onCreateView` method. This way, if only the fragment's view hierarchy is destroyed and the fragment remains intact, you avoid unnecessarily repeating expensive initialization work.

**Special-purpose fragment classes**

Now that we understand the life cycle of fragments, we can look at some of the specialized versions of the `Fragment` class. As we go through each of these specialized classes, remember that they all ultimately inherit from the `Fragment` class and therefore experience the same life cycle behavior. Many of these specialized classes have an impact on which operations are safe to perform at the various points in the life cycle, and some of these classes even add their own life cycle methods. Understanding each of these classes and their interaction with the fragment life cycle is essential to using the classes effectively.

**ListFragment**

One of the simplest fragment derived classes to use, and yet one of the most helpful, is the `ListFragment` class. The `ListFragment` class provides a fragment that encapsulates `ListView` and, as the name implies, is useful for displaying lists of data.

**Associating data with the list**

Unlike the base `Fragment` class, we're not required to override the `onCreateView` callback method for the `ListFragment` class. The `ListFragment` class provides a standard appearance and only requires that we associate some data. The `ListFragment` class does all the work of creating the view hierarchy and displaying this data.

We will associate the data with the `ListFragment` class by calling the `ListFragment` class' `setListAdapter` method and passing a reference to an object that implements the `ListAdapter` interface. Android provides a number of classes that implement this interface, such as `ArrayAdapter`, `SimpleAdapter`, and `SimpleCursorAdapter`. The specific class you use will depend on how your source data is stored. If none of the standard Android classes meet your specific requirements, you can create a custom implementation reasonably easily.
For a discussion about creating a custom list adapter, take a look at the Displaying the Quick Contact Badge Android tutorial at http://developer.android.com/training/contacts-provider/display-contact-badge.html.

The ListFragment class wraps an instance of the ListView class, which is accessible through the getListView method. In most scenarios, we can feel free to interact with the contained ListView instance directly and take advantage of any features offered by the ListView class. The one very important exception is when we set the ListAdapter instance. Both the ListFragment and ListView classes expose a setListAdapter method, but we must be sure to use the ListFragment version of the method.

The ListFragment class relies on certain initialization behaviors that occur within the ListFragment.setListAdapter method; therefore, the process of calling the setListAdapter method directly on the contained ListView instance bypasses this initialization behavior and may cause the application to become unstable.

### Separating data from the display

Up until now, our application has used a fixed layout of several RadioButton views to display the list of books. Using a fixed layout to display such options is not generally a good choice because any changes to the book list require that we go in and directly modify the fragment layout. In practice, we would prefer to have a layout that is independent of the specific titles. We could write code to dynamically generate the RadioButton views, but there is an easier way. We can instead use the ListFragment class.

By switching our application to use the ListFragment class, we can simply store the list of book titles in an array resource and associate the contents of this array resource with the ListFragment instance. In the event of adding more titles or needing to change one of the titles, we can simply modify the array resource file. There is no need for us to make any changes to the actual fragment layout.

Our application already has all the book titles stored as individual string resources, so we just need to add an array resource for them. We'll add the book titles array to the course_arrays.xml resource file within the values resource folder, where we currently have an array resource defined to hold the list of book descriptions.
Within the resources root element of the course_arrays.xml resource file, add a string-array element that includes a name attribute with a value of bookTitles. Within the string-array element, add an item for each book title that references the string resource for each title. We want to be sure that we list the book title array entries in the same order as the book_descriptions array entries because we use the array index as the ID value for each book when we notify the activity of the user's book selection. The array resource entries for the book title and description arrays appear as follows:

```xml
<resources>
  <!-- Book Titles -->
  <string-array name="book_titles">
    <item>@string/dynamicUiTitle</item>
    <item>@string/android4NewTitle</item>
    <item>@string/androidSysDevTitle</item>
    <item>@string/androidEngineTitle</item>
    <item>@string/androidDbProgTitle</item>
  </string-array>

  <!-- Book Descriptions -->
  <string-array name="book_descriptions">
    <item>@string/dynamicUiDescription</item>
    <item>@string/android4NewDescription</item>
    <item>@string/androidSysDevDescription</item>
    <item>@string/androidEngineDescription</item>
    <item>@string/androidDbProgDescription</item>
  </string-array>
</resources>
```

With the titles stored as an array resource, we can now easily create a ListFragment derived class to display the book titles.

