8

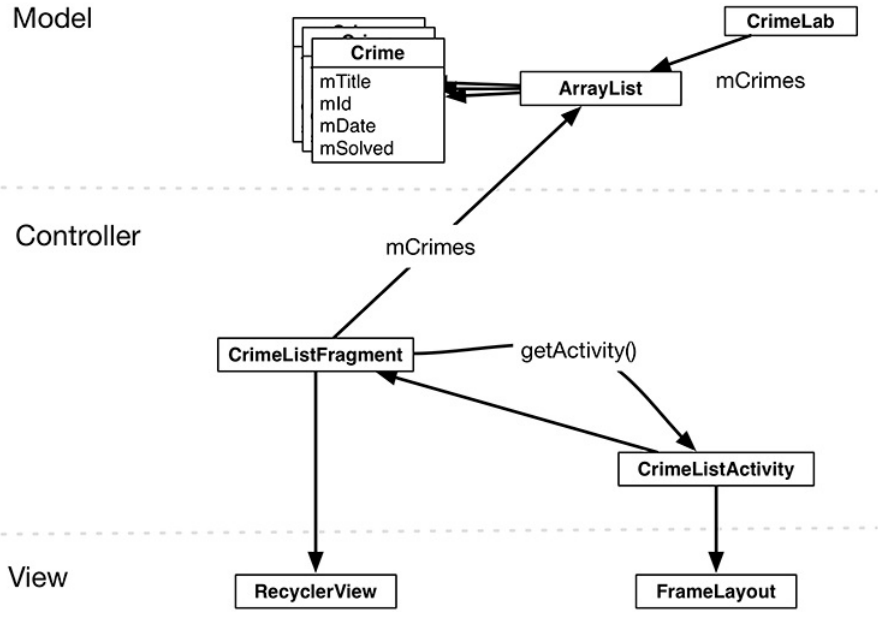
Displaying Lists with RecyclerView

CriminalIntent's model layer currently consists of a single instance of **`Crime`**. In this chapter, you will update CriminalIntent to work with a list of crimes. The list will display each **`Crime`**'s title and date, as shown in Figure 8.1.

Figure 8.1 A list of crimes

Figure 8.2 shows the overall plan for CriminalIntent in this chapter.

Figure 8.2 CriminalIntent with a list of crimes



In the model layer, you have a new object, **`CrimeLab`**, that will be a centralized data stash for **`Crime`** objects.

Displaying a list of crimes requires a new activity and a new fragment in CriminalIntent's controller layer: **`CrimeListActivity`** and **`CrimeListFragment`**.

(Where are **`CrimeActivity`** and **`CrimeFragment`** in Figure 8.2? They are part of the detail view, so we are not showing them here. In Chapter 10, you will connect the list and the detail parts of CriminalIntent.)

In Figure 8.2, you can also see the view objects associated with **`CrimeListActivity`** and **`CrimeListFragment`**. The activity's view will consist of a fragment-containing **`FrameLayout`**. The fragment's view will consist of a **`RecyclerView`**. You will learn more about the **`RecyclerView`** class later in the chapter.

Updating CriminalIntent's Model Layer

The first step is to upgrade CriminalIntent's model layer from a single **`Crime`** object to a **`List`** of **`Crime`** objects.

Singletons and centralized data storage

You are going to store the **`List`** of crimes in a *singleton*. A singleton is a class that allows only one instance of itself to be created.

A singleton exists as long as the application stays in memory, so storing the list in a singleton will keep the crime data available throughout any lifecycle changes in your activities and fragments. Be careful with singleton classes, as they will be destroyed when Android removes your application from memory. The **`CrimeLab`** singleton is not a solution for long-term storage of data, but it does allow the app to have one owner of the crime data and provides a way to easily pass that data between controller classes. (You will learn more about long-term data storage in Chapter 14.)

(See the For the More Curious section at the end of this chapter for more about singleton classes.)

To create a singleton, you create a class with a private constructor and a **`get()`** method. If the instance already exists, then **`get()`** simply returns the instance. If the instance does not exist yet, then **`get()`** will call the constructor to create it.

Right-click the `com.bignerdranch.android.criminalintent` package and choose New → Java Class. Name this class **`CrimeLab`** and click OK.

In `CrimeLab.java`, implement **`CrimeLab`** as a singleton with a private constructor and a **`get()`** method.

Listing 8.1 Setting up the singleton (`CrimeLab.java`)

```
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    public static CrimeLab get(Context context) {
        if (sCrimeLab == null) {
            sCrimeLab = new CrimeLab(context);
        }
        return sCrimeLab;
    }

    private CrimeLab(Context context) {

    }
}
```

There are a few interesting things in this **`CrimeLab`** implementation. First, notice the `s` prefix on the `sCrimeLab` variable. You are using this Android convention to make it clear that `sCrimeLab` is a static variable.

Also, notice the private constructor on the **`CrimeLab`**. Other classes will not be able to create a **`CrimeLab`**, bypassing the **`get()`** method.

Finally, in the **`get()`** method on **`CrimeLab`**, you pass in a **`Context`** object. You will make use of this **`Context`** object in Chapter 14.

Let's give **CrimeLab** some **Crime** objects to store. In **CrimeLab**'s constructor, create an empty **List** of **Crime**s. Also, add two methods: a **getCrimes()** method that returns the **List** and a **getCrime(UUID)** that returns the **Crime** with the given ID.

