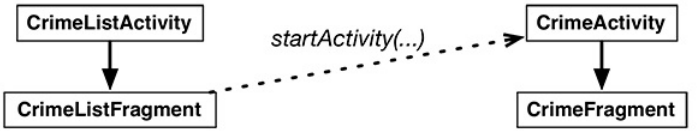
10
Using Fragment Arguments

In this chapter, you will get the list and the detail parts of CriminalIntent working together. When a user presses an item in the list of crimes, a new **CrimeActivity** hosting a **CrimeFragment** will appear and display the details for that instance of **Crime** (Figure 10.1).

Figure 10.1 Starting **CrimeActivity** from **CrimeListActivity**



In GeoQuiz, you had one activity (**QuizActivity**) start another activity (**CheatActivity**). In CriminalIntent, you are going to start the **CrimeActivity** from a fragment. In particular, you will have **CrimeListFragment** start an instance of **CrimeActivity**.

Starting an Activity from a Fragment

Starting an activity from a fragment works nearly the same as starting an activity from another activity. You call the **Fragment.startActivity(Intent)** method, which calls the corresponding **Activity** method behind the scenes.

In **CrimeListFragment**'s **CrimeHolder**, begin by replacing the toast with code that starts an instance of **CrimeActivity**.

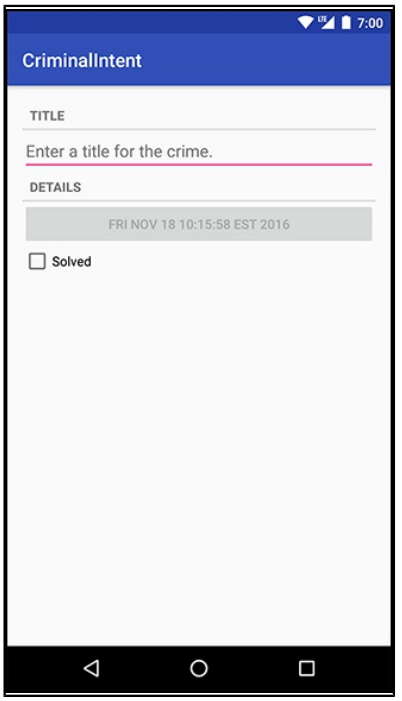
Listing 10.1 Starting **CrimeActivity** (CrimeListFragment.java)

```
private class CrimeHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener {
    ...
    @Override
    public void onClick(View view) {
        Toast.makeText(getActivity(),
                mCrime.getTitle() + " clicked!", Toast.LENGTH_SHORT)
                .show();
        Intent intent = new Intent(getActivity(), CrimeActivity.class);
        startActivity(intent);
    }
}
```

Here **`CrimeListFragment`** creates an explicit intent that names the **`CrimeActivity`** class. **`CrimeListFragment`** uses the **`getActivity()`** method to pass its hosting activity as the **`Context`** object that the **`Intent`** constructor requires.

Run CriminalIntent. Press any list item, and you will see a new **`CrimeActivity`** hosting a **`CrimeFragment`** (Figure 10.2).

Figure 10.2 A blank **`CrimeFragment`**



The **`CrimeFragment`** does not yet display the data for a specific **`Crime`**, because you have not told it which **`Crime`** to display.

Putting an extra

You can tell **CrimeFragment** which **Crime** to display by passing the crime ID as an **Intent** extra when **CrimeActivity** is started.

Start by creating a **newIntent** method in **CrimeActivity**.

Listing 10.2 Creating a **newIntent** method (CrimeActivity.java)

```
public class CrimeActivity extends SingleFragmentActivity {

    public static final String EXTRA_CRIME_ID =
            "com.bignerdranch.android.criminalintent.crime_id";

    public static Intent newIntent(Context packageContext, UUID crimeId) {
        Intent intent = new Intent(packageContext, CrimeActivity.class);
        intent.putExtra(EXTRA_CRIME_ID, crimeId);
        return intent;
    }
    ...
}
```

After creating an explicit intent, you call **putExtra(…)** and pass in a string key and the value the key maps to (the crimeId). In this case, you are calling **putExtra(String, Serializable)** because **UUID** is a **Serializable** object.

