7

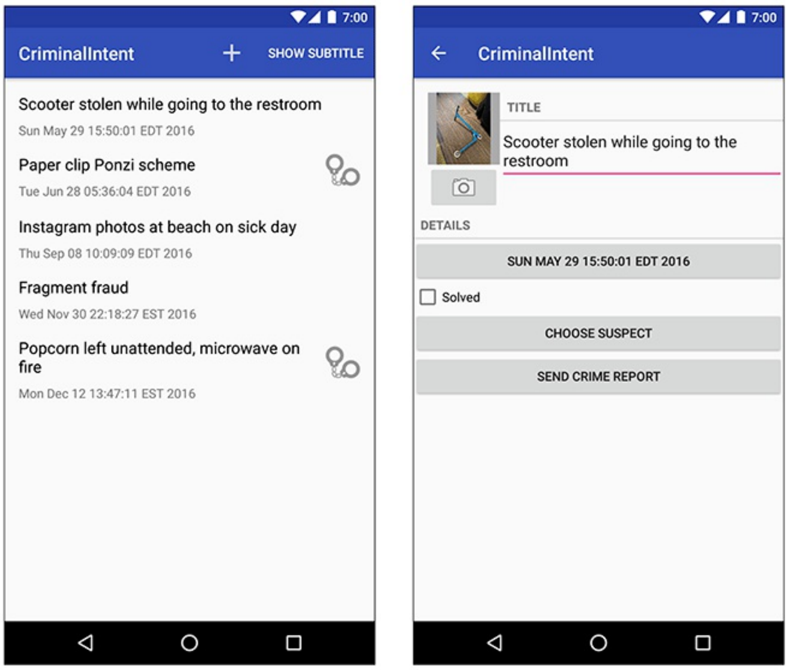
UI Fragments and the Fragment Manager

In this chapter, you will start building an application named CriminalIntent. CriminalIntent records the details of "office crimes" – things like leaving dirty dishes in the breakroom sink or walking away from an empty shared printer after documents have printed.

With CriminalIntent, you can make a record of a crime including a title, a date, and a photo. You can also identify a suspect from your contacts and lodge a complaint via email, Twitter, Facebook, or another app. After documenting and reporting a crime, you can proceed with your work free of resentment and ready to focus on the business at hand.

CriminalIntent is a complex app that will take 13 chapters to complete. It will have a list-detail interface: The main screen will display a list of recorded crimes, and users will be able to add new crimes or select an existing crime to view and edit its details (Figure 7.1).

Figure 7.1 CriminalIntent, a list-detail app



Scooter stolen while going to the restroom
Sun May 29 15:50:01 EDT 2016
Paper clip Ponzi scheme
Tue Jun 28 05:36:04 EDT 2016
Instagram photos at beach on sick day
Thu Sep 08 10:09:09 EDT 2016
Fragment fraud
Wed Nov 30 22:18:27 EST 2016
Popcorn left unattended, microwave on fire
Mon Dec 12 13:47:11 EST 2016

Scooter stolen while going to the restroom
SUN MAY 29 15:50:01 EDT 2016

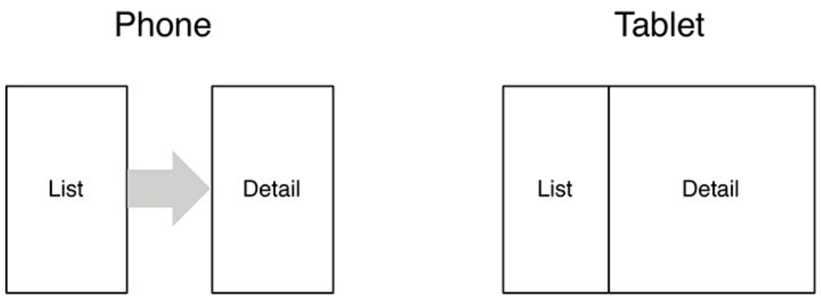
The Need for UI Flexibility

You might imagine that a list-detail application consists of two activities: one managing the list and the other managing the detail view. Clicking a crime in the list would start an instance of the detail activity. Pressing the Back button would destroy the detail activity and return you to the list, where you could select another crime.

That would work, but what if you wanted more sophisticated presentation and navigation between screens?

- Imagine that your user is running CriminalIntent on a tablet. Tablets and some larger phones have screens large enough to show the list and detail at the same time – at least in landscape orientation (Figure 7.2).

Figure 7.2 Ideal list-detail interface for phone and tablet



- Imagine the user is viewing a crime on a phone and wants to see the next crime in the list. It would be better if the user could swipe to see the next crime without having to return to the list. Each swipe should update the detail view with information for the next crime.

What these scenarios have in common is UI flexibility: the ability to compose and recompose an activity's view at runtime depending on what the user or the device requires.

Activities were not built to provide this flexibility. An activity's views may change at runtime, but the code to control those views must live inside the activity. As a result, activities are tightly coupled to the particular screen being used.

Introducing Fragments

You can get around the letter of the Android law by moving the app's UI management from the activity to one or more *fragments*.

A *fragment* is a controller object that an activity can deputize to perform tasks. Most commonly, the task is managing a UI. The UI can be an entire screen or just one part of the screen.

A fragment managing a UI is known as a *UI fragment*. A UI fragment has a view of its own that is inflated from a layout file. The fragment's view contains the interesting UI elements that the user wants to see and interact with.

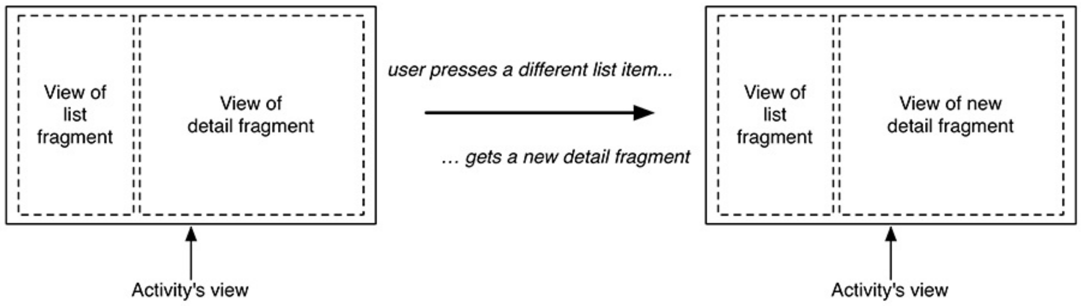
The activity's view contains a spot where the fragment's view will be inserted. In fact, while in this chapter the activity will host a single fragment, an activity can have several spots for the views of several fragments.

You can use the fragment(s) associated with the activity to compose and recompose the screen as your app and users require. The activity's view technically stays the same throughout its lifetime, and no laws of Android are violated.

Let's see how this would work in a list-detail application to display the list and detail together. You would compose the activity's view from a list fragment and a detail fragment. The detail view would show the details of the selected list item.

Selecting another item should display a new detail view. This is easy with fragments; the activity will replace the detail fragment with another detail fragment (Figure 7.3). No activities need to die for this major view change to happen.

Figure 7.3 Detail fragment is swapped out



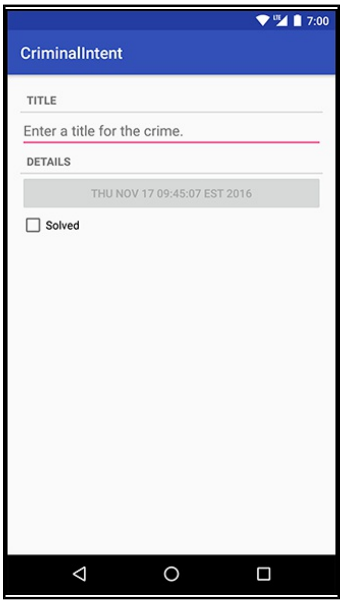
Using UI fragments separates the UI of your app into building blocks, which is useful for more than just list-detail applications. Working with individual blocks, it is easy to build tab interfaces, tack on animated sidebars, and more.

Achieving this UI flexibility comes at a cost: more complexity, more moving parts, and more code. You will reap the benefits of using fragments in Chapter 11 and Chapter 17. The complexity, however, starts now.

Starting CriminalIntent

In this chapter, you are going to start on the detail part of CriminalIntent. Figure 7.4 shows you what CriminalIntent will look like at the end of this chapter.

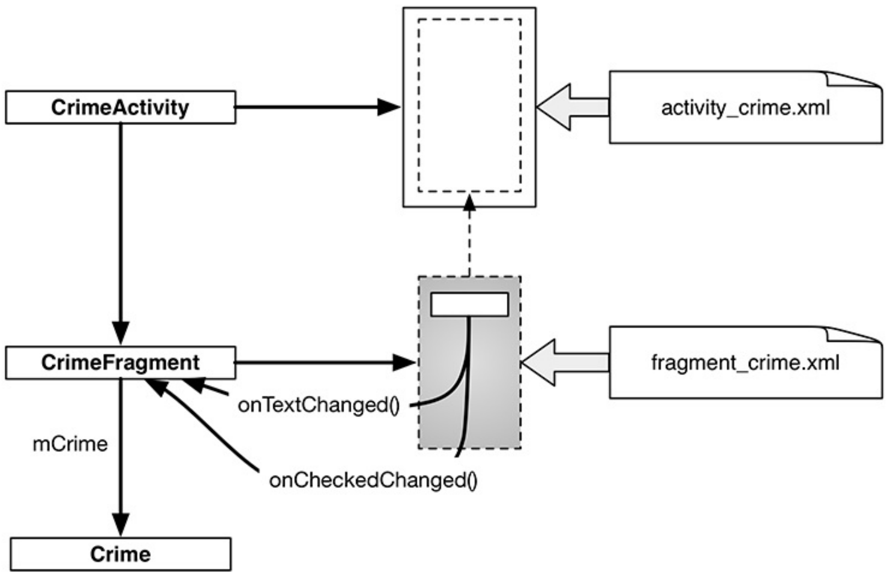
Figure 7.4 CriminalIntent at the end of this chapter



The screen shown in Figure 7.4 will be managed by a UI fragment named **`CrimeFragment`**. An instance of **`CrimeFragment`** will be *hosted* by an activity named **`CrimeActivity`**.

