Level/Subject : **D3/Programming 6**
Topic : List with RecyclerView in Android apps
Week : 4
Activity : Creating List with RecyclerView in Android apps using Android Studio
Alocated time : 60 mins labs
Deliverables : Project folder
Due date : end of week

Competency :

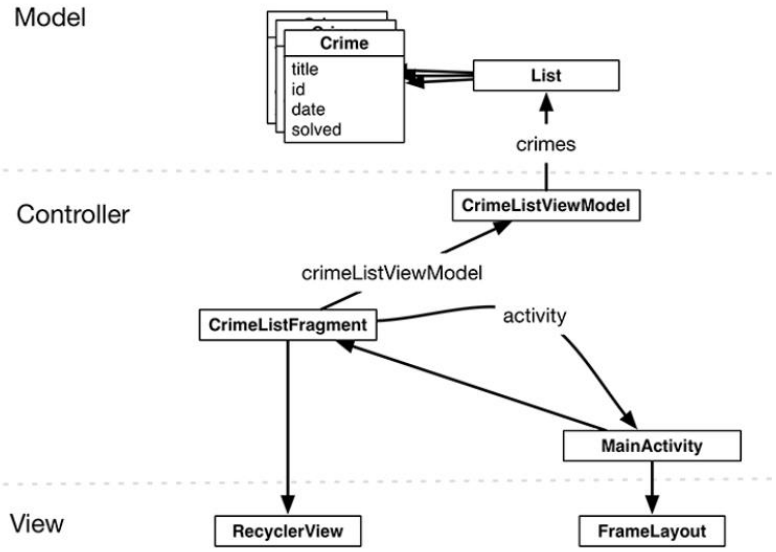
Student expected to be able to create List with RecyclerView in Android apps using Android Studio IDE.

Example Practice Task:

Create List with RecyclerView in Android apps using Android Studio.

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

2. CriminalIntent with list of crimes



Adding a New Fragment and ViewModel

3. The first step is to add a new `ViewModel` to store the `List` of `Crime` objects you will eventually display on the screen. As you learned the `ViewModel` class is part of the `lifecycle-extensions` library. So begin by adding the `lifecycle-extensions` dependency to your `app/build.gradle` file. Do not forget to sync your Gradle files after making this change.

build.gradle (:app)

```
1   plugins {id 'com.android.application'}
4
5   android {...}
30
31  dependencies {
32
33      implementation 'androidx.appcompat:appcompat:1.2.0'
34      implementation 'com.google.android.material:material:1.3.0'
35      implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
36      implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
37      testImplementation 'junit:junit:4.+'
38      androidTestImplementation 'androidx.test.ext:junit:1.1.2'
39      androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'
40  }
```

4. Next, create a new Java class called `CrimeListViewModel`. Update the new `CrimeListViewModel` class to extend from `ViewModel`. Add a member variable to store a list of `Crime`s. populate the list with dummy data.

CrimeListViewModel.java

```
8  public class CrimeListViewModel extends ViewModel {
9      private List<Crime> mCrimes;
10
11     public CrimeListViewModel(){
12     }
13
14     public List<Crime> getCrimes(){
15         if (mCrimes == null) {
16             mCrimes = new ArrayList<>();
17             loadCrimes();
18         }
19         return mCrimes;
20     }
21
22     private void loadCrimes() {
23         for (int i = 0; i < 100; i++){
24             Crime crime = new Crime();
25             crime.setTitle("Crime #" + i);
26             crime.setSolved(i % 2 == 0); //every other one
27             mCrimes.add(crime);
28         }
29     }
30 }
```

5. Eventually, the `List` will contain user-created `Crime`s that can be saved and reloaded. For now, you populate the `List` with 100 boring `Crime` objects.
6. The `CrimeListViewModel` is not a solution for long-term storage of data, but it does encapsulate all the data necessary to populate `CrimeListFragment`'s view.
7. The next step is to add a new `CrimeListFragment` class and associate it with `CrimeListViewModel`. Create the `CrimeListFragment` class and make it a subclass of `androidx.fragment.app.Fragment`.