Now, update the **CrimeHolder** to use the **newIntent** method while passing in the crime ID.

Listing 10.3 Stashing and passing a **Crime** (CrimeListFragment.java)

```
private class CrimeHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener {
    ...
    @Override
    public void onClick(View view) {
        Intent intent = new Intent(getActivity(), CrimeActivity.class);
        Intent intent = CrimeActivity.newIntent(getActivity(), mCrime.getId());
        startActivity(intent);
    }
}
```

Retrieving an extra

The crime ID is now safely stashed in the intent that belongs to **CrimeActivity**. However, it is the **CrimeFragment** class that needs to retrieve and use that data.

There are two ways a fragment can access data in its activity's intent: an easy, direct shortcut and a complex, flexible implementation. First, you are going to try out the shortcut. Then you will implement the complex and flexible solution.

In the shortcut, **CrimeFragment** will simply use the **getActivity()** method to access the **CrimeActivity**'s intent directly. In CrimeFragment.java, retrieve the extra from **CrimeActivity**'s intent and use it to fetch the **Crime**.

Listing 10.4 Retrieving the extra and fetching the **`Crime`** (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {
    ...
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCrime = new Crime();
        UUID crimeId = (UUID) getActivity().getIntent()
                .getSerializableExtra(CrimeActivity.EXTRA_CRIME_ID);
        mCrime = CrimeLab.get(getActivity()).getCrime(crimeId);
    }
    ...
}
```

In Listing 10.4, other than the call to **`getActivity()`**, the code is the same as if you were retrieving the extra from the activity's code. The **`getIntent()`** method returns the **`Intent`** that was used to start **`CrimeActivity`**. You call **`getSerializableExtra(String)`** on the **`Intent`** to pull the **`UUID`** out into a variable.

After you have retrieved the ID, you use it to fetch the **`Crime`** from **`CrimeLab`**.

Updating CrimeFragment's view with Crime data

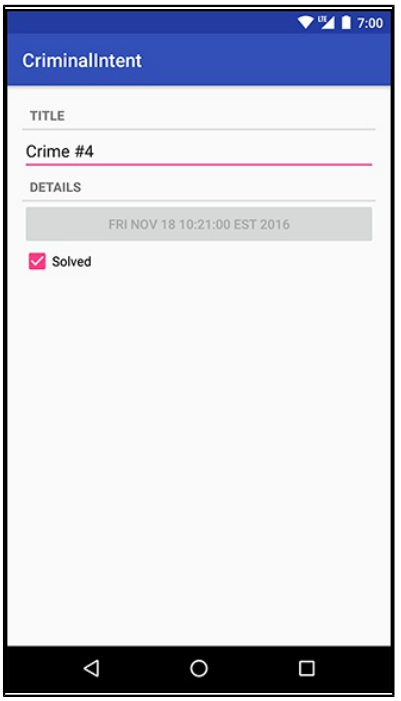
Now that **`CrimeFragment`** fetches a **`Crime`**, its view can display that **`Crime`**'s data. Update **`onCreateView(…)`** to display the **`Crime`**'s title and solved status. (The code for displaying the date is already in place.)

Listing 10.5 Updating view objects (`CrimeFragment.java`)

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
    ...
    mTitleField = (EditText)v.findViewById(R.id.crime_title);
    mTitleField.setText(mCrime.getTitle());
    mTitleField.addTextChangedListener(new TextWatcher() {
        ...
    });
    ...
    mSolvedCheckBox = (CheckBox)v.findViewById(R.id.crime_solved);
    mSolvedCheckBox.setChecked(mCrime.isSolved());
    mSolvedCheckBox.setOnCheckedChangeListener(new OnCheckedChangeListener() {
        ...
    });
    ...
    return v;
}
```

Run CriminalIntent. Select Crime #4 and watch a **`CrimeFragment`** instance with the correct crime data appear (Figure 10.3).