For now, think of hosting as the activity providing a spot in its view hierarchy where the fragment can place its view (Figure 7.5). A fragment is incapable of getting a view on screen itself. Only when it is placed in an activity's hierarchy will its view appear.

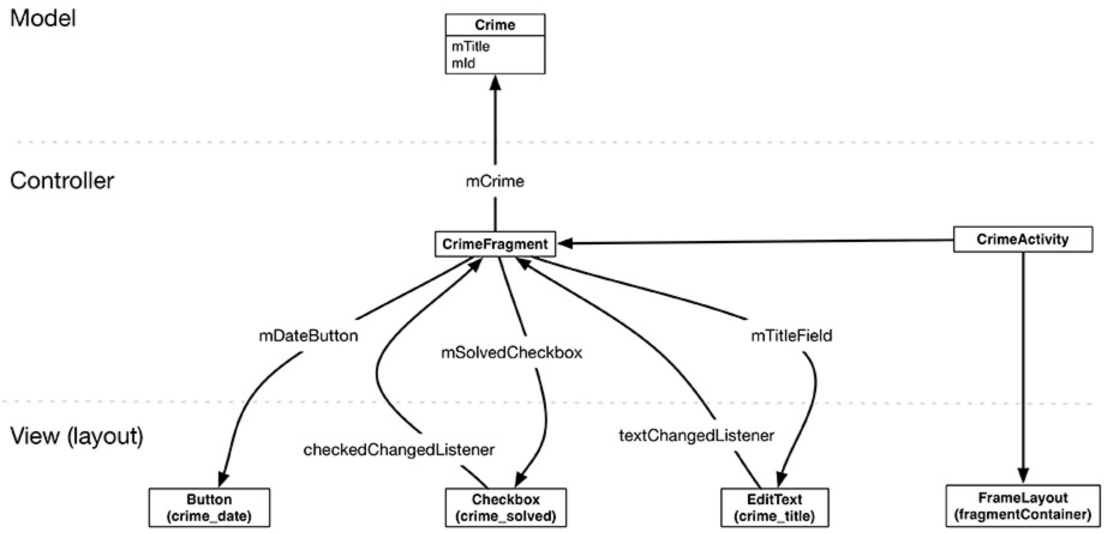
Figure 7.5 **`CrimeActivity`** hosting a **`CrimeFragment`**



CriminalIntent will be a large project, and one way to keep your head wrapped around a project is with an object diagram. Figure 7.6 gives you the big picture of CriminalIntent. You do not have to memorize these objects and their relationships, but it is good to have an idea of where you are heading before you start.

You can see that **CrimeFragment** will do the sort of work that your activities did in GeoQuiz: create and manage the UI and interact with the model objects.

Figure 7.6 Object diagram for CriminalIntent (for this chapter)



Three of the classes shown in Figure 7.6 are classes that you will write: **Crime**, **CrimeFragment**, and **CrimeActivity**.

An instance of **Crime** will represent a single office crime. In this chapter, a crime will have a title, an ID, a date, and a `boolean` that indicates whether the crime has been solved. The title is a descriptive name, like “Toxic sink dump” or “Someone stole my yogurt!” The ID will uniquely identify an instance of **Crime**.

For this chapter, you will keep things very simple and use a single instance of **Crime**. **CrimeFragment** will have a member variable (`mCrime`) to hold this isolated incident.

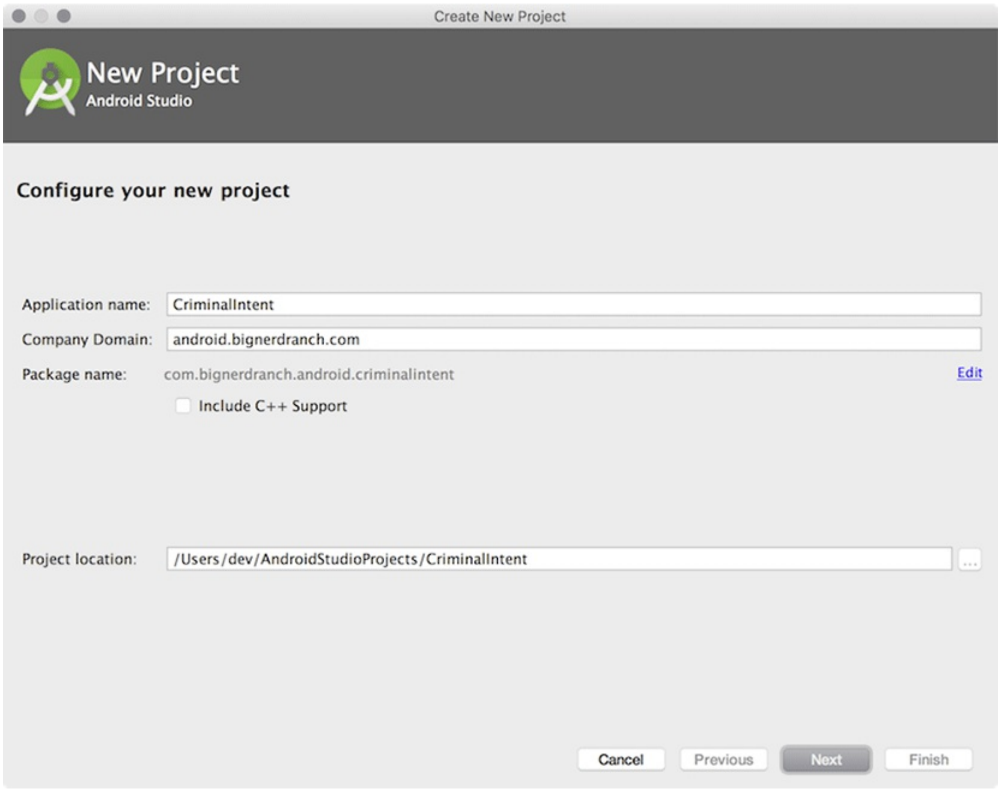
CrimeActivity’s view will consist of a **FrameLayout** that defines the spot where the **CrimeFragment**’s view will appear.

CrimeFragment’s view will consist of a **LinearLayout** with a few child views inside of it, including an **EditText**, a **Button**, and a **CheckBox**. **CrimeFragment** will have member variables for each of these views and will set listeners on them to update the model layer when there are changes.

Creating a new project

Enough talk; time to build a new app. Create a new Android application (File → New Project...). Name the application CriminalIntent and make sure the company domain is `android.bignerdranch.com`, as shown in Figure 7.7.

Figure 7.7 Creating the CriminalIntent application



Application name: CriminalIntent
Company Domain: android.bignerdranch.com
Package name: com.bignerdranch.android.criminalintent

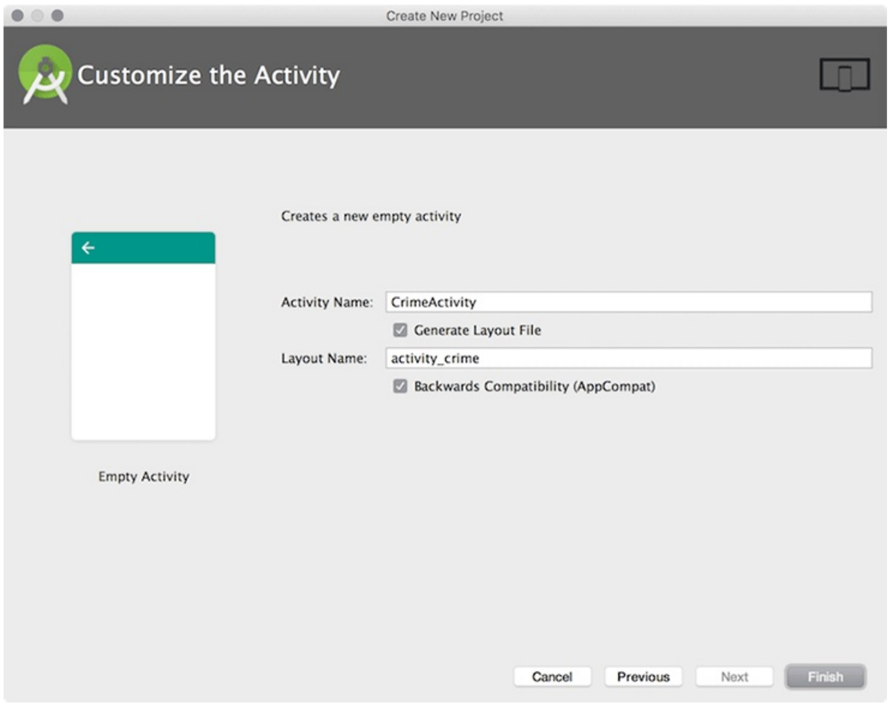
Project location: /Users/dev/AndroidStudioProjects/CriminalIntent

Click Next and specify a minimum SDK of API 19: Android 4.4. Also ensure that only the Phone and Tablet application type is checked.