CrimeListFragment.java

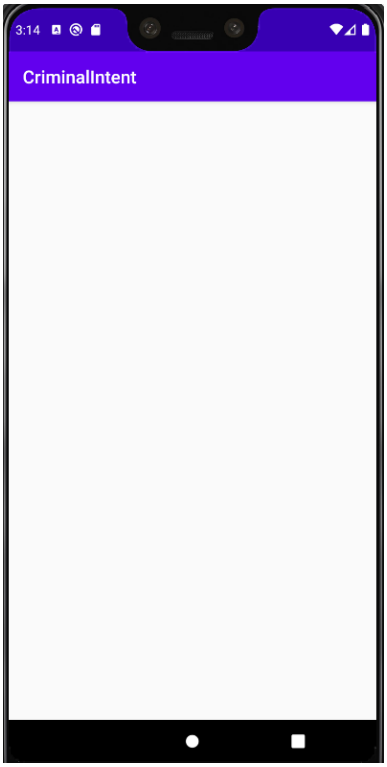
```java
public class CrimeListFragment extends Fragment {
    private static final String TAG = "CrimeListFragment";

    private CrimeListViewModel mCrimeListViewModel;

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        mCrimeListViewModel = new ViewModelProvider(this).get(CrimeListViewModel.class);
        Log.d(TAG, "Total crimes: " + mCrimeListViewModel.getCrimes().size());
    }

    public static CrimeListFragment newInstance(){
        return new CrimeListFragment();
    }
}
```

8. For now, `CrimeListFragment` is an empty shell of a fragment. It logs the number of crimes found in `CrimeListViewModel`.
9. One good practice to follow is to provide a `newInstance(…)` function that your activities can call to get an instance of your fragment. This is similar to the `newIntent()` function you used in GeoQuiz.
10. you learned about the `ViewModel` lifecycle when used with an activity. This lifecycle is slightly different when the `ViewModel` is used with a fragment. It still only has two states, created or destroyed/nonexistent, but it is now tied to the lifecycle of the fragment instead of the activity.
11. The `ViewModel` will remain active as long as the fragment's view is onscreen. The `ViewModel` will persist across rotation (even though the fragment instance will not) and be accessible to the new fragment instance.
12. The `ViewModel` will be destroyed when the fragment is destroyed. This can happen when the user presses the Back button to dismiss the screen. It can also happen if the hosting activity replaces the fragment with a different one. Even though the same activity is on the screen, both the fragment and its associated `ViewModel` will be destroyed, since they are no longer needed.
13. One special case is when you add the fragment transaction to the back stack. When the activity replaces the current fragment with a different one, if the transaction is added to the back stack, the fragment instance and its `ViewModel` will not be destroyed. This maintains your state: If the user presses the Back button, the fragment transaction is reversed. The original fragment instance is put back on the screen, and all the data in the `ViewModel` is preserved.
14. Next, update `MainActivity` to host an instance of `CrimeListFragment` instead of `CrimeFragment`.