Listing 8.2 Setting up the **List** of **Crime** objects (CrimeLab.java)

```
public class CrimeLab {
    private static CrimeLab sCrimeLab;

    private List<Crime> mCrimes;

    public static CrimeLab get(Context context) {
        ...
    }

    private CrimeLab(Context context) {
        mCrimes = new ArrayList<>();
    }

    public List<Crime> getCrimes() {
        return mCrimes;
    }

    public Crime getCrime(UUID id) {
        for (Crime crime : mCrimes) {
            if (crime.getId().equals(id)) {
                return crime;
            }
        }

        return null;
    }
}
```

List<E> is an interface that supports an ordered list of objects of a given type. It defines methods for retrieving, adding, and deleting elements. A commonly used implementation of **List** is **ArrayList**, which uses a regular Java array to store the list elements.

Because mCrimes holds an **ArrayList** – and **ArrayList** is also a **List** – both **ArrayList** and **List** are valid types for mCrimes. In situations like this, we recommend using the interface type for the variable declaration: **List**. That way, if you ever need to use a different kind of **List** implementation – like **LinkedList**, for example – you can do so easily.

The mCrimes instantiation line uses *diamond notation*, <>, which was introduced in Java 7. This shorthand notation tells the compiler to infer the type of items the **List** will contain based on the generic argument passed in the variable declaration. Here, the compiler will infer that the **ArrayList** contains **Crime**s because the variable declaration `private List<Crime> mCrimes;` specifies **Crime** for the generic argument. (The more verbose equivalent, which developers were required to use prior to Java 7, is `mCrimes = new ArrayList<Crime>();`.)

Eventually, the **`List`** will contain user-created **`Crime`**s that can be saved and reloaded. For now, populate the **`List`** with 100 boring **`Crime`** objects.

Listing 8.3 Generating crimes (`CrimeLab.java`)

```
private CrimeLab(Context context) {
    mCrimes = new ArrayList<>();
    for (int i = 0; i < 100; i++) {
        Crime crime = new Crime();
        crime.setTitle("Crime #" + i);
        crime.setSolved(i % 2 == 0); // Every other one
        mCrimes.add(crime);
    }
}
```

Now you have a fully loaded model layer with 100 crimes.

An Abstract Activity for Hosting a Fragment

In a moment, you will create the **`CrimeListActivity`** class that will host a **`CrimeListFragment`**. First, you are going to set up a view for **`CrimeListActivity`**.

A generic fragment-hosting layout

For **`CrimeListActivity`**, you can simply reuse the layout defined in `activity_crime.xml` (which is copied in Listing 8.4). This layout provides a **`FrameLayout`** as a container view for a fragment, which is then named in the activity's code.

Listing 8.4 `activity_crime.xml` is already generic

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

Because `activity_crime.xml` does not name a particular fragment, you can use it for any activity hosting a single fragment. Rename it `activity_fragment.xml` to reflect its larger scope.

In the project tool window, right-click `res/layout/activity_crime.xml`. (Be sure to right-click `activity_crime.xml` and not `fragment_crime.xml`.)

From the context menu, select Refactor → Rename.... Rename this layout `activity_fragment.xml` and click Refactor.

When you rename a resource, the references to it should be updated automatically. If you see an error in `CrimeActivity.java`, then you need to manually update the reference in **`CrimeActivity`**, as shown in Listing 8.5.

Listing 8.5 Updating layout file for **`CrimeActivity`** (`CrimeActivity.java`)

```
public class CrimeActivity extends AppCompatActivity {
    /** Called when the activity is first created. */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_crime);
        setContentView(R.layout.activity_fragment);

        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();
        }
    }
}
```

An abstract Activity class

To create the **`CrimeListActivity`** class, you could reuse **`CrimeActivity`**'s code. Look back at the code you wrote for **`CrimeActivity`** (which is copied in Listing 8.6). It is simple and almost generic. In fact, the only nongeneric code is the instantiation of the **`CrimeFragment`** before it is added to the **`FragmentManager`**.

Listing 8.6 **`CrimeActivity`** is almost generic (`CrimeActivity.java`)

```
public class CrimeActivity extends AppCompatActivity {
    /** Called when the activity is first created. */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment);

        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();
        }
    }
}
```

Nearly every activity you will create in this book will require the same code. To avoid typing it again and again, you are going to stash it in an abstract class.

Right-click on the `com.bignerdranch.android.criminalintent` package, select New → Java Class, and name the new class **`SingleFragmentActivity`**. Make this class a subclass of **`AppCompatActivity`** and make it an abstract class. Your generated file should look like this:

Listing 8.7 Creating an abstract Activity (`SingleFragmentActivity.java`)

```
public abstract class SingleFragmentActivity extends AppCompatActivity {

}
```

Now, add the following code to `SingleFragmentActivity.java`. Except for the highlighted portions, it is identical to your old **`CrimeActivity`** code.