Click Next again to select the type of activity to add. Choose Empty Activity and continue along in the wizard.

In the final step of the New Project wizard, name the activity **`CrimeActivity`** and click Finish (Figure 7.8).

Figure 7.8 Creating **`CrimeActivity`**



Creates a new empty activity

Activity Name: CrimeActivity

☑ Generate Layout File

Layout Name: activity_crime

☑ Backwards Compatibility (AppCompat)

Two types of fragments

Fragments were introduced in API level 11 along with the first Android tablets and the sudden need for UI flexibility. You must choose which implementation of fragments that you want use: native fragments or support fragments.

The native implementation of fragments is built into the device that the user runs your app on. If you support many different versions of Android, each of those Android versions could have a slightly different implementation of fragments (for example, a bug could be fixed in one version and not the versions prior to it). The support implementation of fragments is built into a library that you include in your application. This means that each device you run your app on will depend on the same implementation of fragments no matter the Android version.

In CriminalIntent, you will use the support implementation of fragments. Detailed reasoning for this decision is laid out at the end of the chapter in the section called *For the More Curious: Why Support Fragments Are Superior*.

Adding dependencies in Android Studio

You will use the implementation of fragments that comes with the *AppCompat* library. The AppCompat library is one of Google's many compatibility libraries that you will use throughout this book. You will learn much more about the AppCompat library in Chapter 13.

To use the AppCompat library, it must be included in your list of dependencies. Your project comes with two `build.gradle` files, one for the project as a whole and one for your app module. Open the `build.gradle` file located in your app module.

Listing 7.1 Gradle dependencies (`app/build.gradle`)

```
apply plugin: 'com.android.application'

android {
    ...
}

dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    ...
    compile 'com.android.support:appcompat-v7:25.0.1'
    ...
}
```

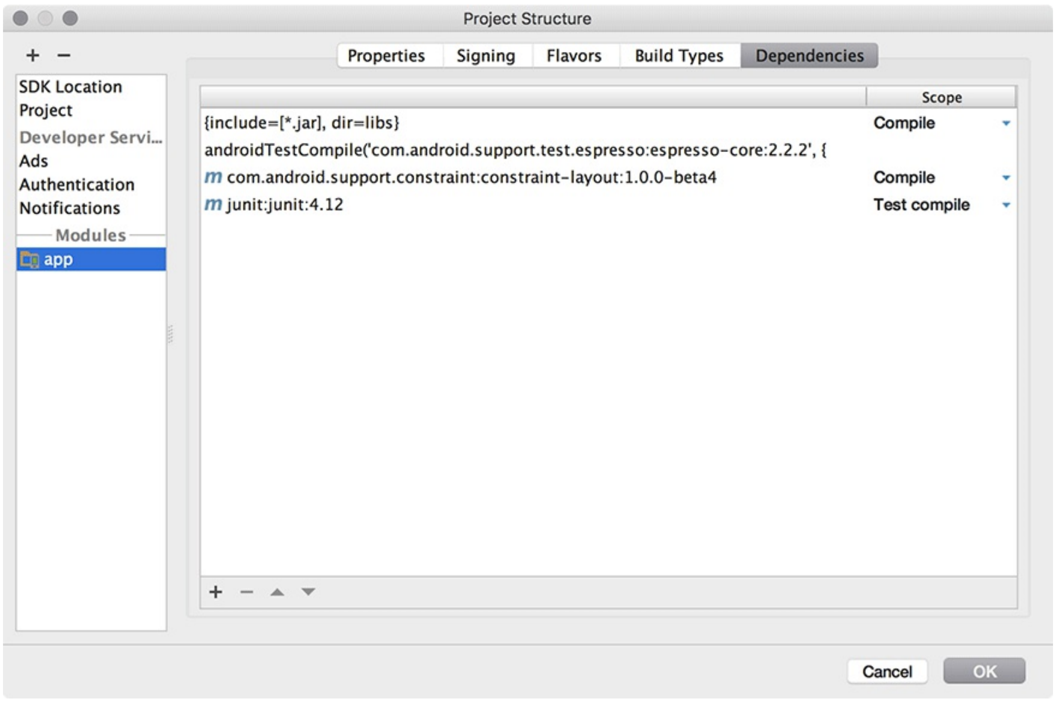
In the current dependencies section of your `build.gradle` file, you should see something similar to Listing 7.1 that specifies that the project depends on all of the `.jar` files in its `libs` directory. You will also see dependencies for other libraries that are automatically included when projects are created with Android Studio, most likely including the AppCompat library.

Gradle allows for the specification of dependencies that you have not copied into your project. When your app is compiled, Gradle will find, download, and include the dependencies for you. All you have to do is specify an exact string incantation and Gradle will do the rest.

If you do not have the AppCompat library listed in your dependencies, Android Studio has a tool to help you add the library and come up with this string incantation. Navigate to the project structure for your project (File → Project Structure...).

Select the app module on the left and the Dependencies tab in the app module. The dependencies for the app module are listed here (Figure 7.9).

Figure 7.9 App dependencies

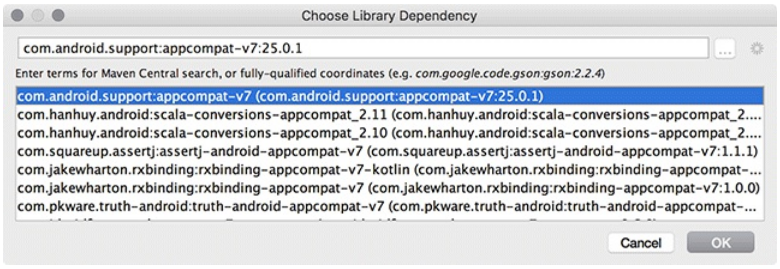


	Scope
{include=[*.jar], dir=libs}	Compile
androidTestCompile('com.android.support.test.espresso:espresso-core:2.2.2', {	
m com.android.support.constraint:constraint-layout:1.0.0-beta4	Compile
m junit:junit:4.12	Test compile

(You may have additional dependencies specified. If you do, do not remove them.)

You should see the AppCompat dependency listed. If you do not, add it with the + button and choose Library dependency. Choose the appcompat-v7 library from the list and click OK (Figure 7.10).

Figure 7.10 A collection of dependencies



```
com.android.support:appcompat-v7:25.0.1
Enter terms for Maven Central search, or fully-qualified coordinates (e.g. com.google.code.gson:gson:2.2.4)
com.android.support:appcompat-v7 (com.android.support:appcompat-v7:25.0.1)
com.hanhuy.android:scala-conversions-appcompat_2.11 (com.hanhuy.android:scala-conversions-appcompat_2....
com.hanhuy.android:scala-conversions-appcompat_2.10 (com.hanhuy.android:scala-conversions-appcompat_2....
com.squareup.assertj:assertj-android-appcompat-v7 (com.squareup.assertj:assertj-android-appcompat-v7:1.1.1)
com.jakewharton.rxbinding:rxbinding-appcompat-v7-kotlin (com.jakewharton.rxbinding:rxbinding-appcompat-...
com.jakewharton.rxbinding:rxbinding-appcompat-v7 (com.jakewharton.rxbinding:rxbinding-appcompat-v7:1.0.0)
com.pkware.truth-android:truth-android-appcompat-v7 (com.pkware.truth-android:truth-android-appcompat-...
```

Navigate back to the editor window showing `app/build.gradle`, and you should now see AppCompat included, as shown in Listing 7.1.

(If you modify this file manually, outside of the project structure window, you will need to sync your project with the Gradle file to reflect any updates that you have made. This sync asks Gradle to update the build based on your changes by either downloading or removing dependencies. Changes within the project structure window will trigger this sync automatically. To manually perform this sync, navigate to Tools → Android → Sync Project with Gradle Files.)

The dependency string `compile 'com.android.support:appcompat-v7:25.0.0'` uses the Maven coordinates format `groupId:artifactId:version`. (Maven is a dependency management tool. You can learn more about it at `maven.apache.org/`.)

The `groupId` is the unique identifier for a set of libraries available on the Maven repository. Often the library's base package name is used as the `groupId`, which is `com.android.support` for the AppCompat library.

The `artifactId` is the name of a specific library within the package. In this case, the name of the library you are referring to is `appcompat-v7`.

Last but not least, the `version` represents the revision number of the library. CriminalIntent depends on the 25.0.0 version of the appcompat-v7 library. Version 25.0.0 is the latest version as of this writing, but any version newer than that should also work for this project. In fact, it is a good idea to use the latest version of the support library so that you can use newer APIs and receive the latest bug fixes. If Android Studio added a newer version of the library for you, do not roll it back to the version shown above.

Now that the AppCompat library is a dependency in the project, make sure that your project uses it. In the project tool window, find and open `CrimeActivity.java`. Verify that **`CrimeActivity`**'s superclass is **`AppCompatActivity`**.

Listing 7.2 Tweaking template code (`CrimeActivity.java`)

```
public class CrimeActivity extends AppCompatActivity {

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

}
```

Before proceeding with **`CrimeActivity`**, let's create the model layer for CriminalIntent by writing the **`Crime`** class.