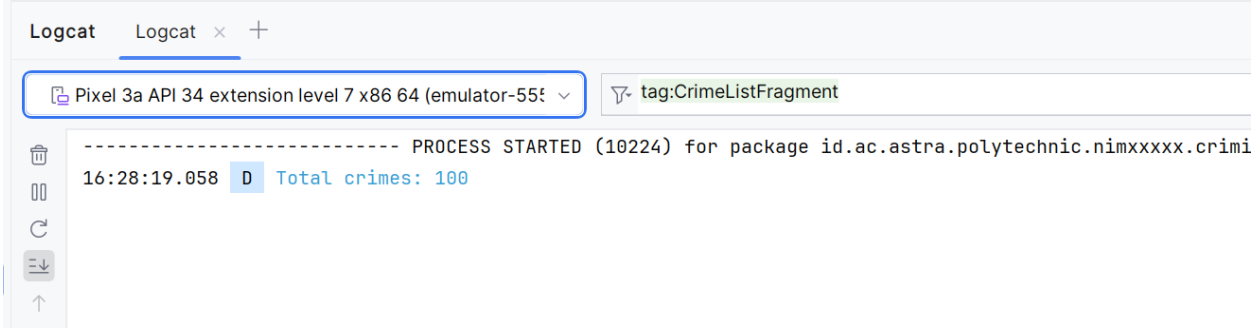
activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.fragment.app.FragmentContainerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/fragment_container"
    android:name="id.ac.astra.polytechnic.nimxxxxx.criminalintent.CrimeListFragment"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"/>
```

15. For now, you have hardcoded `MainActivity` to always display a `CrimeListFragment`. Later you will update `MainActivity` to swap out `CrimeListFragment` and `CrimeFragment` on demand as the user navigates through the app.
16. Run CriminalIntent, and you will see `MainActivity`'s `FrameLayout` hosting an empty `CrimeListFragment`



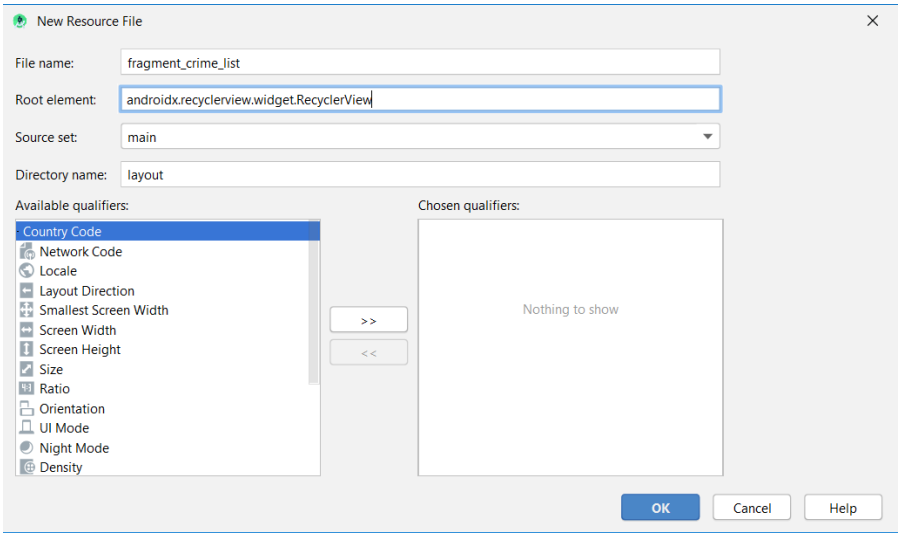
17. Search the Logcat output for `CrimeListFragment`. You will see a log statement showing the total number of crimes available:



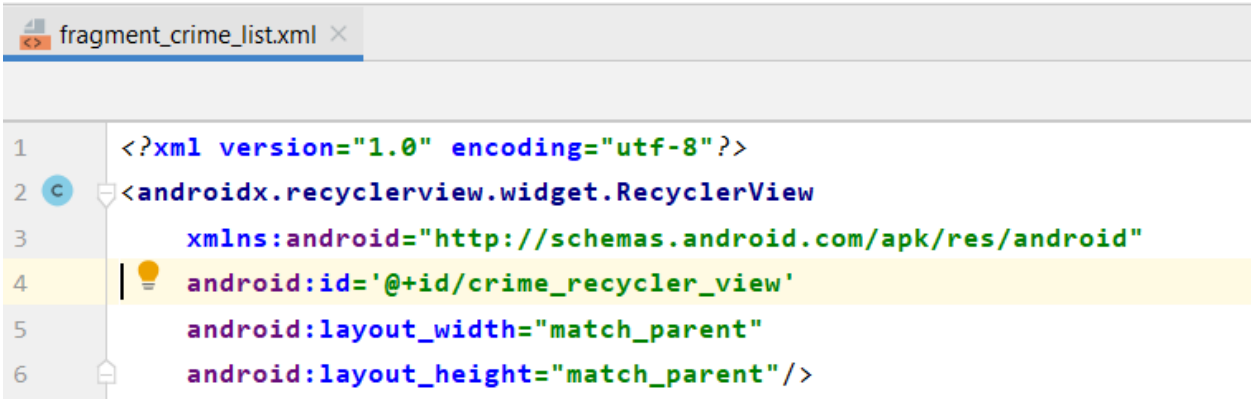
Adding a RecyclerView

18. You want `CrimeListFragment` to display a list of crimes to the user. To do this you will use a `RecyclerView`.
19. The `RecyclerView` class lives in another Jetpack library. The first step to using a `RecyclerView` is to add the RecyclerView library as a dependency.

20. Again, sync your Gradle files before moving on.
21. Your `RecyclerView` will live in `CrimeListFragment`'s layout file. First, you must create the layout file. Create a new layout resource file named `fragment_crime_list`. For the `Root element`, specify `androidx.recyclerview.widget.RecyclerView`



22. In the new `layout/fragment_crime_list.xml` file, add an ID attribute to the `RecyclerView`. Collapse the close tag into the opening tag, since you will not add any children to the `RecyclerView`.

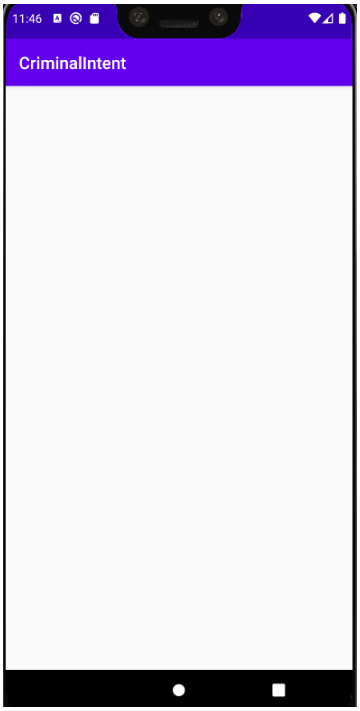


23. Now that `CrimeListFragment`'s view is set up, hook up the view to the fragment. Modify `CrimeListFragment` to use this layout file and to find the `RecyclerView` in the layout file

CrimeListFragment.java

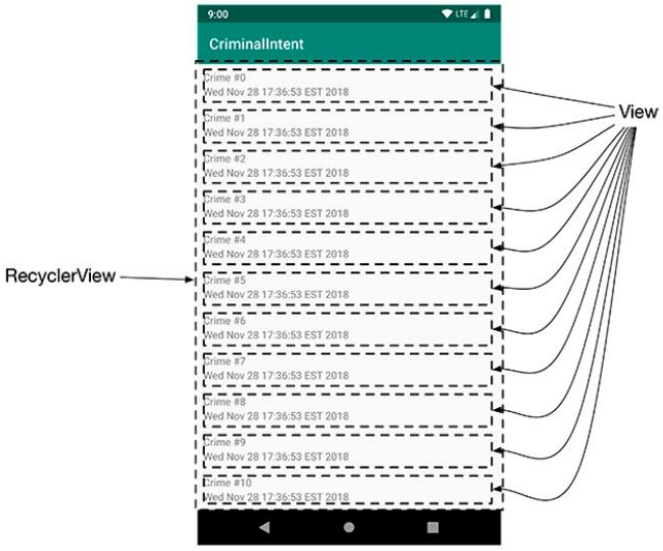
```
16  public class CrimeListFragment extends Fragment {
17      private static final String TAG = "CrimeListFragment";
18
19      private CrimeListViewModel mCrimeListViewModel;
20      private RecyclerView mCrimeRecyclerView;
21
22      public static CrimeListFragment newInstance(){
23          return new CrimeListFragment();
24      }
25
26      @Override
27      public void onCreate(Bundle savedInstanceState) {...}
34
35      @Override