Listing 8.8 Adding a generic superclass (`SingleFragmentActivity.java`)

```
public abstract class SingleFragmentActivity extends AppCompatActivity {

    protected abstract Fragment createFragment();

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

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

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

In this code, you set the activity's view to be inflated from `activity_fragment.xml`. Then you look for the fragment in the **`FragmentManager`** in that container, creating and adding it if it does not exist.

The only difference between the code in Listing 8.8 and the code in **`CrimeActivity`** is an abstract method named **`createFragment()`** that you use to instantiate the fragment. Subclasses of **`SingleFragmentActivity`** will implement this method to return an instance of the fragment that the activity is hosting.

Using an abstract class

Try it out with **CrimeActivity**. Change **CrimeActivity**'s superclass to **SingleFragmentActivity**, remove the implementation of **onCreate(Bundle)**, and implement the **createFragment()** method as shown in Listing 8.9.

Listing 8.9 Cleaning up **CrimeActivity** (CrimeActivity.java)

```
public class CrimeActivity extends AppCompatActivity SingleFragmentActivity {
    /** Called when the activity is first created. */
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_fragment);

        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();
        }
    }

    @Override
    protected Fragment createFragment() {
        return new CrimeFragment();
    }
}
```

Creating the new controllers

Now, you will create the two new controller classes: **CrimeListActivity** and **CrimeListFragment**.

Right-click on the com.bignerdranch.android.criminalintent package, select New → Java Class, and name the class **CrimeListActivity**.

Modify the new **CrimeListActivity** class to also subclass **SingleFragmentActivity** and implement the **createFragment()** method.

Listing 8.10 Implementing **CrimeListActivity** (CrimeListActivity.java)

```
public class CrimeListActivity extends SingleFragmentActivity {

    @Override
    protected Fragment createFragment() {
        return new CrimeListFragment();
    }
}
```

If you have other methods in your **CrimeListActivity**, such as **onCreate**, remove them. Let **SingleFragmentActivity** do its job and keep **CrimeListActivity** simple.

The **CrimeListFragment** class has not yet been created. Let's remedy that.

Right-click on the com.bignerdranch.android.criminalintent package again, select New → Java Class, and name the class **CrimeListFragment**.

Listing 8.11 Implementing **CrimeListFragment** (CrimeListFragment.java)

```
public class CrimeListFragment extends Fragment {

      // Nothing yet

}
```

For now, **CrimeListFragment** will be an empty shell of a fragment. You will work with this fragment later in the chapter.

Now your activity code is nice and tidy. And **SingleFragmentActivity** will save you a lot of typing and time as you proceed through the book.

Declaring CrimeListActivity

Now that you have created **CrimeListActivity**, you must declare it in the manifest. In addition, you want the list of crimes to be the first screen that the user sees when CriminalIntent is launched, so **CrimeListActivity** should be the launcher activity.

In the manifest, declare **CrimeListActivity** and move the launcher intent filter from **CrimeActivity**'s declaration to **CrimeListActivity**'s declaration.

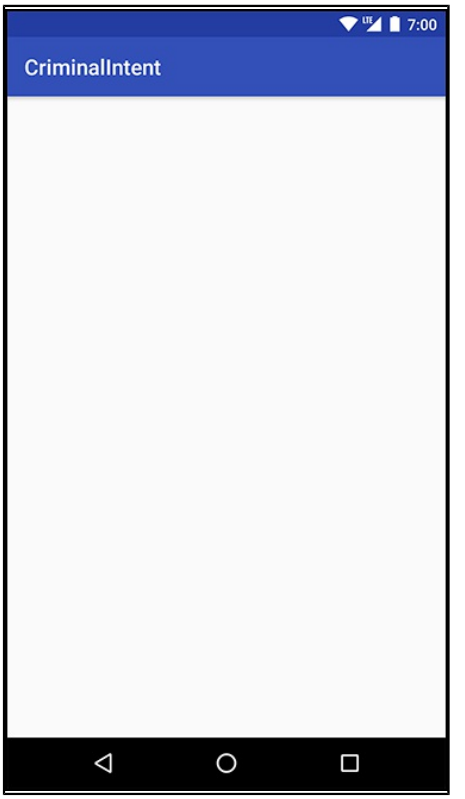
Listing 8.12 Declaring **CrimeListActivity** as the launcher activity (AndroidManifest.xml)

```
<application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity android:name=".CrimeListActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <activity android:name=".CrimeActivity">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>