Creating the Crime class

In the project tool window, right-click the `com.bignerdranch.android.criminalintent` package and select New → Java Class. Name the class **`Crime`** and click OK.

In `Crime.java`, add fields to represent the crime's ID, title, date, and status and a constructor that initializes the ID and date fields (Listing 7.3).

Listing 7.3 Adding to **`Crime`** class (`Crime.java`)

```
public class Crime {

    private UUID mId;
    private String mTitle;
    private Date mDate;
    private boolean mSolved;

    public Crime() {
        mId = UUID.randomUUID();
        mDate = new Date();
    }
}
```

`UUID` is a Java utility class included in the Android framework. It provides an easy way to generate universally unique ID values. In the constructor you generate a random unique ID by calling `UUID.randomUUID()`.

Android Studio may find two classes with the name **`Date`**. Use the Option+Return (or Alt+Enter) shortcut to manually import the class. When asked which version of the **`Date`** class to import, choose the **`java.util.Date`** version.

Initializing the **`Date`** variable using the default **`Date`** constructor sets `mDate` to the current date. This will be the default date for a crime.

Next, you want to generate a getter for the read-only `mId` and both a getter and setter for `mTitle`, `mDate`, and `mSolved`. Right-click after the constructor and select Generate... → Getter and select the `mId` variable. Then, generate the getter and setter for `mTitle`, `mDate`, and `mSolved` by repeating the process, but selecting Getter and Setter in the Generate... menu.

Listing 7.4 Generated getters and setters (`Crime.java`)

```
public class Crime {

    private UUID mId;
    private String mTitle;
    private Date mDate;
    private boolean mSolved;

    public Crime() {
        mId = UUID.randomUUID();
        mDate = new Date();
    }

    public UUID getId() {
        return mId;
    }

    public String getTitle() {
        return mTitle;
    }

    public void setTitle(String title) {
        mTitle = title;
    }

    public Date getDate() {
        return mDate;
    }

    public void setDate(Date date) {
        mDate = date;
    }

    public boolean isSolved() {
        return mSolved;
    }

    public void setSolved(boolean solved) {
        mSolved = solved;
    }
}
```

That is all you need for the **`Crime`** class and for CriminalIntent's model layer in this chapter.

At this point, you have created the model layer and an activity that is capable of hosting a support fragment. Now you will get into the details of how the activity performs its duties as host.

Hosting a UI Fragment

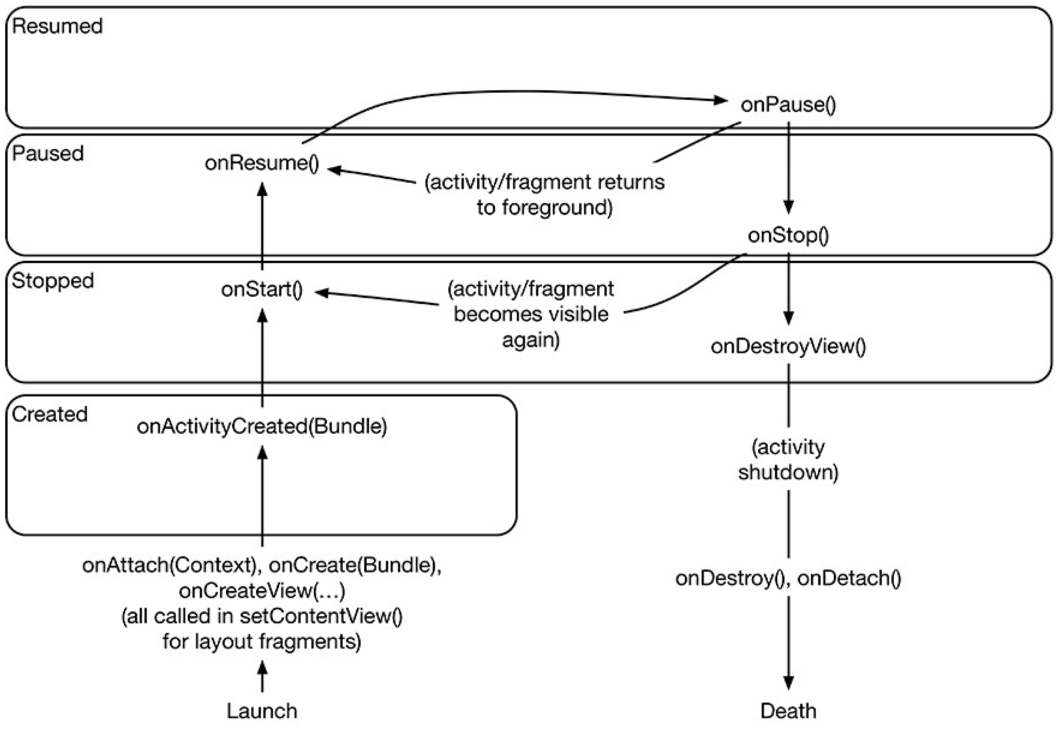
To host a UI fragment, an activity must:

- define a spot in its layout for the fragment's view
- manage the lifecycle of the fragment instance

The fragment lifecycle

Figure 7.11 shows the fragment lifecycle. It is similar to the activity lifecycle: It has stopped, paused, and resumed states, and it has methods you can override to get things done at critical points – many of which correspond to activity lifecycle methods.

Figure 7.11 Fragment lifecycle diagram


Resumed
onPause()
Paused
onResume()
(activity/fragment returns to foreground)
onStop()
Stopped
onStart()
(activity/fragment becomes visible again)
onDestroyView()
Created
onActivityCreated(Bundle)
(activity shutdown)
onAttach(Context), onCreate(Bundle), onCreateView(…) (all called in setContentView() for layout fragments)
onDestroy(), onDetach()
Launch
Death

The correspondence is important. Because a fragment works on behalf of an activity, its state should reflect the activity's state. Thus, it needs corresponding lifecycle methods to handle the activity's work.

One critical difference between the fragment lifecycle and the activity lifecycle is that fragment lifecycle methods are called by the hosting activity, not the OS. The OS knows nothing about the fragments that an activity is using to manage things. Fragments are the activity's internal business.

You will see more of the fragment lifecycle methods as you continue building CriminalIntent.

Two approaches to hosting

You have two options when it comes to hosting a UI fragment in an activity:

- add the fragment to the activity's *layout*
- add the fragment in the activity's *code*

The first approach is known as using a *layout fragment*. It is straightforward but inflexible. If you add the fragment to the activity's layout, you hardwire the fragment and its view to the activity's view and cannot swap out that fragment during the activity's lifetime.

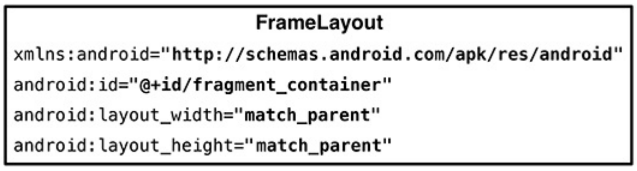
The second approach, adding the fragment to the activity's code, is more complex – but it is the only way to have control at runtime over your fragments. You determine when the fragment is added to the activity and what happens to it after that. You can remove the fragment, replace it with another, and then add the first fragment back again.

Thus, to achieve real UI flexibility you must add your fragment in code. This is the approach you will use for **CrimeActivity**'s hosting of a **CrimeFragment**. The code details will come later in the chapter. First, you are going to define **CrimeActivity**'s layout.

Defining a container view

You will be adding a UI fragment in the hosting activity's code, but you still need to make a spot for the fragment's view in the activity's view hierarchy. In **CrimeActivity**'s layout, this spot will be the **FrameLayout** shown in Figure 7.12.

Figure 7.12 Fragment-hosting layout for **CrimeActivity**



FrameLayout

xmlns:android="**http://schemas.android.com/apk/res/android**"
android:id="**@+id/fragment_container**"
android:layout_width="**match_parent**"
android:layout_height="**match_parent**"

This **FrameLayout** will be the *container view* for a **CrimeFragment**. Notice that the container view is completely generic; it does not name the **CrimeFragment** class. You can and will use this same layout to host other fragments.

Locate **CrimeActivity**'s layout at `res/layout/activity_crime.xml`. Open this file and replace the default layout with the **FrameLayout** diagrammed in Figure 7.12. Your XML should match Listing 7.5.