36      public View onCreateView(LayoutInflater inflater,
37                               ViewGroup container,
38                               Bundle savedInstanceState) {
39          View view = inflater.inflate(R.layout.fragment_crime_list,
40                  container, false);
41          mCrimeRecyclerView = (RecyclerView) view.findViewById(R.id.crime_recycler_view);
42          mCrimeRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));
43          return view;
44      }
45  }
```

24. 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.
25. `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.
26. There are a few built-in `LayoutManagers` 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, you will use `GridLayoutManager` to arrange items in a grid instead.
27. Run the app. You should again see a blank screen, but now you are looking at an empty `RecyclerView`.

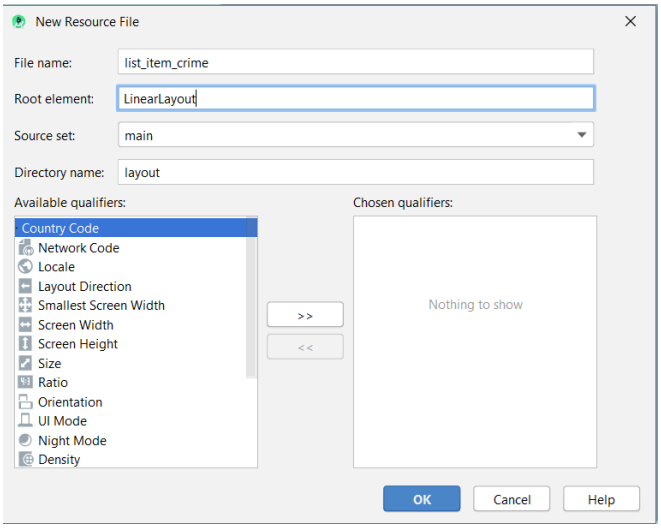


Creating an Item View Layout

28. `RecyclerView` is a subclass of `ViewGroup`. It displays a list of child `View` objects, called `item views`. Each item view represents a single object from the list of data backing the recycler view (in your case, a single crime from the crime list). Depending on the complexity of what you need to display, these child `Views` can be complex or very simple.
29. For your first implementation, each item in the list will display the title and date of a `Crime`, as shown



30. Each item displayed on the `RecyclerView` will have its own view hierarchy, exactly the way `CrimeFragment` has a view hierarchy for the entire screen. Specifically, the `View` object on each row will be a `LinearLayout` containing two `TextViews`.
31. 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`. Name the file `list_item_crime`, set the root element to `LinearLayout`, and click `OK`.

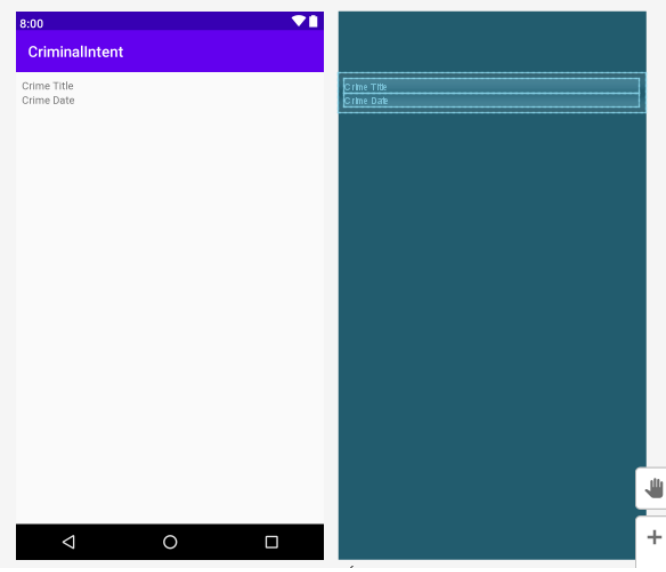


32. Update your layout file to add padding to the `LinearLayout` and to add the two `TextViews`, as shown

list_item_crime.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height='wrap_content'
    android:padding='8dp'>

    <TextView
        android:id='@+id/crime_title'
        android:layout_width='match_parent'
        android:layout_height='wrap_content'
        android:text='Crime Title'/>

    <TextView
        android:id='@+id/crime_date'
        android:layout_width='match_parent'
        android:layout_height='wrap_content'
        android:text='Crime Date'/>

</LinearLayout>
```