</application>
```

`CrimeListActivity` is now the launcher activity. Run CriminalIntent and you will see **`CrimeListActivity`**'s **`FrameLayout`** hosting an empty **`CrimeListFragment`**, as shown in Figure 8.3.

Figure 8.3 Blank **`CrimeListActivity`** screen



RecyclerView, Adapter, and ViewHolder

Now, you want **CrimeListFragment** to display a list of crimes to the user. To do this, you will use a **RecyclerView**.

RecyclerView is a subclass of **ViewGroup**. It displays a list of child **View** objects, one for each item in your list of items. Depending on the complexity of what you need to display, these child **View**s can be complex or very simple.

For your first implementation, each item in the list will display the title and date of a **Crime**. The **View** object on each row will be a **LinearLayout** containing two **TextView**s, as shown in Figure 8.4.

Figure 8.4 A **RecyclerView** with child **View**s

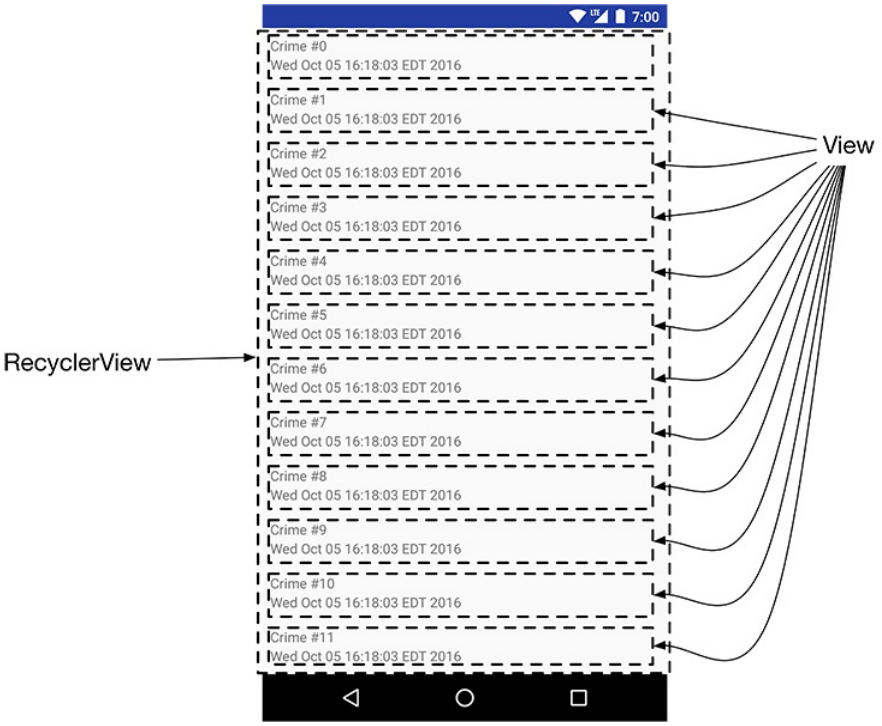


Figure 8.4 shows 12 rows of **View**s. Later you will be able to run CriminalIntent and swipe to scroll through 100 **View**s to see all of your **Crime**s. Does that mean that you have 100 **View** objects in memory? Thanks to your **RecyclerView**, no.

Creating a **View** for every item in the list all at once could easily become unworkable. As you can imagine, a list can have far more than 100 items, and your list items can be much more involved than your simple implementation here. Also, a **Crime** only needs a **View** when it is onscreen, so there is no need to have 100 **View**s ready and waiting. It would make far more sense to create view objects only as you need them.

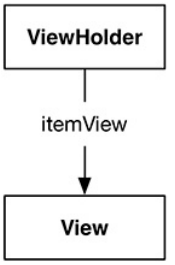
RecyclerView does just that. Instead of creating 100 **View**s, it creates 12 – enough to fill the screen. When a view is scrolled off the screen, **RecyclerView** reuses it rather than throwing it away. In short, it lives up to its name: It recycles views over and over.

ViewHolders and Adapters

The **RecyclerView**'s only responsibilities are recycling **TextView**s and positioning them onscreen. To get the **TextView**s in the first place, it works with two classes that you will build in a moment: an **Adapter** subclass and a **ViewHolder** subclass.

The **ViewHolder**'s job is small, so let's talk about it first. The **ViewHolder** does one thing: It holds on to a **View** (Figure 8.5).

Figure 8.5 The lowly **ViewHolder**



A small job, but that is what **ViewHolder**s do. A typical **ViewHolder** subclass looks like this:

Listing 8.13 A typical **ViewHolder** subclass

```
public class ListRow extends RecyclerView.ViewHolder {
    public ImageView mThumbnail;

    public ListRow(View view) {
        super(view);

        mThumbnail = (ImageView) view.findViewById(R.id.thumbnail);
    }
}
```

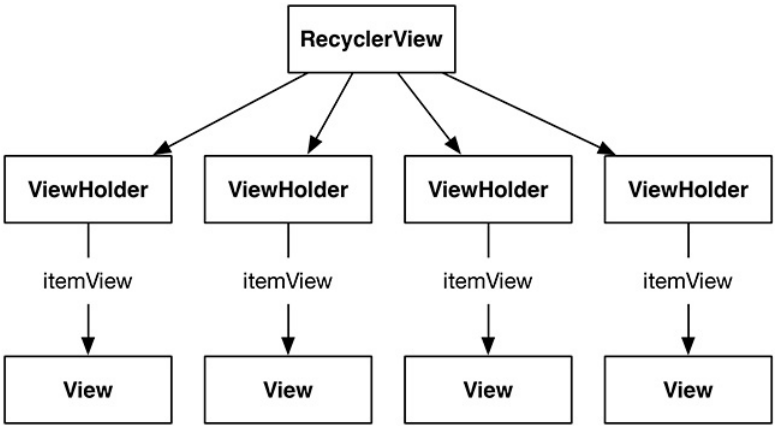
You can then create a **ListRow** and access both `mThumbnail`, which you created yourself, and `itemView`, a field that your superclass **RecyclerView.ViewHolder** assigns for you. The `itemView` field is your **ViewHolder**'s reason for existing: It holds a reference to the entire **View** you passed into **super(view)**.

Listing 8.14 Typical usage of a **ViewHolder**