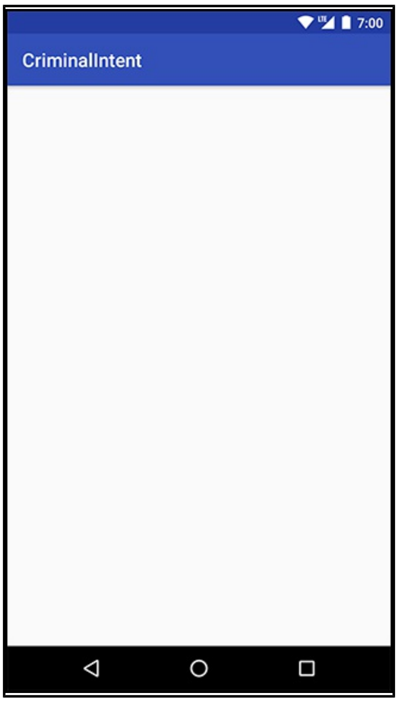
Listing 7.5 Creating the fragment container layout (`activity_crime.xml`)

```
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/fragment_container"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Note that while `activity_crime.xml` consists solely of a container view for a single fragment, an activity's layout can be more complex and define multiple container views as well as widgets of its own.

You can preview your layout file or run CriminalIntent to check your code. You will see an empty **`FrameLayout`** below a toolbar containing the text CriminalIntent (Figure 7.13). (If the preview window does not render the screen correctly, or you see errors, build the project by selecting Build → Rebuild Project. If that still does not work correctly, run the app on your emulator or device. As of this writing, the preview window can be finicky.)

Figure 7.13 An empty **`FrameLayout`**



The **`FrameLayout`** is empty because the **`CrimeActivity`** is not yet hosting a fragment. Later, you will write code that puts a fragment's view inside this **`FrameLayout`**. But first, you need to create a fragment.

(The toolbar at the top of your app is included automatically because of the way you configured your activity. You will learn more about the toolbar in Chapter 13.)

Creating a UI Fragment

The steps to create a UI fragment are the same as those you followed to create an activity:

- compose a UI by defining widgets in a layout file
- create the class and set its view to be the layout that you defined
- wire up the widgets inflated from the layout in code

Defining CrimeFragment's layout

CrimeFragment's view will display the information contained within an instance of **Crime**.

First, define the strings that the user will see in `res/values/strings.xml`.

Listing 7.6 Adding strings (`res/values/strings.xml`)

```
<resources>
    <string name="app_name">CriminalIntent</string>
    <string name="crime_title_hint">Enter a title for the crime.</string>
    <string name="crime_title_label">Title</string>
    <string name="crime_details_label">Details</string>
    <string name="crime_solved_label">Solved</string>
</resources>
```

Next, you will define the UI. The layout for **CrimeFragment** will consist of a vertical **LinearLayout** that contains two **TextView**s, an **EditText**, a **Button**, and a **Checkbox**.

To create a layout file, right-click the `res/layout` folder in the project tool window and select New → Layout resource file. Name this file `fragment_crime.xml` and enter **LinearLayout** as the root element. Click OK and Android Studio will generate the file for you.

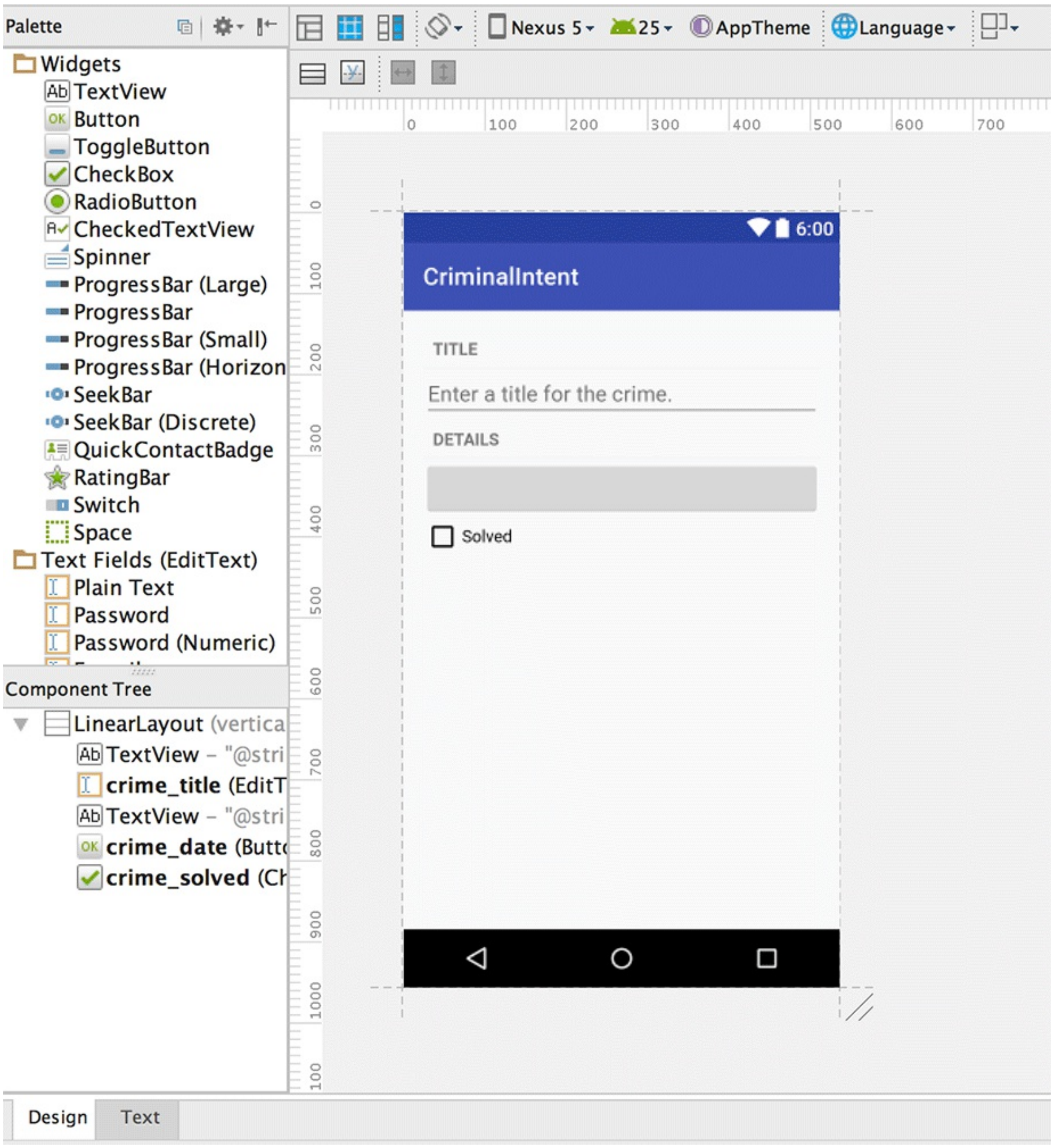
When the file opens, navigate to the XML. The wizard has added the **LinearLayout** for you. Add the widgets that make up the fragment's layout to fragment_crime.xml.

Listing 7.7 Layout file for fragment's view (fragment_crime.xml)

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_margin="16dp"
    android:orientation="vertical">

    <TextView
        style="?android:listSeparatorTextViewStyle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_title_label"/>

    <EditText
        android:id="@+id/crime_title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/crime_title_hint"/>

    <TextView
        style="?android:listSeparatorTextViewStyle"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_details_label"/>

    <Button
        android:id="@+id/crime_date"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <CheckBox
        android:id="@+id/crime_solved"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/crime_solved_label"/>

</LinearLayout>
```

Check the Design view to see a preview of your fragment’s view (Figure 7.14).

Figure 7.14 Previewing updated crime fragment layout



(The updated `fragment_crime.xml` code includes new syntax related to view style: `<style="?android:listSeparatorTextViewStyle"`. Fear not. You will learn the meaning behind this syntax in the section called *Styles, themes, and theme attributes* in Chapter 9.)

Creating the CrimeFragment class

Right-click the `com.bignerdranch.android.criminalintent` package and select New → Java Class. Name the class **`CrimeFragment`** and click OK to generate the class.

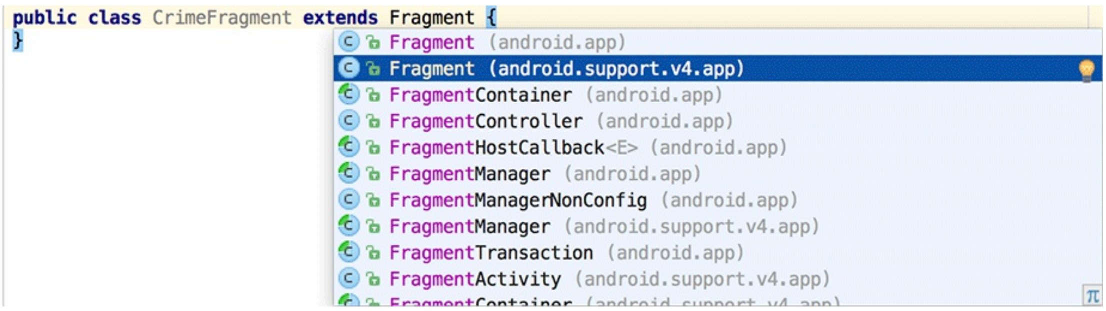
Now, turn this class into a fragment. Update **`CrimeFragment`** to subclass the **`Fragment`** class.