33. 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

34. The `RecyclerView` expects an item view to be wrapped in an instance of `ViewHolder`. A `ViewHolder` stores a reference to an item's view (and sometimes references to specific widgets within that view).
35. Define a view holder by adding an inner class in `CrimeListFragment` that extends from `RecyclerView.ViewHolder`.

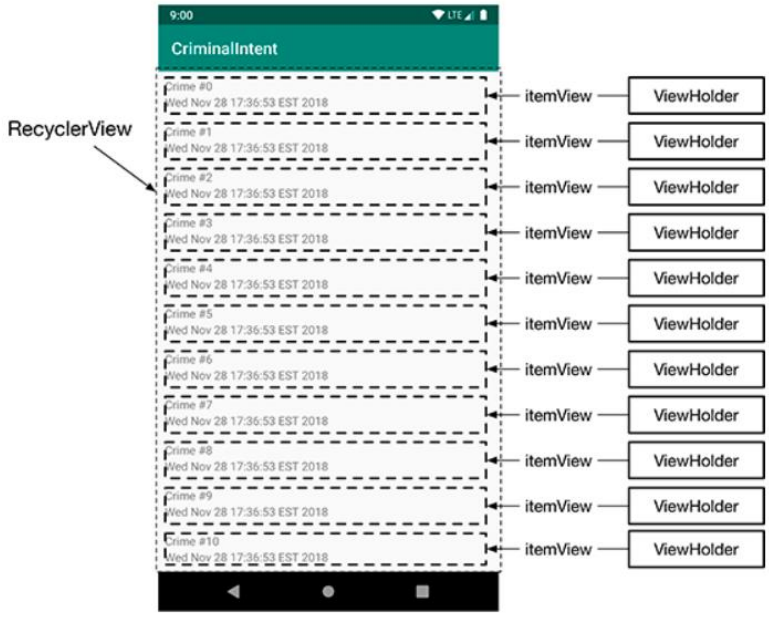
CrimeListFragment.java

```
16  public class CrimeListFragment extends Fragment {
```

...

```
35      @Override
36  @   public View onCreateView(LayoutInflater inflater,
37                               ViewGroup container,
38                               Bundle savedInstanceState) {...}
45
46      private class CrimeHolder extends RecyclerView.ViewHolder{
47
48  @       public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
49              super(inflater.inflate(R.layout.list_item_crime, parent, false));
50          }
51      }
52  }
```