**Creating the ListFragment derived class with Android Studio**

The first step is to add a new class to our project. To do this, we'll create a new class named BookListFragment2 that extends the ListFragment class. In Chapter 1, Fragments and UI Modularization, we created the fragment class manually. For the BookListFragment2 class, we'll use Android Studio.
To create the `BookListFragment2` class, we first need to open the **New Android Activity** dialog by performing the following steps:

1. Select the Android Studio **File** menu.
2. Then, select **New**.
3. Select **Fragment**.
4. Select **Fragment (List)**.

Now, we will perform the following steps within the **New Android Activity** dialog:

1. In the **Object Kind** field, enter **String**.
2. In the **Fragment class name** field, enter `BookListFragment2`.
3. Then, unselect the **Include fragment factory methods?** checkbox.
4. Unselect the **Switch to grid view on large screens?** checkbox.

The **New Android Activity** dialog should now look similar to the following screenshot:
Click on the Finish button to complete the creation of the `BookListFragment2` class.

The generated class has the `onCreate` method stubbed to populate the list with dummy data. To load in the list of book titles, update the `onCreate` method, as shown in the following code:

```java
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // TODO: Change Adapter to display your content
    String[] bookTitles =
        getResources().getStringArray(R.array.book_titles);
    setListAdapter(new ArrayAdapter<String>(getActivity(),
        android.R.layout.simple_list_item_1,
        android.R.id.text1, bookTitles));
}
```

In the `onCreate` method, we will first call the base class implementation that is required by all classes that extend `ListFragment`. We will then load the `bookTitles` array resource. We will call the `setListAdapter` method by passing an instance of the `ArrayAdapter`. The array adapter takes the context as the first parameter, which we will get by accessing the activity, and then it takes the array as the third parameter. The second parameter is the ID of the resource used to lay out each entry in the list. This resource can be a custom resource or one of the built-in Android resources. In our case, we will use the built-in Android layout resource `android.R.layout.simple_list_item_1`, which displays a single string value for each row within `ListView`.

Creating a custom layout resource for the `ListFragment` class is just like doing so for the `ListView` class and is discussed in detail in the Android developer documentation at [http://developer.android.com/reference/android/app/ListFragment.html](http://developer.android.com/reference/android/app/ListFragment.html).

**Handling the ListFragment item selection**

For our application to work correctly, we need to inform the activity each time the user selects one of the titles. As we use an interface to loosely couple our fragment with the activity, this turns out to be a pretty simple task.
When Android Studio generates the BookListFragment2 class, it includes a nested interface declaration within the BookListFragment2 class named OnFragmentInteractionListener along with code to use the interface to notify the activity of user selections within the list. Scroll to the bottom of the BookListFragment2 class, and you'll see the OnFragmentInteractionListener interface declaration, as shown in the following code:

```java
public interface OnFragmentInteractionListener {
    // TODO: Update argument type and name
    public void onFragmentInteraction(String id);
}
```

As we previously created our OnSelectedBookChangeListener interface, we don't need the OnFragmentInteractionListener interface; so, we can delete it and then update the BookListFragment2 class to use our existing OnSelectedBookChangeListener interface.