Figure 10.3 The crime that you wanted to see



The downside to direct retrieval

Having the fragment access the intent that belongs to the hosting activity makes for simple code. However, it costs you the encapsulation of your fragment. **`CrimeFragment`** is no longer a reusable building block because it expects that it will always be hosted by an activity whose **`Intent`** defines an extra named `com.bignerdranch.android.criminalintent.crime_id`.

This may be a reasonable expectation on **`CrimeFragment`**'s part, but it means that **`CrimeFragment`**, as currently written, cannot be used with just any activity.

A better solution is to stash the crime ID someplace that belongs to **`CrimeFragment`** rather than keeping it in **`CrimeActivity`**'s personal space. The **`CrimeFragment`** could then retrieve this data without relying on the presence of a particular extra in the activity's intent. The "someplace" that belongs to a fragment is known as its *arguments bundle*.

Fragment Arguments

Every fragment instance can have a **`Bundle`** object attached to it. This bundle contains key-value pairs that work just like the intent extras of an **`Activity`**. Each pair is known as an *argument*.

To create fragment arguments, you first create a **`Bundle`** object. Next, you use type-specific "put" methods of **`Bundle`** (similar to those of **`Intent`**) to add arguments to the bundle:

```
    Bundle args = new Bundle();
    args.putSerializable(ARG_MY_OBJECT, myObject);
    args.putInt(ARG_MY_INT, myInt);
    args.putCharSequence(ARG_MY_STRING, myString);
```

Attaching arguments to a fragment

To attach the arguments bundle to a fragment, you call **`Fragment.setArguments(Bundle)`**. Attaching arguments to a fragment must be done after the fragment is created but before it is added to an activity.

To hit this window, Android programmers follow a convention of adding a static method named **`newInstance()`** to the **`Fragment`** class. This method creates the fragment instance and bundles up and sets its arguments.

When the hosting activity needs an instance of that fragment, you have it call the **`newInstance(…)`** method rather than calling the constructor directly. The activity can pass in any required parameters to **`newInstance(…)`** that the fragment needs to create its arguments.

In **`CrimeFragment`**, write a **`newInstance(UUID)`** method that accepts a **`UUID`**, creates an arguments bundle, creates a fragment instance, and then attaches the arguments to the fragment.

Listing 10.6 Writing a **`newInstance(UUID)`** method (`CrimeFragment.java`)

```
public class CrimeFragment extends Fragment {

    private static final String ARG_CRIME_ID = "crime_id";

    private Crime mCrime;
    private EditText mTitleField;
    private Button mDateButton;
    private CheckBox mSolvedCheckbox;

    public static CrimeFragment newInstance(UUID crimeId) {
        Bundle args = new Bundle();
        args.putSerializable(ARG_CRIME_ID, crimeId);

        CrimeFragment fragment = new CrimeFragment();
        fragment.setArguments(args);
        return fragment;
    }
    ...
}
```

Now, **`CrimeActivity`** should call **`CrimeFragment.newInstance(UUID)`** when it needs to create a **`CrimeFragment`**. It will pass in the **`UUID`** it retrieved from its extra. Return to **`CrimeActivity`** and, in **`createFragment()`**, retrieve the extra from **`CrimeActivity`**'s intent and pass it into **`CrimeFragment.newInstance(UUID)`**.

You can now also make **EXTRA_CRIME_ID** private, because no other class will access that extra.