Listing 7.8 Subclassing the **`Fragment`** class (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {

}
```

As you subclass the **`Fragment`** class, you will notice that Android Studio finds two classes with the **`Fragment`** name. You will see **`Fragment (android.app)`** and **`Fragment (android.support.v4.app)`**. The android.app **`Fragment`** is the version of fragments built into the Android OS. You will use the support library version, so be sure to select the android.support.v4.app version of the **`Fragment`** class when you see the dialog, as shown in Figure 7.15.

Figure 7.15 Choosing the support library's **`Fragment`** class



```
public class CrimeFragment extends Fragment {
}
```

Fragment (android.app)
Fragment (android.support.v4.app)
FragmentContainer (android.app)
FragmentController (android.app)
FragmentHostCallback<E> (android.app)
FragmentManager (android.app)
FragmentManagerNonConfig (android.app)
FragmentManager (android.support.v4.app)
FragmentTransaction (android.app)
FragmentActivity (android.support.v4.app)
π

Your code should match Listing 7.9.

Listing 7.9 Supporting the **`Fragment`** import (`CrimeFragment.java`)

```
package com.bignerdranch.android.criminalintent;

import android.support.v4.app.Fragment;

public class CrimeFragment extends Fragment {

}
```

If you do not see this dialog or the wrong fragment class was imported, you can manually import the correct class. If you have an import for **`android.app.Fragment`**, remove that line of code. Import the correct **`Fragment`** class with the Option+Return (or Alt+Enter) shortcut. Be sure to select the support version of the **`Fragment`** class.

Implementing fragment lifecycle methods

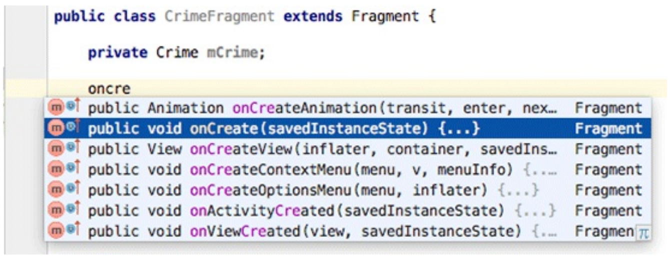
`CrimeFragment` is a controller that interacts with model and view objects. Its job is to present the details of a specific crime and update those details as the user changes them.

In GeoQuiz, your activities did most of their controller work in activity lifecycle methods. In CriminalIntent, this work will be done by fragments in fragment lifecycle methods. Many of these methods correspond to the **`Activity`** methods you already know, such as **`onCreate(Bundle)`**.

In `CrimeFragment.java`, add a member variable for the **`Crime`** instance and an implementation of **`Fragment.onCreate(Bundle)`**.

Android Studio can provide some assistance when overriding methods. As you define the **`onCreate(Bundle)`** method, type the first few characters of the method name where you want to place the method. Android Studio will provide a list of suggestions, as shown in Figure 7.16.

Figure 7.16 Overriding the **`onCreate(Bundle)`** method



```
public class CrimeFragment extends Fragment {

    private Crime mCrime;

    oncre
public Animation onCreateAnimation(transit, enter, nex…   Fragment
public void onCreate(savedInstanceState) {...}           Fragment
public View onCreateView(inflater, container, savedIns…  Fragment
public void onCreateContextMenu(menu, v, menuInfo) {...  Fragment
public void onCreateOptionsMenu(menu, inflater) {...}    Fragment
public void onActivityCreated(savedInstanceState) {...}  Fragment
public void onViewCreated(view, savedInstanceState) {…   Fragmen
```

Press Return to select the **`onCreate(Bundle)`** method, and Android Studio will create the method declaration for you. Update your code to create a new **`Crime`**, matching Listing 7.10.

Listing 7.10 Overriding **`Fragment.onCreate(Bundle)`** (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    private Crime mCrime;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCrime = new Crime();
    }
}
```

There are a couple of things to notice in this implementation. First, **`Fragment.onCreate(Bundle)`** is a public method, whereas **`Activity.onCreate(Bundle)`** is protected. **`Fragment.onCreate(Bundle)`** and other **`Fragment`** lifecycle methods must be public, because they will be called by whatever activity is hosting the fragment.

Second, similar to an activity, a fragment has a bundle to which it saves and retrieves its state. You can override **`Fragment.onSaveInstanceState(Bundle)`** for your own purposes just as you can override **`Activity.onSaveInstanceState(Bundle)`**.

Also, note what does *not* happen in **`Fragment.onCreate(Bundle)`**: You do not inflate the fragment's view. You configure the fragment instance in **`Fragment.onCreate(Bundle)`**, but you create and configure the fragment's view in another fragment lifecycle method:

```
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState)
```

This method is where you inflate the layout for the fragment's view and return the inflated **`View`** to the hosting activity. The **`LayoutInflater`** and **`ViewGroup`** parameters are necessary to inflate the layout. The **`Bundle`** will contain data that this method can use to re-create the view from a saved state.

In `CrimeFragment.java`, add an implementation of **`onCreateView(…)`** that inflates `fragment_crime.xml`. You can use the same trick from Figure 7.16 to fill out the method declaration.

Listing 7.11 Overriding **`onCreateView(…)`** (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    private Crime mCrime;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCrime = new Crime();
    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_crime, container, false);
        return v;
    }
}
```

Within **`onCreateView(…)`**, you explicitly inflate the fragment's view by calling **`LayoutInflater.inflate(…)`** and passing in the layout resource ID. The second parameter is your view's parent, which is usually needed to configure the widgets properly. The third parameter tells the layout inflater whether to add the inflated view to the view's parent. You pass in `false` because you will add the view in the activity's code.

Wiring widgets in a fragment

You are now going to hook up the **EditText**, **Checkbox**, and **Button** in your fragment. The **onCreateView(…)** method is the place to wire up these widgets.

Start with the **EditText**. After the view is inflated, get a reference to the **EditText** and add a listener.

Listing 7.12 Wiring up the **EditText** widget (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    private Crime mCrime;
    private EditText mTitleField;
    ...
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_crime, container, false);

        mTitleField = (EditText) v.findViewById(R.id.crime_title);
        mTitleField.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(
                CharSequence s, int start, int count, int after) {
                // This space intentionally left blank
            }

            @Override
            public void onTextChanged(
                CharSequence s, int start, int before, int count) {
                mCrime.setTitle(s.toString());
            }

            @Override
            public void afterTextChanged(Editable s) {
                // This one too
            }
        });

        return v;
    }
}
```

Getting references in **Fragment.onCreateView(…)** works nearly the same as in **Activity.onCreate(Bundle)**. The only difference is that you call **View.findViewById(int)** on the fragment's view. The **Activity.findViewById(int)** method that you used before is a convenience method that calls **View.findViewById(int)** behind the scenes. The **Fragment** class does not have a corresponding convenience method, so you have to call the real thing.

Setting listeners in a fragment works exactly the same as in an activity. In Listing 7.12, you create an anonymous class that implements the verbose **TextWatcher** interface. **TextWatcher** has three methods, but you only care about one: **onTextChanged(…)**.

In **onTextChanged(…)**, you call **toString()** on the **CharSequence** that is the user's input. This method returns a string, which you then use to set the **Crime**'s title.

Next, connect the **Button** to display the date of the crime, as shown in Listing 7.13.

Listing 7.13 Setting **Button** text (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    private Crime mCrime;
    private EditText mTitleField;
    private Button mDateButton;
    ...
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_crime, container, false);
        ...
        mDateButton = (Button) v.findViewById(R.id.crime_date);
        mDateButton.setText(mCrime.getDate().toString());
        mDateButton.setEnabled(false);

        return v;
    }
}
```

Disabling the button ensures that it will not respond in any way to the user pressing it. It also changes its appearance to advertise its disabled state. In Chapter 12, you will enable the button and allow the user to choose the date of the crime.

Moving on to the **CheckBox**, get a reference and set a listener that will update the `mSolved` field of the **Crime**, as shown in Listing 7.14.

Listing 7.14 Listening for **CheckBox** changes (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    private Crime mCrime;
    private EditText mTitleField;
    private Button mDateButton;
    private CheckBox mSolvedCheckBox;
    ...
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
            Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_crime, container, false);
        ...
        mSolvedCheckBox = (CheckBox)v.findViewById(R.id.crime_solved);
        mSolvedCheckBox.setOnCheckedChangeListener(new OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(CompoundButton buttonView,
                    boolean isChecked) {
                mCrime.setSolved(isChecked);
            }
        });

        return v;
    }
}
```

After typing in the code as above, click on **`OnCheckedChangeListener`**:

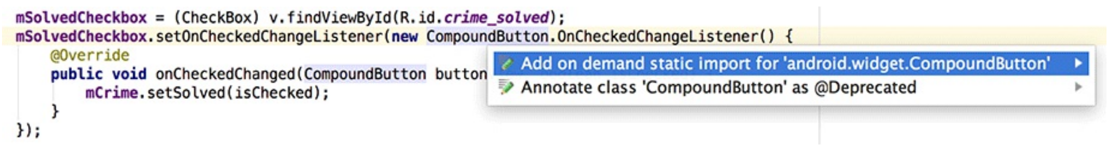
```
mSolvedCheckBox.setOnCheckedChangeListener(new OnCheckedChangeListener()
```

and use the Option+Return (Alt+Enter) shortcut to add the necessary import statement. You will be presented with two options. Choose the **`android.widget.CompoundButton`** version.

Depending on which version of Android Studio you are using, the autocomplete feature may insert `CompoundButton.OnCheckedChangeListener()` instead of leaving the code as `OnCheckedChangeListener()`. Either implementation is fine. But to remain consistent with the solution presented in this book, click on `CompoundButton` and hit Option+Return (Alt+Enter).

Select the option to Add on demand static import for 'android.widget.CompoundButton' (Figure 7.17). This will update the code so it matches Listing 7.14.