```
ListRow row = new ListRow(inflater.inflate(R.layout.list_row, parent, false));
View view = row.itemView;
ImageView thumbnailView = row.mThumbnail;
```

A **RecyclerView** never creates **View**s by themselves. It always creates **ViewHolder**s, which bring their `itemView`s along for the ride (Figure 8.6).

Figure 8.6 A **RecyclerView** with its **ViewHolder**s



When the **View** is simple, **ViewHolder** has few responsibilities. For more complicated **View**s, the **ViewHolder** makes wiring up the different parts of `itemView` to a **Crime** simpler and more efficient. You will see how this works later on in this chapter, when you build a complex **View** yourself.

Adapters

Figure 8.6 is somewhat simplified. **RecyclerView** does not create **ViewHolder**s itself. Instead, it asks an *adapter*. An adapter is a controller object that sits between the **RecyclerView** and the data set that the **RecyclerView** should display.

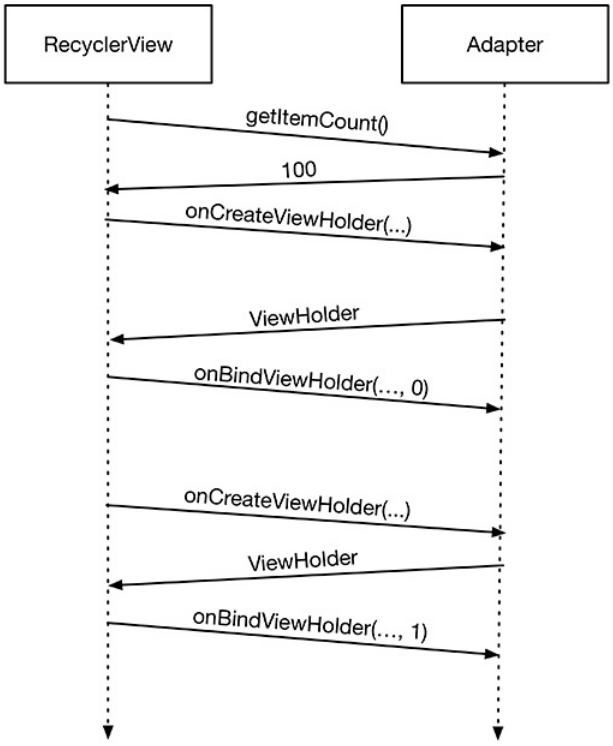
The adapter is responsible for:

- creating the necessary **ViewHolder**s
- binding **ViewHolder**s to data from the model layer

To build an adapter, you first define a subclass of **RecyclerView.Adapter**. Your adapter subclass will wrap the list of crimes you get from **CrimeLab**.

When the **RecyclerView** needs a view object to display, it will have a conversation with its adapter. Figure 8.7 shows an example of a conversation that a **RecyclerView** might initiate.

Figure 8.7 A scintillating **RecyclerView**-**Adapter** conversation



First, the **RecyclerView** asks how many objects are in the list by calling the adapter's **getItemCount()** method.

Then the **RecyclerView** calls the adapter's **onCreateViewHolder(ViewGroup, int)** method to create a new **ViewHolder**, along with its juicy payload: a **View** to display.

Finally, the **RecyclerView** calls **onBindViewHolder(ViewHolder, int)**. The **RecyclerView** will pass a **ViewHolder** into this method along with the position. The adapter will look up the model data for that position and *bind* it to the **ViewHolder**'s **View**. To bind it, the adapter fills in the **View** to reflect the data in the model object.

After this process is complete, **RecyclerView** will place a list item on the screen.
Note that **onCreateViewHolder(ViewGroup, int)** will happen a lot less often than **onBindViewHolder(ViewHolder, int)**. Once enough **ViewHolder**s have been created, **RecyclerView** stops calling **onCreateViewHolder(…)**. Instead, it saves time and memory by recycling old **ViewHolder**s.

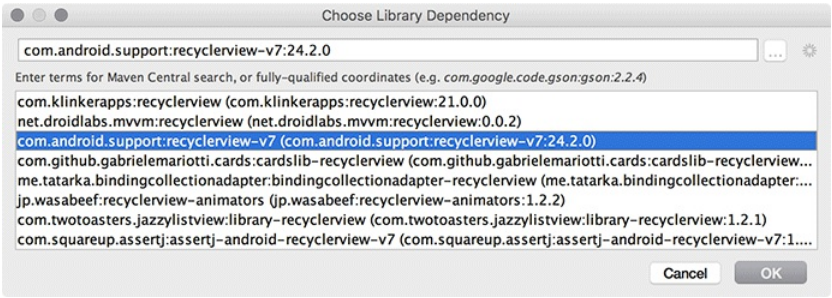
Using a RecyclerView

Enough talk; time for the implementation. The **`RecyclerView`** class lives in one of Google's many support libraries. The first step to using a **`RecyclerView`** is to add the RecyclerView library as a dependency.

Navigate to your project structure window with File → Project Structure.... Select the app module on the left, then the Dependencies tab. Use the + button and choose Library dependency to add a dependency.

Find and select the recyclerview-v7 library and click OK to add the library as a dependency, as shown in Figure 8.8.

Figure 8.8 Adding the RecyclerView dependency



Your **`RecyclerView`** will live in **`CrimeListFragment`**'s layout file. First, you must create the layout file. Right-click on the **`res/layout`** directory and select New → Layout resource file. Name the file `fragment_crime_list` and click OK to create the file.

Open the new `fragment_crime_list` file and modify the root view to be a **`RecyclerView`** and to give it an ID attribute.

Listing 8.15 Adding **`RecyclerView`** to a layout file (`fragment_crime_list.xml`)

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

~~</LinearLayout>~~
<android.support.v7.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/crime_recycler_view"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
```

Now that **CrimeListFragment**'s view is set up, hook up the view to the fragment. Modify **CrimeListFragment** to use this layout file and to find the **RecyclerView** in the layout file, as shown in Listing 8.16.