Listing 10.7 Using **newInstance(UUID)** (CrimeActivity.java)

```
public class CrimeActivity extends SingleFragmentActivity {

    public private static final String EXTRA_CRIME_ID =
            "com.bignerdranch.android.criminalintent.crime_id";
    ...
    @Override
    protected Fragment createFragment() {
        return new CrimeFragment();
        UUID crimeId = (UUID) getIntent()
                .getSerializableExtra(EXTRA_CRIME_ID);
        return CrimeFragment.newInstance(crimeId);
    }
}
```

Notice that the need for independence does not go both ways. **CrimeActivity** has to know plenty about **CrimeFragment**, including that it has a **newInstance(UUID)** method. This is fine. Hosting activities should know the specifics of how to host their fragments, but fragments should not have to know specifics about their activities. At least, not if you want to maintain the flexibility of independent fragments.

Retrieving arguments

When a fragment needs to access its arguments, it calls the **Fragment** method **getArguments()** and then one of the type-specific "get" methods of **Bundle**.

Back in **CrimeFragment.onCreate(…)**, replace your shortcut code with retrieving the **UUID** from the fragment arguments.

Listing 10.8 Getting crime ID from the arguments (CrimeFragment.java)

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    UUID crimeId = (UUID) getActivity().getIntent()
            .getSerializableExtra(CrimeActivity.EXTRA_CRIME_ID);
    UUID crimeId = (UUID) getArguments().getSerializable(ARG_CRIME_ID);
    mCrime = CrimeLab.get(getActivity()).getCrime(crimeId);