Scroll to the top of the BookListFragment2 class and locate the mListener field declaration as shown in the following code:

```java
private OnFragmentInteractionListener mListener;
```

Update the mListener field declaration to use the OnSelectedBookChangeListener interface so that the declaration now appears as shown in the following code:

```java
private OnSelectedBookChangeListener mListener;
```

With the mListener field, we are able to store a reference to the containing activity as an OnSelectedBookChangeListener interface reference. The generated BookListFragment2 class code sets the mListener reference in the onAttach callback method. As we discussed earlier in this chapter, the onAttach method is called when the fragment instance is attached to the containing activity and receives a reference to this activity. Update the onAttach method to use the OnSelectedBookChangeListener interface rather than the OnFragmentInteractionListener interface so that the method now appears as shown in the following code:

```java
public void onAttach(Activity activity) {
    super.onAttach(activity);
    try {
        mListener = (OnSelectedBookChangeListener) activity;
    } catch (ClassCastException e) {
        throw new ClassCastException(activity.toString()
            + " must implement OnSelectedBookChangeListener");
    }
}
```
The `onAttach` method simply assigns the activity to the `mListener` field casting the activity to the `OnSelectedBookChangeListener` interface. The method also includes a `try\catch` block to display an appropriate error message if the containing activity does not implement the `OnSelectedBookChangeListener` interface.

The generated `BookListFragment2` class includes an `onListItemClick` method that is called when the user makes a selection from the list and receives several selection-related parameters, including the zero-based position of the user selection. Update the `onListItemClick` method to use the `OnSelectedBookChangeListener` interface so that the method appears as shown in the following code:

```java
public void onListItemClick(ListView l, View v, int position, long id) {
    super.onListItemClick(l, v, position, id);
    if (null != mListener) {
        mListener.onSelectedBookChanged(position);
    }
}
```

After calling the `onListItemClick` method on the base class, the preceding code verifies that the `mListener` field is set. If it has, the `onSelectedBookChanged` method is called, passing the position of the user selection. This code will now inform the activity each time the user makes a selection from the list, just as the `BookListFragment` class implementation did when the user selected a radio button.

All the activity classes in our application that use our `BookListFragment2` class already implement the `OnSelectionChangeListener` interface, so there is no change required to the activity classes.

**Updating the layout resources**

We will now update the `activity_main.xml` resource file to use the `BookListFragment2` class instead of the original `BookListFragment` class, as shown in the following code:

```xml
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- List of Book Titles ** using the ListFragment **-->
    <fragment
```
Fragment Life Cycle and Specialization

android:layout_width="match_parent"
android:layout_height="0dp"
android:layout_weight="1"
android:name="com.jwhh.fragments.BookListFragment2"
android:id="@+id/fragmentTitles"/>

<!-- Description of selected book -->
<fragment
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:name="com.jwhh.fragments.BookDescFragment"
    android:id="@+id/fragmentDescription"/>
</LinearLayout>

We need to make the same change in the activity_main_wide.xml file.

Our program is now fully functional using the ListFragment class and appears as follows:

<table>
<thead>
<tr>
<th>Android Books</th>
<th>This book introduces readers to Fragments, the specific scenarios Fragments address and how to use them. The book will take the reader from the basics of using Fragments, through their use in creating more adaptive and dynamic user experiences. The book teaches readers how to capitalize on many of the built-in UI features that are available through Fragments such as list-based navigation, tab-based navigation, and layout adaptation,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating Dynamic UI with Android Fragments</td>
<td></td>
</tr>
<tr>
<td>Android 4: New Features for Application Development</td>
<td></td>
</tr>
<tr>
<td>Instant Android Systems Development How-to</td>
<td></td>
</tr>
<tr>
<td>AndEngine for Android Game Development Cookbook</td>
<td></td>
</tr>
<tr>
<td>Android Database Programming</td>
<td></td>
</tr>
</tbody>
</table>

Any changes that we need to make to the titles can now all be made in the resources file and require no changes to the user interface code.
DialogFragment

Up until now, we've been looking at fragments as a new way to divide our application's user interface into subsections of the available display area. Although fragments are new, the concept of having an aspect of our application user interface as a subsection of the available display area is not new. Whenever an application displays a dialog, the application does exactly this.