Figure 7.17 Adding on demand static import



```
mSolvedCheckbox = (CheckBox) v.findViewById(R.id.crime_solved);
mSolvedCheckbox.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
    @Override
    public void onCheckedChanged(CompoundButton button
        mCrime.setSolved(isChecked);
    }
});
```

Add on demand static import for 'android.widget.CompoundButton'
Annotate class 'CompoundButton' as @Deprecated

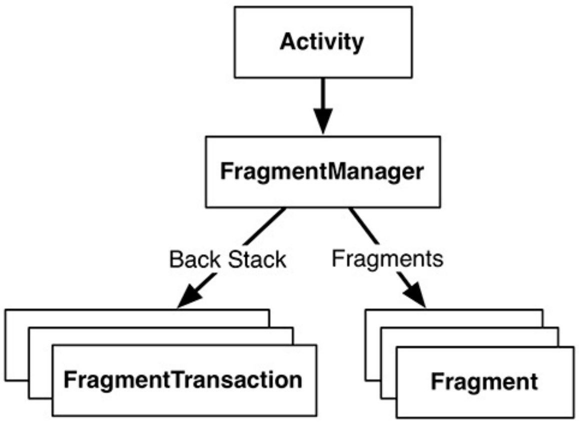
Your code for **`CrimeFragment`** is now complete. It would be great if you could run CriminalIntent now and play with the code you have written. But you cannot. Fragments cannot put their views on screen on their own. To realize your efforts, you first have to add a **`CrimeFragment`** to **`CrimeActivity`**.

Adding a UI Fragment to the FragmentManager

When the **Fragment** class was introduced in Honeycomb, the **Activity** class was changed to include a piece called the **FragmentManager**. The **FragmentManager** is responsible for managing your fragments and adding their views to the activity's view hierarchy (Figure 7.18).

The **FragmentManager** handles two things: a list of fragments and a back stack of fragment transactions (which you will learn about shortly).

Figure 7.18 The **FragmentManager**



For CriminalIntent, you will only be concerned with the **FragmentManager**'s list of fragments.

To add a fragment to an activity in code, you make explicit calls to the activity's **FragmentManager**. The first step is to get the **FragmentManager** itself. Do so in **onCreate(Bundle)** in `CrimeActivity.java`.

Listing 7.15 Getting the **FragmentManager** (`CrimeActivity.java`)

```
public class CrimeActivity extends AppCompatActivity {

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

        FragmentManager fm = getSupportFragmentManager();
    }
}
```

If you see an error after adding this line of code, check the import statements to make sure that the support version of the **FragmentManager** class was imported.

You call **getSupportFragmentManager()** because you are using the support library and the **AppCompatActivity** class. If you were not interested in using the support library, then you would subclass **Activity** and call **getFragmentManager()**.

Fragment transactions

Now that you have the **FragmentManager**, add the following code to give it a fragment to manage. (We will step through this code afterward. Just get it in for now.)

Listing 7.16 Adding a **CrimeFragment** (CrimeActivity.java)

```
public class CrimeActivity extends AppCompatActivity {

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

        FragmentManager fm = getSupportFragmentManager();
        Fragment fragment = fm.findFragmentById(R.id.fragment_container);

        if (fragment == null) {
            fragment = new CrimeFragment();
            fm.beginTransaction()
                .add(R.id.fragment_container, fragment)
                .commit();
        }
    }
}
```

The best place to start understanding the code you just added is not at the beginning. Instead, find the **add(…)** operation and the code around it. This code creates and commits a *fragment transaction*.

```
        if (fragment == null) {
            fragment = new CrimeFragment();
            fm.beginTransaction()
                .add(R.id.fragment_container, fragment)
                .commit();
```

Fragment transactions are used to add, remove, attach, detach, or replace fragments in the fragment list. They are the heart of how you use fragments to compose and recompose screens at runtime. The **FragmentManager** maintains a back stack of fragment transactions that you can navigate.

The **FragmentManager.beginTransaction()** method creates and returns an instance of **FragmentTransaction**. The **FragmentTransaction** class uses a *fluent interface* – methods that configure **FragmentTransaction** return a **FragmentTransaction** instead of `void`, which allows you to chain them together. So the code highlighted above says, “Create a new fragment transaction, include one add operation in it, and then commit it.”

The **add(…)** method is the meat of the transaction. It has two parameters: a container view ID and the newly created **CrimeFragment**. The container view ID should look familiar. It is the resource ID of the **FrameLayout** that you defined in `activity_crime.xml`.

A container view ID serves two purposes:

- It tells the **FragmentManager** where in the activity's view the fragment's view should appear.
- It is used as a unique identifier for a fragment in the **FragmentManager**'s list.

When you need to retrieve the **CrimeFragment** from the **FragmentManager**, you ask for it by container view ID:

```
FragmentManager fm = getSupportFragmentManager();
Fragment fragment = fm.findFragmentById(R.id.fragment_container);

if (fragment == null) {
    fragment = new CrimeFragment();
    fm.beginTransaction()
        .add(R.id.fragment_container, fragment)
        .commit();
}
```

It may seem odd that the **FragmentManager** identifies the **CrimeFragment** using the resource ID of a **FrameLayout**. But identifying a UI fragment by the resource ID of its container view is built into how the **FragmentManager** operates. If you are adding multiple fragments to an activity, you would typically create separate containers with separate IDs for each of those fragments.

Now we can summarize the code you added in Listing 7.16 from start to finish.

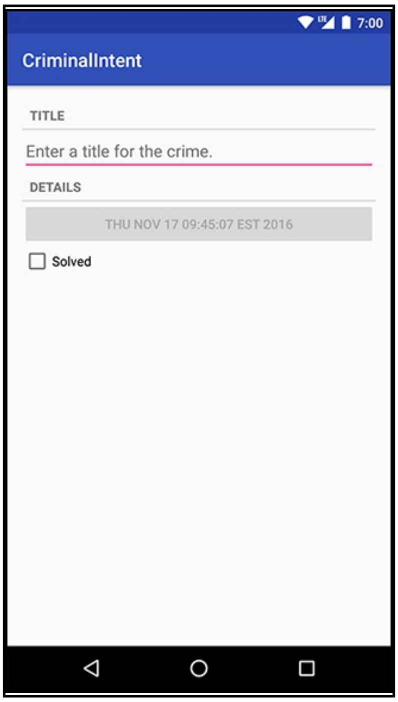
First, you ask the **FragmentManager** for the fragment with a container view ID of `R.id.fragment_container`. If this fragment is already in the list, the **FragmentManager** will return it.

Why would a fragment already be in the list? The call to **CrimeActivity.onCreate(Bundle)** could be in response to **CrimeActivity** being *re-created* after being destroyed on rotation or to reclaim memory. When an activity is destroyed, its **FragmentManager** saves out its list of fragments. When the activity is re-created, the new **FragmentManager** retrieves the list and re-creates the listed fragments to make everything as it was before.

On the other hand, if there is no fragment with the given container view ID, then `fragment` will be null. In this case, you create a new **CrimeFragment** and a new fragment transaction that adds the fragment to the list.

CrimeActivity is now hosting a **CrimeFragment**. Run CriminalIntent to prove it. You should see the view defined in `fragment_crime.xml`, as shown in Figure 7.19.

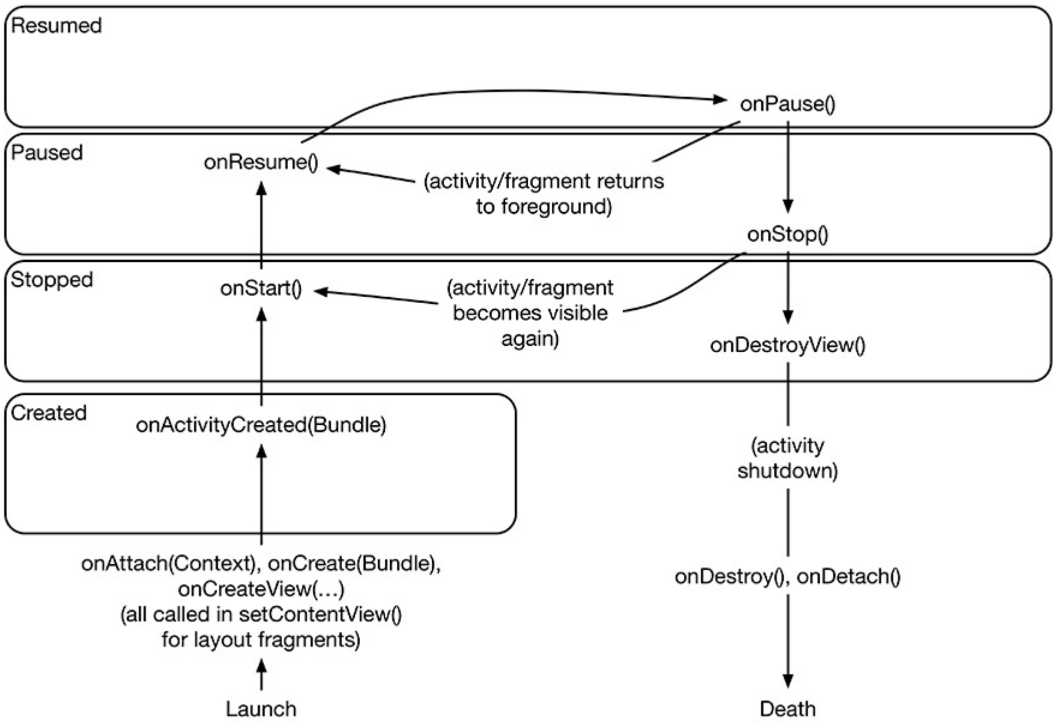
Figure 7.19 **CrimeFragment**'s view hosted by **CrimeActivity**



The FragmentManager and the fragment lifecycle

Now that you know about the **`FragmentManager`**, let's take another look at the fragment lifecycle (Figure 7.20).