}
```

Run CriminalIntent. The app will behave the same, but you should feel all warm and fuzzy inside for maintaining **CrimeFragment**'s independence. You are also well prepared for the next chapter, where you will implement more sophisticated navigation in CriminalIntent.

Reloading the List

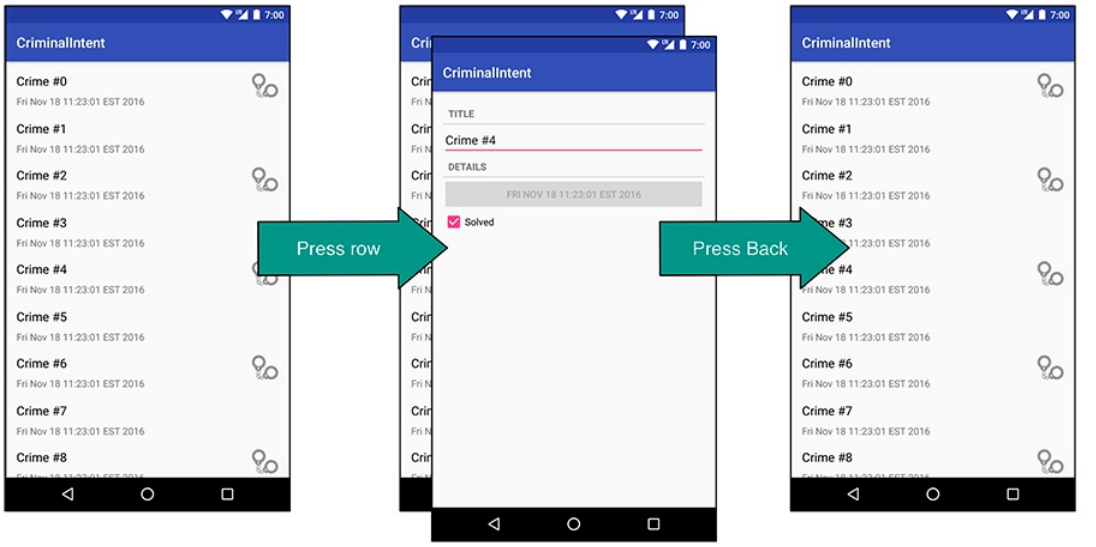
There is one more detail to take care of. Run CriminalIntent, press a list item, and then modify that **`Crime`**'s details. These changes are saved to the model, but when you return to the list, the **`RecyclerView`** is unchanged.

The **`RecyclerView`**'s **`Adapter`** needs to be informed that the data has changed (or may have changed) so that it can refetch the data and reload the list. You can work with the **`ActivityManager`**'s back stack to reload the list at the right moment.

When **`CrimeListFragment`** starts an instance of **`CrimeActivity`**, the **`CrimeActivity`** is put on top of the stack. This pauses and stops the instance of **`CrimeListActivity`** that was initially on top.

When the user presses the Back button to return to the list, the **`CrimeActivity`** is popped off the stack and destroyed. At that point, the **`CrimeListActivity`** is started and resumed (Figure 10.4).

Figure 10.4 CriminalIntent's back stack



When the **`CrimeListActivity`** is resumed, it receives a call to **`onResume()`** from the OS. When **`CrimeListActivity`** receives this call, its **`FragmentManager`** calls **`onResume()`** on the fragments that the activity is currently hosting. In this case, the only fragment is **`CrimeListFragment`**.

In **CrimeListFragment**, override **onResume()** and trigger a call to **updateUI()** to reload the list. Modify the **updateUI()** method to call **notifyDataSetChanged()** if the **CrimeAdapter** is already set up.

Listing 10.9 Reloading the list in **onResume()** (CrimeListFragment.java)

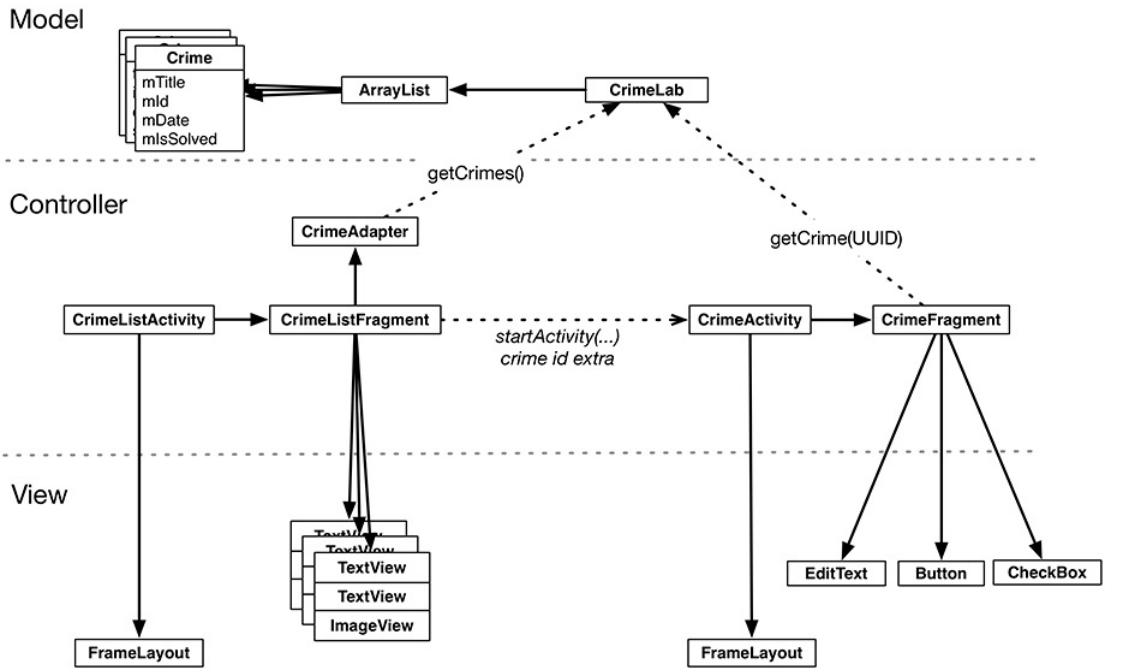
```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    ...
}