Historically, the challenge of working with dialogs is that, even though they are conceptually just another window within an application, we must handle many of the tasks related to dialogs differently than other aspects of our application user interface. Doing something as simple as handling a button click requires the dialog-specific DialogInterface.OnClickListener interface, rather than the View.OnClickListener interface that we use when handling a click event from non-dialog-related parts of our user interface code. An even more complicated issue is that of orientation changes. Dialogs automatically close in response to an orientation change and therefore can create inconsistent application behavior if a user changes device orientation while a dialog is visible.

The DialogFragment class eliminates much of the special handling related to dialogs. With the DialogFragment class, displaying and managing a dialog becomes much more consistent with other aspects of our application user interface.

Styles

When an application displays an instance of the DialogFragment class, the window for the DialogFragment instance has up to three parts to it: the layout area, title, and frame. A DialogFragment instance always contains the layout area, but we can control whether it includes the title and frame by setting the DialogFragment class' style using the setStyle method. The DialogFragment class supports four styles with an instance of the DialogFragment class having exactly one style applied. The following table shows the four available styles:

<table>
<thead>
<tr>
<th>Style</th>
<th>Has title</th>
<th>Has frame</th>
<th>Accepts input</th>
</tr>
</thead>
<tbody>
<tr>
<td>STYLE_NORMAL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>STYLE_NO_TITLE</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>STYLE_NO_FRAME</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>STYLE_NO_INPUT</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Note that the styles remove features cumulatively. For example, STYLE_NO_TITLE indicates no title, whereas STYLE_NO_FRAME indicates no frame and no title. If we do not call the setStyle method, Android creates the DialogFragment instance with the style set to STYLE_NORMAL.

The style affects the remainder of the behavior of the DialogFragment class and therefore must be set in the onCreate callback method. An attempt to set the DialogFragment class' style any later in the life cycle is ignored.

If you wish to provide the dialog with a special theme, the theme's resource ID can also be passed to the setStyle method. To allow Android to select an appropriate theme based on the style, simply pass 0 as the theme resource ID. The following code sets the DialogFragment instance to have no title and use the Android-selected theme for this style, as in the following code:

```java
class MyDialogFragment extends DialogFragment {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setStyle(DialogFragment.STYLE_NO_TITLE, 0);
    }
}
```

### Layout

Populating the layout of an instance of the DialogFragment class is similar to that of a standard fragment derived class. We will simply override the onCreateView method and inflate the layout resource via the following code:

```java
public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
    View theView = inflater.inflate(R.layout.fragment_my_dialog,
    container, false);
    return theView;
}
```

Creating a layout resource for use with a DialogFragment derived class works exactly like creating a layout resource for any other fragment derived class. To have our DialogFragment instance display a line of text and two buttons, we will define the fragment_my_dialog.xml layout resource as shown in the following XML:

```xml
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
```
Displaying DialogFragment

Displaying our DialogFragment derived class is largely just a matter of creating the class instance and calling the show method. We need to keep in mind, though, that although our DialogFragment instance appears as a standard dialog when it displays, it is actually a fragment. As with all fragments, it is managed by the containing activity's FragmentManager instance. As a result, we need to pass a reference to the activity's FragmentManager instance as part of the call to the DialogFragment class' show method, as we do in the following code:

```java
MyDialogFragment theDialog = new MyDialogFragment();
theDialog.show(getFragmentManager(), null);
```
With our `DialogFragment` derived class' style set to `STYLE_NO_TITLE` and using the `fragment_my_dialog.xml` layout resource file shown earlier, the previous code displays the following screenshot:

![Do you really want to?](image)

### Event handling in DialogFragment

One of the key values of the `DialogFragment` class is that it provides greater consistency in our code than that available when using the traditional `Dialog` class. Most aspects of working with the `DialogFragment` class are the same as when working with other fragments. No longer does displaying a dialog have to be handled so differently than other aspects of our application user interface. For example, no special handling is required to deal with orientation changes. Another place where this greater consistency is evident is in event handling because our button click event handling can use the standard view class event interfaces.