Figure 7.20 The fragment lifecycle, again


Resumed
onPause()
Paused
onResume()
(activity/fragment returns to foreground)
onStop()
Stopped
onStart()
(activity/fragment becomes visible again)
onDestroyView()
Created
onActivityCreated(Bundle)
(activity shutdown)
onAttach(Context), onCreate(Bundle), onCreateView(…) (all called in setContentView() for layout fragments)
onDestroy(), onDetach()
Launch
Death

The **`FragmentManager`** of an activity is responsible for calling the lifecycle methods of the fragments in its list. The **`onAttach(Context)`**, **`onCreate(Bundle)`**, and **`onCreateView(…)`** methods are called when you add the fragment to the **`FragmentManager`**.

The **`onActivityCreated(Bundle)`** method is called after the hosting activity's **`onCreate(Bundle)`** method has executed. You are adding the **`CrimeFragment`** in **`CrimeActivity.onCreate(Bundle)`**, so this method will be called after the fragment has been added.

What happens if you add a fragment while the activity is already resumed? In that case, the **`FragmentManager`** immediately walks the fragment through whatever steps are necessary to get it caught up to the activity's state. For example, as a fragment is added to an activity that is already resumed, that fragment gets calls to **`onAttach(Context)`**, **`onCreate(Bundle)`**, **`onCreateView(…)`**, **`onActivityCreated(Bundle)`**, **`onStart()`**, and then **`onResume()`**.

Once the fragment's state is caught up to the activity's state, the hosting activity's **`FragmentManager`** will call further lifecycle methods around the same time it receives the corresponding calls from the OS to keep the fragment's state aligned with that of the activity.

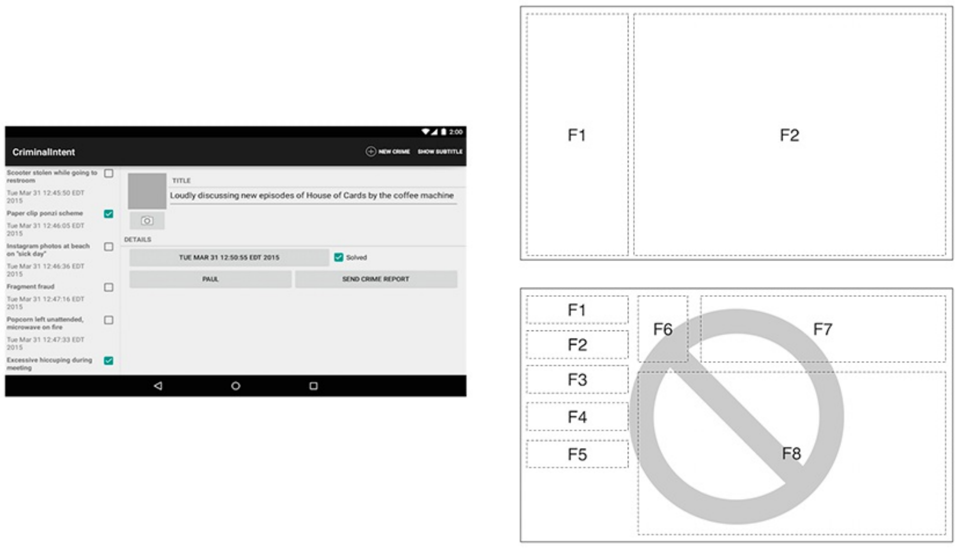
Application Architecture with Fragments

Designing your app with fragments the right way is supremely important. Many developers, after first learning about fragments, try to use them for every reusable component in their application. This is the wrong way to use fragments.

Fragments are intended to encapsulate major components in a reusable way. A major component in this case would be on the level of an entire screen of your application. If you have a significant number of fragments on screen at once, your code will be littered with fragment transactions and unclear responsibility. A better architectural solution for reuse with smaller components is to extract them into a custom view (a class that subclasses **`View`** or one of its subclasses).

Use fragments responsibly. A good rule of thumb is to have no more than two or three fragments on the screen at a time (Figure 7.21).

Figure 7.21 Less is more



The reason all our activities will use fragments

From here on, all of the apps in this book will use fragments – no matter how simple. This may seem like overkill. Many of the examples you will see in following chapters could be written without fragments. The UIs could be created and managed from activities, and doing so might even be less code.

However, we believe it is better for you to become comfortable with the pattern you will most likely use in real life.

You might think it would be better to begin a simple app without fragments and add them later, when (or if) necessary. There is an idea in Extreme Programming methodology called YAGNI. YAGNI stands for “You Aren’t Gonna Need It,” and it urges you not to write code if you think you *might* need it later. Why? Because YAGNI. It is tempting to say “YAGNI” to fragments.

Unfortunately, adding fragments later can be a minefield. Changing an activity to an activity hosting a UI fragment is not difficult, but there are swarms of annoying gotchas. Keeping some interfaces managed by activities and having others managed by fragments only makes things worse because you have to keep track of this meaningless distinction. It is far easier to write your code using fragments from the beginning and not worry about the pain and annoyance of reworking it later, or having to remember which style of controller you are using in each part of your application.

Therefore, when it comes to fragments, we have a different principle: AUF, or “Always Use Fragments.” You can kill a lot of brain cells deciding whether to use a fragment or an activity, and it is just not worth it. AUF!

For the More Curious: Fragments and the Support Library

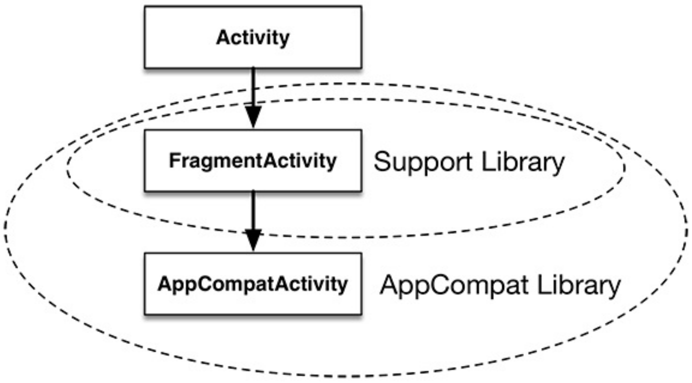
In this chapter, you included the AppCompat library so that you can use support fragments. AppCompat on its own does not include a support fragment implementation. The AppCompat library depends on the support-v4 library, which is where the support fragment implementation lives.

Google provides many different support libraries, including support-v4, appcompat-v7, recyclerview-v7, and many more. The support-v4 library is typically referred to as *the* support library. This was the first support library Google provided to developers. Over time, more and more tools have been added to this library, and it became a grab bag of things with no real focus. At that point, Google decided to develop a suite of support libraries rather than a single library.

The support library (support-v4) contains the support implementation of fragments that you used in this chapter. For example, this is where you will find the source of **`android.support.v4.app.Fragment`**. The support library also includes an **`Activity`** subclass: **`FragmentActivity`**. To use support fragments, your activities must subclass **`FragmentActivity`**.

As shown in Figure 7.22, **`AppCompatActivity`** is a subclass of this **`FragmentActivity`** class, which is how you were able to use support fragments in this chapter. If you were using support fragments without the AppCompat library, you would include the support-v4 dependency in your project and you would subclass **`FragmentActivity`** in each of your activity classes instead of **`AppCompatActivity`**.

Figure 7.22 AppCompatActivity class hierarchy



If all of this sounds confusing, that is because it is. But not to worry – most Android developers use these libraries as you did in this chapter: using the AppCompat library rather than the support library directly. You will learn more about the features of the AppCompat library in Chapter 13.

For the More Curious: Why Support Fragments Are Superior

This book uses the support library implementation of fragments over the implementation built into the Android OS, which may seem like an unusual choice. After all, the support library implementation of fragments was initially created so that developers could use fragments on old versions of Android that do not support the API. Today, most developers can exclusively work with versions of Android that include support for fragments natively.

We still prefer support fragments. Why? Support fragments are superior because you can update the version of the support library in your application and ship a new version of your app at any time. New releases of the support library come out multiple times a year. When a new feature is added to the fragment API, that feature is also added to the support library fragment API along with any available bug fixes. To use this new goodness, just update the version of the support library in your application.

As an example, official support for fragment nesting (hosting a fragment in a fragment) was added in Android 4.2. If you are using the Android OS implementation of fragments and supporting Android 4.0 and newer, you cannot use this API on all devices that your app supports. If you are using the support library, you can update the version of the library in your app and nest fragments until you run out of memory on the device.

There are no significant downsides to using the support library's fragments. The implementation of fragments is nearly identical in the support library as it is in the OS. The only real downside is that you have to include the support library in your project, and it has a nonzero size. However, it is currently under a megabyte – and you will likely use the support library for some of its other features as well.

We take a practical approach in this book and in our own application development. The support library is king.

If you are strong-willed and do not believe in the advice above, you can use the fragment implementation built into the Android OS.

To use standard library fragments, you would make three changes to the project:

- Subclass the standard library **`Activity`** class (`android.app.Activity`) instead of **`FragmentActivity`** or **`AppCompatActivity`**. Activities have support for fragments out of the box on API level 11 or higher.
- Subclass `android.app.Fragment` instead of `android.support.v4.app.Fragment`.
- To get the `FragmentManager`, call **`getFragmentManager()`** instead of **`getSupportFragmentManager()`**.