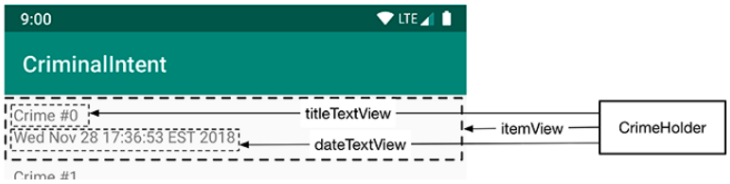
36. A `RecyclerView` never creates `View`s by themselves. It always creates `ViewHolder`s, which bring their `itemView`s along for the ride



37. When the `View` for each item 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. (For example, you do not need to search through the item view hierarchy to get a handle to the title text view every time you need to set the title.)
38. Update `CrimeHolder` to find the title and date text views in `itemView`'s hierarchy when an instance is first created. Store references to the text views in properties.

```
47      private class CrimeHolder extends RecyclerView.ViewHolder{
48
49          private TextView mTitleTextView;
50          private TextView mDateTextView;
51
52  @       public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
53              super(inflater.inflate(R.layout.list_item_crime, parent, false));
54
55              mTitleTextView = (TextView) itemView.findViewById(R.id.crime_title);
56              mDateTextView = (TextView) itemView.findViewById(R.id.crime_date);
57          }
58      }
59  }
```

39. The updated view holder now stashes references to the title and date text views so you can easily change the value displayed later without having to go back through the item's view hierarchy



Implementing an Adapter to Populate the RecyclerView

40. `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 when asked
- binding `ViewHolder`s to data from the model layer when asked

41. The recycler view is responsible for:

- asking the adapter to create a new `ViewHolder`
- asking the adapter to bind a `ViewHolder` to the item from the backing data at a given position

42. Time to create your adapter. Add a new inner class named `CrimeAdapter` to `CrimeListFragment`. Add a primary constructor that expects a list of crimes as input and stores the crime list passed in a field member, as shown.
43. In your new `CrimeAdapter`, you are also going to override three functions: `onCreateViewHolder(…)`, `onBindViewHolder(…)`, and `getItemCount()`. To save you

typing (and typos), Android Studio can generate these overrides for you. Once you have typed the initial line of the new code, put your cursor on `CrimeAdapter` and press (Alt-Enter). Select `Implement members` from the pop-up. In the `Implement members` dialog, select all three function names and click `OK`. Then you only need to fill in the bodies as shown.

CrimeListFragment.java

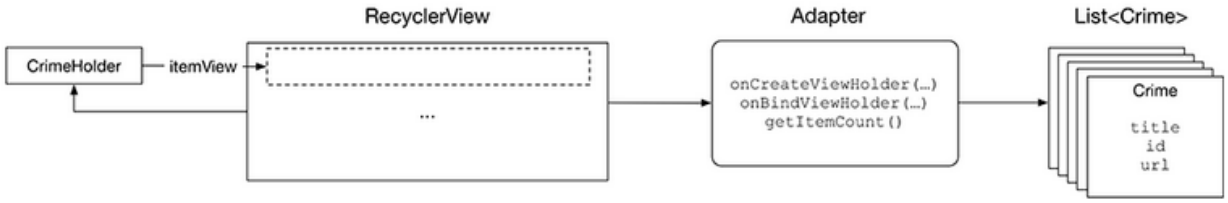
```
36      private class CrimeHolder extends RecyclerView.ViewHolder{...}
48
49      private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder>{
50
51          private List<Crime> mCrimes;
52
53          public CrimeAdapter(List<Crime> crimes){
54              mCrimes = crimes;
55          }
56
57          @Override
58          public CrimeHolder onCreateViewHolder(ViewGroup parent, int viewType) {
59              LayoutInflater layoutInflater = LayoutInflater.from(getActivity());
60
61              return new CrimeHolder(layoutInflater, parent);
62          }
63
64          @Override
65          public void onBindViewHolder(CrimeHolder holder, int position) {
66
67          }
68
69          @Override
70          public int getItemCount() {
71              return mCrimes.size();
72          }
73      }
74  }
```

44. `Adapter.onCreateViewHolder(…)` is responsible for creating a view to display, wrapping the view in a view holder, and returning the result. In this case, you inflate `list_item_view.xml` and pass the resulting view to a new instance of `CrimeHolder`. (For now, you can ignore `onCreateViewHolder(…)`'s parameters. You only need these values if you are doing something fancy, like displaying different types of views within the same recycler view.