@Override
public void onResume() {
    super.onResume();
    updateUI();
}

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

    if (mAdapter == null) {
        mAdapter = new CrimeAdapter(crimes);
        mCrimeRecyclerView.setAdapter(mAdapter);
    } else {
        mAdapter.notifyDataSetChanged();
    }
}
```

Why override **onResume()** to update the **RecyclerView** and not **onStart()**? You cannot assume that your activity will be stopped when another activity is in front of it. If the other activity is transparent, your activity may just be paused. If your activity is paused and your update code is in **onStart()**, then the list will not be reloaded. In general, **onResume()** is the safest place to take action to update a fragment's view.

Run CriminalIntent. Select a crime and change its details. When you return to the list, you will immediately see your changes.

You have made progress with CriminalIntent in the last two chapters. Let's take a look at an updated object diagram (Figure 10.5).

Figure 10.5 Updated object diagram for CriminalIntent



Getting Results with Fragments

In this chapter, you did not need a result back from the started activity. But what if you did? Your code would look a lot like it did in GeoQuiz. Instead of using **`Activity`**'s **`startActivityForResult(…)`** method, you would use **`Fragment.startActivityForResult(…)`**. Instead of overriding **`Activity.onActivityResult(…)`**, you would override **`Fragment.onActivityResult(…)`**:

```
public class CrimeListFragment extends Fragment {

    private static final int REQUEST_CRIME = 1;
    ...
    private class CrimeHolder extends RecyclerView.ViewHolder
            implements View.OnClickListener {
        ...
        @Override
        public void onClick(View view) {
            Intent intent = CrimeActivity.newIntent(getActivity(), mCrime.getId());
            startActivityForResult(intent, REQUEST_CRIME);
        }
    }

    @Override
    public void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == REQUEST_CRIME) {
            // Handle result
        }
    }
    ...
}
```

`Fragment.startActivityForResult(Intent, int)` is similar to the **`Activity`** method with the same name. It includes some additional code to route the result to your fragment from its host activity.

Returning results from a fragment is a bit different. A fragment can receive a result from an activity, but it cannot have its own result. Only activities have results. So while **`Fragment`** has its own **`startActivityForResult(…)`** and **`onActivityResult(…)`** methods, it does not have any **`setResult(…)`** methods.

Instead, you tell the *host activity* to return a value. Like this:

```
public class CrimeFragment extends Fragment {
    ...
    public void returnResult() {
        getActivity().setResult(Activity.RESULT_OK, null);
    }
}
```

For the More Curious: Why Use Fragment Arguments?

This all seems so complicated. Why not just set an instance variable on the **`CrimeFragment`** when it is created?

Because it would not always work. When the OS re-creates your fragment – either across a configuration change or when the user has switched out of your app and the OS reclaims memory – all of your instance variables will be lost. Also, remember that there is no way to cheat low-memory death, no matter how hard you try.

If you want something that works in all cases, you have to persist your arguments.

One option is to use the saved instance state mechanism. You can store the crime ID as a normal instance variable, save the crime ID in **`onSaveInstanceState(Bundle)`**, and snag it from the **`Bundle`** in **`onCreate(Bundle)`**. This will work in all situations.

However, that solution is hard to maintain. If you revisit this fragment in a few years and add another argument, you may not remember to save the argument in **`onSaveInstanceState(Bundle)`**. Going this route is less explicit.

Android developers prefer the fragment arguments solution because it is very explicit and clear in its intentions. In a few years, you will come back and know that the crime ID is an argument and is safely shuttled along to new instances of this fragment. If you add another argument, you will know to stash it in the arguments bundle.

Challenge: Efficient RecyclerView Reloading

The **`notifyDataSetChanged`** method on your **`Adapter`** is a handy way to ask the **`RecyclerView`** to reload all of the items that are currently visible.

The use of this method in CriminalIntent is wildly inefficient because at most one **`Crime`** will have changed when returning to the **`CrimeListFragment`**.

Use the **`RecyclerView.Adapter`**'s **`notifyItemChanged(int)`** method to reload a single item in the list. Modifying the code to call that method is easy. The challenge is discovering which position has changed and reloading the correct item.

Challenge: Improving CrimeLab Performance

`CrimeLab`'s **`get(UUID)`** method works, but checking each crime's ID against the ID you are looking for one at a time can be improved upon. Improve the performance of the lookup, making sure that CriminalIntent's existing behavior remains unchanged as you refactor.