To handle the button clicks, our `DialogFragment` derived class simply implements the `View.OnClickListener` interface. The following code shows setting the yes and no button click events to call back to our `DialogFragment` derived class in our class' `onCreateView` callback method:

```java
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View theView = inflater.inflate(R.layout.fragment_my_dialog, container, false);

    // Connect the Yes button click event and request focus
    View yesButton = theView.findViewById(R.id.btnYes);
    yesButton.setOnClickListener(this);
    yesButton.requestFocus();

    // Connect the No button click event
    View noButton = theView.findViewById(R.id.btnNo);
    noButton.setOnClickListener(this);

    return theView;
}
```
Note that we're setting up the button click handling just as we would if we were working within any other fragment or even directly within the activity.

We can also handle notifying the activity of the user's interaction with the DialogFragment derived class consistently with the way we do with other fragments. Just as we did when notifying the activity of book title selections, our DialogFragment derived class simply provides an interface to notify the activity which of the available buttons the user selected, as shown in the following code:

```java
public class MyDialogFragment extends DialogFragment implements View.OnClickListener {
    // Interface Activity implements for notification
    public interface OnButtonClickListener {
        void onButtonClick(int buttonId);
    }
    // Other members elided for clarity
}
```

As long as the activity implements the interface, our DialogFragment derived class can notify the activity of the button that the user clicked.

In the handler for our button click events, we'll follow the same pattern as we did in the previous chapter. We will access the containing activity, cast it to the expected interface, and call the interface method, as shown in the following code:

```java
public void onClick(View view) {
    int buttonId = view.getId();

    // Notify the Activity of the button selection
    OnButtonClickListener parentActivity = (OnButtonClickListener) getActivity();
    parentActivity.onButtonClick(buttonId);

    // Close the dialog fragment
    dismiss();
}
```

Note that there is one bit of special handling in the onClick method. Just as with the traditional Dialog class, we must call the dismiss method on the DialogFragment derived class when we no longer wish to display it.
The Dialog identity

Although we treat our DialogFragment derived class as just another fragment, a part of its identity is still tied to the traditional Dialog class. In fact, Android actually wraps our DialogFragment derived class within a traditional Dialog instance. This occurs in a callback method specific to the DialogFragment class named onCreateDialog that Android calls just prior to calling the onCreateView callback method.

The Dialog instance that the onCreateDialog method returns is the window that is ultimately displayed to the user. The layout we create within our DialogFragment derived class is simply wrapped within the Dialog window. We can access this Dialog instance later in the life cycle to access behavior related to the Dialog class or even override the method to provide our own Dialog instance.

Accessing behavior related to Dialog

Accessing the behavior related to Dialog of our DialogFragment derived class requires a reference to the Dialog instance created in the onCreateDialog method. We retrieve this reference by calling the getDialog method. Once we have the reference to the Dialog instance, we can access aspects of the class’ Dialog identity that are not otherwise available.

When we create a DialogFragment derived class with the style set to STYLE_NORMAL, the displayed dialog includes a title area above the layout area. The value of the title can only be set by calling the setTitle method on the Dialog instance that wraps our DialogFragment instance. A similar issue arises in dealing with the dialog-cancellation behavior. By default, the user can cancel a dialog by tapping on the activity behind the dialog. In many cases, this may be unacceptable as we want to require the user to acknowledge one of the choices within the dialog. The following code sets these behaviors related to Dialog after the button click handling is set up:

```java
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View theView = inflater.inflate(R.layout.fragment_my_dialog, container, false);

    View yesButton = theView.findViewById(R.id.btnYes);
    yesButton.setOnClickListener(this);
    yesButton.requestFocus();

    View noButton = theView.findViewById(R.id.btnNo);
    noButton.setOnClickListener(this);
```

[58]
// Set the dialog aspects of the dialog fragment
Dialog dialog = getDialog();
dialog.setTitle(getString(R.string.myDialogFragmentTitle));
dialog.setCanceledOnTouchOutside(false);

return theView;
}

The code first sets the dialog title and then sets the option to prevent the user from closing the dialog by tapping on the activity window. For the call to the setTitle method to work, we will need to change the call to the setStyle method in the onCreate callback method to set the style to STYLE_NORMAL so that the dialog will have a title area.