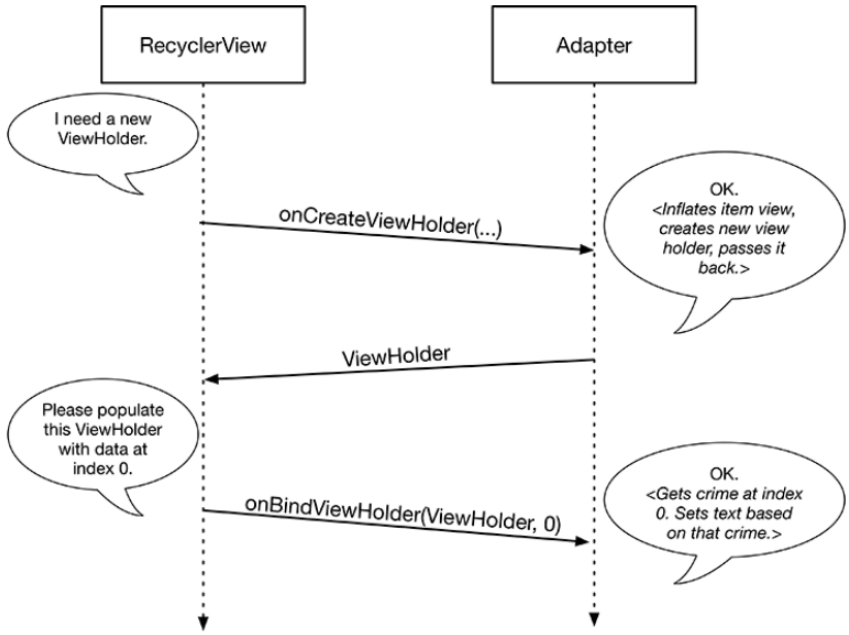
45. `Adapter.onBindViewHolder(holder: CrimeHolder, position: Int)` is responsible for populating a given `holder` with the crime from a given `position`. In this case, you get the crime from the crime list at the requested position. You then use the title and data from that crime to set the text in the corresponding text views.
46. When the recycler view needs to know how many items are in the data set backing it (such as when the recycler view first spins up), it will ask its adapter by calling

`Adapter.getItemCount()`. Here, `getItemCount()` returns the number of items in the list of crimes to answer the recycler view's request.

47. The `RecyclerView` itself does not know anything about the `Crime` object or the list of `Crime` objects to be displayed. Instead, the `CrimeAdapter` knows all of a `Crime`'s intimate and personal details. The adapter also knows about the list of crimes that backs the recycler view



48. When the `RecyclerView` needs a view object to display, it will have a conversation with its adapter.



49. The `RecyclerView` calls the adapter's `onCreateViewHolder(ViewGroup, Int)` function to create a new `ViewHolder`, along with its juicy payload: a `View` to display. The `ViewHolder` (and its `itemView`) that the adapter creates and hands back to the `RecyclerView` has not yet been populated with data from a specific item in the data set.
50. Next, the `RecyclerView` calls `onBindViewHolder(ViewHolder, Int)`, passing a `ViewHolder` into this function 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.
51. After this process is complete, `RecyclerView` will place a list item on the screen.

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

CrimeListFragment.java

```
19  public class CrimeListFragment extends Fragment {
20      private static final String TAG = "CrimeListFragment";
21
22      private CrimeListViewModel mCrimeListViewModel;
23      private RecyclerView mCrimeRecyclerView;
24      private CrimeAdapter mAdapter;
25
26      private void updateUI(){
27          List<Crime> crimes = mCrimeListViewModel.getCrimes();
28          mAdapter = new CrimeAdapter(crimes);
29          mCrimeRecyclerView.setAdapter(mAdapter);
30      }
31
```

...

```
32      @Override
33      public void onCreate(Bundle savedInstanceState) {...}
39
40      @Override
41      public View onCreateView(LayoutInflater inflater,
42                               ViewGroup container,
43                               Bundle savedInstanceState) {
44          View view = inflater.inflate(R.layout.fragment_crime_list,
45                  container, false);
46          mCrimeRecyclerView = (RecyclerView) view.findViewById(R.id.crime_recycler_view);
47          mCrimeRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));
48          updateUI();
49          return view;
50      }
```

53. Run CriminalIntent and scroll through your new `RecyclerView`
54. Swipe or drag down and you will see even more views scroll across your screen.

55. 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.

Cleaning Up Binding List Items

56. We recommend you place all the code that will do the real work of binding inside your `CrimeHolder`. First, add a property to stash the `Crime` being bound. While you are at it, make the existing text view properties private. Add a `bind(Crime)` function to `CrimeHolder`. In this new function, cache the crime being bound into a property and set the text values on `titleTextView` and `dateTextView`.