Listing 8.16 Setting up the view for **CrimeListFragment** (CrimeListFragment.java)

```
public class CrimeListFragment extends Fragment {

    // Nothing yet
    private RecyclerView mCrimeRecyclerView;

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

        mCrimeRecyclerView = (RecyclerView) view
                .findViewById(R.id.crime_recycler_view);
        mCrimeRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));

        return view;
    }

}
```

Note that as soon as you create your **RecyclerView**, you give it another object called a **LayoutManager**. **RecyclerView** requires a **LayoutManager** to work. If you forget to give it one, it will crash.

RecyclerView does not position items on the screen itself. It delegates that job to the **LayoutManager**. The **LayoutManager** positions every item and also defines how scrolling works. So if **RecyclerView** tries to do those things when the **LayoutManager** is not there, the **RecyclerView** will immediately fall over and die.

There are a few built-in **LayoutManager**s to choose from, and you can find more as third-party libraries. You are using the **LinearLayoutManager**, which will position the items in the list vertically. Later on in this book, you will use **GridLayoutManager** to arrange items in a grid instead.

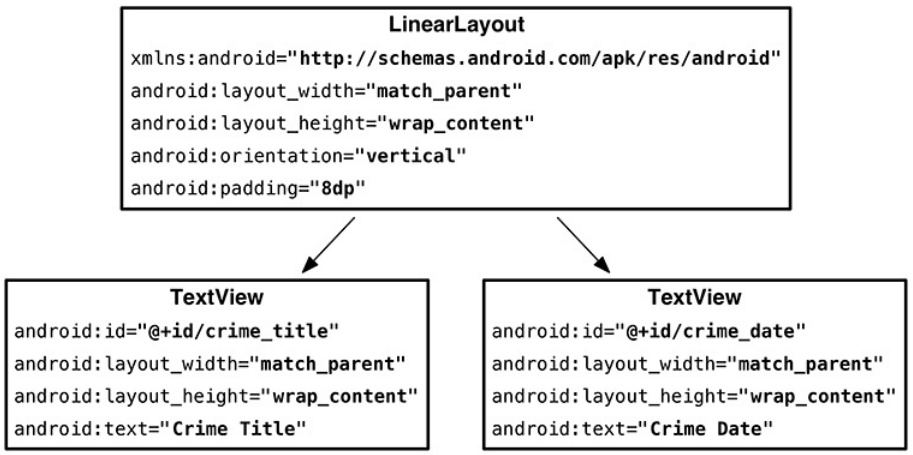
Run the app. You should again see a blank screen, but now you are looking at an empty **RecyclerView**. You will not see any **Crime**s represented on the screen until the **Adapter** and **ViewHolder** implementations are defined.

A view to display

Each item displayed on the **RecyclerView** will have its own view hierarchy, exactly the way **CrimeFragment** has a view hierarchy for the entire screen. You create a new layout for a list item view the same way you do for the view of an activity or a fragment. In the project tool window, right-click the `res/layout` directory and choose New → Layout resource file. In the dialog that appears, name the file `list_item_crime` and click OK.

Update your layout file to add the two **TextView**s as shown in Figure 8.9.

Figure 8.9 Updating the list item layout file (`list_item_crime.xml`)



Take a look at the design preview, and you will see that you have created exactly one row of the completed product. In a moment, you will see how **RecyclerView** will create those rows for you.

Implementing a ViewHolder and an Adapter

The next job is to define the **ViewHolder** that will inflate and own your layout. Define it as an inner class in **CrimeListFragment**.

Listing 8.17 The beginnings of a **ViewHolder** (`CrimeListFragment.java`)

```
public class CrimeListFragment extends Fragment {
    ...
     private class CrimeHolder extends RecyclerView.ViewHolder {
        public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
            super(inflater.inflate(R.layout.list_item_crime, parent, false));
        }
    }
}
```

In **CrimeHolder**'s constructor, you inflate `list_item_crime.xml`. Immediately you pass it into **super(…)**, **ViewHolder**'s constructor. The base **ViewHolder** class will then hold on to the `fragment_crime_list.xml` view hierarchy. If you need that view hierarchy, you can find it in **ViewHolder**'s `itemView` field.

CrimeHolder is all skin and bones right now. Later in the chapter, **CrimeHolder** will beef up as you give it more work to do.

With the **ViewHolder** defined, create the adapter.

Listing 8.18 The beginnings of an adapter (CrimeListFragment.java)

```
public class CrimeListFragment extends Fragment {
    ...
    private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder> {

        private List<Crime> mCrimes;

        public CrimeAdapter(List<Crime> crimes) {
            mCrimes = crimes;
        }
    }
}
```

(The code in Listing 8.18 will not compile. You will fix this in a moment.)

When the **RecyclerView** needs to display a new **ViewHolder** or connect a **Crime** object to an existing **ViewHolder**, it will ask this adapter for help by calling a method on it. The **RecyclerView** itself will not know anything about the **Crime** object, but the **Adapter** will know all of **Crime**'s intimate and personal details.

Next, implement three method overrides in **CrimeAdapter**. (You can automatically generate these overrides by putting your cursor on top of `extends`, keying in Option-Return (Alt+Enter), selecting Implement methods, and clicking OK. Then you only need to fill in the bodies.)

Listing 8.19 Filling out **CrimeAdapter** (CrimeListFragment.java)

```
    private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder> {
        ...
        @Override
        public CrimeHolder onCreateViewHolder(ViewGroup parent, int viewType) {
            LayoutInflater layoutInflater = LayoutInflater.from(getActivity());

            return new CrimeHolder(layoutInflater, parent);
        }

        @Override
        public void onBindViewHolder(CrimeHolder holder, int position) {

        }

        @Override
        public int getItemCount() {
            return mCrimes.size();
        }
    }
```

onCreateViewHolder is called by the **RecyclerView** when it needs a new **ViewHolder** to display an item with. In this method, you create a **LayoutInflater** and use it to construct a new **CrimeHolder**.