**Wrapping an existing dialog in a fragment**

There may be times where we like the programming consistency that the DialogFragment class offers but want to take advantage of the features provided by a class that is derived from the traditional Dialog class. By overriding the DialogFragment class' onCreateDialog method, we can do exactly this. Overriding the onCreateDialog method allows us to replace the DialogFragment class' default Dialog instance with the one we create. A great example of when this is useful is in leveraging the Android AlertDialog class.

The AlertDialog class provides a variety of default behaviors and allows us to display text, an icon, and buttons all without having to create a layout resource. There is something we must keep in mind when we leverage a class that inherits from the traditional Dialog class. Although outside interaction with our class will be consistent with other DialogFragment derived classes, any interactions with the traditional Dialog class that occur within our DialogFragment derived class will be done in the traditional Dialog class way. For example, to create a DialogFragment derived class that utilizes the AlertDialog class requires that our class implement the Dialog class way of handling click events; that is, it must implement the DialogInterface.OnClickListener interface, as shown in the following code:

```java
public class AlertDialogFragment extends DialogFragment
    implements DialogInterface.OnClickListener{
    }
```
Within our class' `onCreateDialog` method, we will create the `AlertDialog` instance using the `AlertDialog.Builder` class just as if we were going to display the `AlertDialog` instance directly. Within the `onCreateDialog` method, we will set all the options on the `AlertDialog.Builder` instance, including the title, message, icon, and buttons. Note that we never call the `AlertDialog.Builder` class' `show` method; instead, we call its `create` method. We will then take the reference to the newly created `AlertDialog` instance and return it from the `onCreateDialog` method. All of these steps are shown in the following code:

```java
public Dialog onCreateDialog(Bundle savedInstanceState) {
    // Create the Builder for the AlertDialog
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

    // Set the AlertDialog options
    builder.setTitle(R.string.alert_dialog_title)
        .setMessage(R.string.alert_dialog_message)
        .setIcon(R.drawable.ic_launcher)
        .setCancelable(false)
        .setPositiveButton(R.string.text_yes, this)
        .setNegativeButton(R.string.text_no, this);

    // Create and return the AlertDialog
    AlertDialog alertDialog = builder.create();
    return alertDialog;
}
```

The `Dialog` instance we create is now managed as a part of the `DialogFragment` instance. Everything else we do with our `AlertDialogFragment` class will be just as it is with the other `DialogFragment` derived classes we create.

When our app shows our `AlertDialogFragment` class, it looks like this:

![AlertDialog Fragment](image)
Note that we didn't need to override the `onCreateView` callback method because the `Dialog` instance we created in the `onCreateDialog` callback method provides the desired display characteristics.

Overriding the `DialogFragment` class' `onCreateDialog` callback method is a powerful technique that allows us to enjoy the benefits of the `DialogFragment` class while still leveraging any existing investment we may have in traditional `Dialog` classes—whether they are a built-in class, such as the `AlertDialog` class, or a custom `Dialog` class that we may have as part of our own code library.

**Summary**

Understanding the fragment life cycle empowers us to leverage the phases of creation and destruction of fragments to more efficiently manage fragments and the data associated with them. By working with this natural life cycle, we can take advantage of the specialized fragment classes to create a rich user experience while following a more consistent programming model than was previously available.

In the next chapter, we will build on our understanding of the fragment life cycle to take more direct control of fragments to dynamically add and remove them within individual activities.
Where to buy this book
You can buy Creating Dynamic UIs with Android Fragments Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.