Your adapter must have an override for **onBindViewHolder(…)**, but for now you can leave it empty. In a moment, you will return to it.

Now that you have an **Adapter**, connect it to your **RecyclerView**. Implement a method called **updateUI** that sets up **CrimeListFragment**'s UI. For now it will create a **CrimeAdapter** and set it on the **RecyclerView**.

Listing 8.20 Setting an **Adapter** (CrimeListFragment.java)

```
public class CrimeListFragment extends Fragment {

    private RecyclerView mCrimeRecyclerView;
    private CrimeAdapter mAdapter;

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

        mCrimeRecyclerView = (RecyclerView) view
                .findViewById(R.id.crime_recycler_view);
        mCrimeRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));

        updateUI();

        return view;
    }

    private void updateUI() {
        CrimeLab crimeLab = CrimeLab.get(getActivity());
        List<Crime> crimes = crimeLab.getCrimes();

        mAdapter = new CrimeAdapter(crimes);
        mCrimeRecyclerView.setAdapter(mAdapter);
    }
    ...
}
```

In later chapters, you will add more to **updateUI()** as configuring your UI gets more involved.

Run CriminalIntent and scroll through your new **RecyclerView**, which should look like Figure 8.10.

Figure 8.10 A beautiful list of... beautiful, beautiful beautifuls

Hmm. Looking a little repetitive there, Mr. **RecyclerView**. Swipe or drag down, and you will see even more identical views scroll across your screen.

In the screenshot above, there are 11 rows, which means that **onCreateViewHolder(…)** was called 11 times. If you scroll down, a few more **CrimeHolder**s may be created, but at a certain point **RecyclerView** will stop creating new **CrimeHolder**s. Instead, it will recycle old **CrimeHolder**s as they scroll off the top of the screen. **RecyclerView**, you were named well indeed.

For the moment, every row is identical. In your next step, you will fill each **CrimeHolder** with fresh information as it is recycled by binding to it.

Binding List Items

Binding is taking Java code (like model data in a **Crime**, or click listeners) and hooking it up to a widget. So far, in all the exercises up until this point in the book, you bound each and every time you inflated a view. This meant there was no need to split that work into its own method. However, now that views are being recycled, it pays to have creation in one place and binding in another.

All the code that will do the real work of binding will go inside your **CrimeHolder**. That work starts with pulling out all the widgets you are interested in. This only needs to happen one time, so write this code in your constructor.

Listing 8.21 Pulling out views in the constructor (`CrimeListFragment.java`)

```
private class CrimeHolder extends RecyclerView.ViewHolder {

    private TextView mTitleTextView;
    private TextView mDateTextView;

    public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
        super(inflater.inflate(R.layout.list_item_crime, parent, false));

        mTitleTextView = (TextView) itemView.findViewById(R.id.crime_title);
        mDateTextView = (TextView) itemView.findViewById(R.id.crime_date);
    }
}
```

Your **`CrimeHolder`** will also need a **`bind(Crime)`** method. This will be called each time a new **`Crime`** should be displayed in your **`CrimeHolder`**. First, add **`bind(Crime)`**.

Listing 8.22 Writing a **`bind(Crime)`** method (`CrimeListFragment.java`)

```
private class CrimeHolder extends RecyclerView.ViewHolder {

    private Crime mCrime;
    ...
    public void bind(Crime crime) {
        mCrime = crime;
        mTitleTextView.setText(mCrime.getTitle());
        mDateTextView.setText(mCrime.getDate().toString());
    }
}
```

When given a **`Crime`**, **`CrimeHolder`** will now update the title **`TextView`** and date **`TextView`** to reflect the state of the **`Crime`**.

Next, call your newly minted **`bind(Crime)`** method each time the **`RecyclerView`** requests that a given **`CrimeHolder`** be bound to a particular crime.

Listing 8.23 Calling the **`bind(Crime)`** method (`CrimeListFragment.java`)

```
private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder> {
    ...
    @Override
    public void onBindViewHolder(CrimeHolder holder, int position) {
        Crime crime = mCrimes.get(position);
        holder.bind(crime);
    }
    ...
}
```

Run CriminalIntent one more time, and every visible **`CrimeHolder`** should now display a distinct **`Crime`** (Figure 8.11).

Figure 8.11 All right, all right, all right

When you fling the view up, the scrolling animation should feel as smooth as warm butter. This effect is a direct result of keeping **`onBindViewHolder(…)`** small and efficient, doing only the minimum amount of work necessary.

Take heed: Always be efficient in your **`onBindViewHolder(…)`**. Otherwise, your scroll animation could feel as chunky as cold Parmesan cheese.

Responding to Presses

As icing on the **`RecyclerView`** cake, CriminalIntent should also respond to a press on these list items. In Chapter 10, you will launch the detail view for a **`Crime`** when the user presses on that **`Crime`** in the list. For now, show a **`Toast`** when the user takes action on a **`Crime`**.

As you may have noticed, **`RecyclerView`**, while powerful and capable, has precious few real responsibilities. (May it be an example to us all.) The same goes here: Handling touch events is mostly up to you. If you need them, **`RecyclerView`** can forward along raw touch events. But most of the time this is not necessary.

Instead, you can handle them like you normally do: by setting an **`OnClickListener`**. Since each **`View`** has an associated **`ViewHolder`**, you can make your **`ViewHolder`** the **`OnClickListener`** for its **`View`**.

Modify the **`CrimeHolder`** to handle presses for the entire row.

Listing 8.24 Detecting presses in **`CrimeHolder`** (`CrimeListFragment.java`)

```
private class CrimeHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener {
    ...
    public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
        super(inflater.inflate(R.layout.list_item_crime, parent, false));
        itemView.setOnClickListener(this);
        ...
    }
    ...
    @Override
    public void onClick(View view) {
        Toast.makeText(getActivity(),
                mCrime.getTitle() + " clicked!", Toast.LENGTH_SHORT)
                .show();
    }
}
```

In Listing 8.24, the **`CrimeHolder`** itself is implementing the **`OnClickListener`** interface. On the **`itemView`**, which is the **`View`** for the entire row, the **`CrimeHolder`** is set as the receiver of click events.

Run CriminalIntent and press on an item in the list. You should see a **`Toast`** indicating that the item was clicked.

For the More Curious: ListView and GridView

The core Android OS includes **`ListView`**, **`GridView`**, and **`Adapter`** classes. Until the release of Android 5.0, these were the preferred ways to create lists or grids of items.

The API for these components is very similar to that of a **`RecyclerView`**. The **`ListView`** or **`GridView`** class is responsible for scrolling a collection of items, but it does not know much about each of those items. The **`Adapter`** is responsible for creating each of the **`View`**s in the list. However, **`ListView`** and **`GridView`** do not enforce that you use the **`ViewHolder`** pattern (though you can – and should – use it).

These old implementations are replaced by the **`RecyclerView`** implementation because of the complexity required to alter the behavior of a **`ListView`** or **`GridView`**.

Creating a horizontally scrolling **`ListView`**, for example, is not included in the **`ListView`** API and requires a lot of work. Creating custom layout and scrolling behavior with a **`RecyclerView`** is still a lot of work, but **`RecyclerView`** was built to be extended, so it is not quite so bad.

Another key feature of **`RecyclerView`** is the animation of items in the list. Animating the addition or removal of items in a **`ListView`** or **`GridView`** is a complex and error-prone task. **`RecyclerView`** makes this much easier, includes a few built-in animations, and allows for easy customization of these animations.

For example, if you found out that the crime at position 0 moved to position 5, you could animate that change like so:

```
mRecyclerView.getAdapter().notifyItemMoved(0, 5);
```

For the More Curious: Singletons

The singleton pattern, as used in the **`CrimeLab`**, is very common in Android. Singletons get a bad rap because they can be misused in a way that makes an app hard to maintain.

Singletons are often used in Android because they outlive a single fragment or activity. A singleton will still exist across rotation and will exist as you move between activities and fragments in your application.

Singletons make a convenient owner of your model objects. Imagine a more complex CriminalIntent application that had many activities and fragments modifying crimes. When one controller modifies a crime, how would you make sure that updated crime was sent over to the other controllers? If the **`CrimeLab`** is the owner of crimes and all modifications to crimes pass through it, propagating changes is much easier. As you transition between controllers, you can pass the crime ID as an identifier for a particular crime and have each controller pull the full crime object from the **`CrimeLab`** using that ID.

However, singletons do have a few downsides. For example, while they allow for an easy place to stash data with a longer lifetime than a controller, singletons do have a lifetime. Singletons will be destroyed, along with all of their instance variables, as Android reclaims memory at some point after you switch out of an application. Singletons are not a long-term storage solution. (Writing the files to disk or sending them to a web server is.)

Singletons can also make your code hard to unit test. There is not a great way to replace the **`CrimeLab`** instance in this chapter with a mock version of itself because the code is calling a static method directly on the **`CrimeLab`** object. In practice, Android developers usually solve this problem using a tool called a *dependency injector*. This tool allows for objects to be shared as singletons, while still making it possible to replace them when needed.

Singletons also have the potential to be misused. The temptation is to use singletons for everything because they are convenient – you can get to them wherever you are, and store whatever information you need to get at later. But when you do that, you are avoiding answering important questions: Where is this data used? Where is this method important?

A singleton does not answer those questions. So whoever comes after you will open up your singleton and find something that looks like somebody's disorganized junk drawer. Batteries, zip ties, old photographs? What is all this here for? Make sure that anything in your singleton is truly global and has a strong reason for being there.

On balance, however, singletons are a key component of a well-architected Android app – when used correctly.

Challenge: RecyclerView ViewTypes

For this advanced challenge, you will create two types of rows in your **`RecyclerView`**: a normal row and a row for more serious crimes. To implement this, you will work with the *view type* feature available in **`RecyclerView.Adapter`**. Add a new property, `mRequiresPolice`, to the **`Crime`** object and use it to determine which view to load on the **`CrimeAdapter`** by implementing the **`getItemViewType(int)`** method (`developer.android.com/reference/android/support/v7/widget/RecyclerView.Adapter.html#getItemViewType(int)`).

In the **`onCreateViewHolder(ViewGroup, int)`** method, you will also need to add logic that returns a different **`ViewHolder`** based on the new `viewType` value returned by **`getItemViewType(int)`**. Use the original layout for crimes that do not require police intervention and a new layout with a streamlined interface containing a button that says "contact police" for crimes that do.