CrimeListFragment.java

```
55
56      private class CrimeHolder extends RecyclerView.ViewHolder{
57
58          private TextView mTitleTextView;
59          private TextView mDateTextView;
60
61          private Crime mCrime;
62
63  @       public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
64              super(inflater.inflate(R.layout.list_item_crime, parent, false));
65
66              mTitleTextView = (TextView) itemView.findViewById(R.id.crime_title);
67              mDateTextView = (TextView) itemView.findViewById(R.id.crime_date);
68          }
69
70          public void bind(Crime crime){
71              mCrime = crime;
72              mTitleTextView.setText(mCrime.getTitle());
73              mDateTextView.setText(mCrime.getDate().toString());
74          }
75      }
```

57. When given a `Crime` to bind, `CrimeHolder` will now update the title `TextView` and date `TextView` to reflect the state of the `Crime`.
58. Next, call your newly minted `bind(Crime)` function each time the `RecyclerView` requests that a given `CrimeHolder` be bound to a particular crime.

CrimeListFragment.java

```
77      private class CrimeAdapter extends RecyclerView.Adapter<CrimeHolder>{
78
79          private List<Crime> mCrimes;
80
81          public CrimeAdapter(List<Crime> crimes) { mCrimes = crimes; }
84
85          @Override
86          public CrimeHolder onCreateViewHolder(ViewGroup parent, int viewType) {
87              LayoutInflater layoutInflater = LayoutInflater.from(getActivity());
88
89              return new CrimeHolder(layoutInflater, parent);
90          }
91
92          @Override
93          public void onBindViewHolder(CrimeHolder holder, int position) {
94              Crime crime = mCrimes.get(position);
95              holder.bind(crime);
96          }
97
98          @Override
99          public int getItemCount() { return mCrimes.size(); }
102     }
```

59. Run CriminalIntent one more time

Responding to Presses

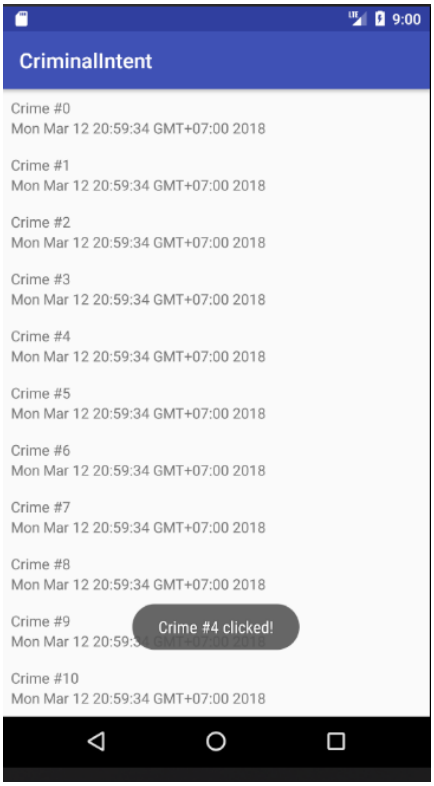
60. As icing on the **RecyclerView** cake, CriminalIntent should also respond to a press on these list items.
61. For now, show a **Toast** when the user takes action on a **Crime**.
62. As you may have noticed, **RecyclerView**, while powerful and capable, has precious few real responsibilities.
63. 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.
64. Instead, you can handle them like you normally do: by setting an **OnClickListener**.
65. Since each **View** has an associated **ViewHolder**, you can make your **ViewHolder** the **OnClickListener** for its **View**.
66. Modify the **CrimeHolder** to handle presses for the entire row.

CrimeListFragment.java

```
57      private class CrimeHolder extends RecyclerView.ViewHolder
58              implements View.OnClickListener{
59
60          private TextView mTitleTextView;
61          private TextView mDateTextView;
62
63          private Crime mCrime;
64
65          public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {
66              super(inflater.inflate(R.layout.list_item_crime, parent, false));
67              itemView.setOnClickListener(this);
68
69              mTitleTextView = (TextView) itemView.findViewById(R.id.crime_title);
70              mDateTextView = (TextView) itemView.findViewById(R.id.crime_date);
71          }
72
73          public void bind(Crime crime){
74              mCrime = crime;
75              mTitleTextView.setText(mCrime.getTitle());
76              mDateTextView.setText(mCrime.getDate().toString());
77          }
78
79          @Override
80          public void onClick(View view) {
81              Toast.makeText(getActivity(),
82                      mCrime.getTitle() + " clicked!", Toast.LENGTH_SHORT)
83                      .show();
84          }
85      }
```

67. **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.

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



Notes:

1. Create folder PRG6_M4_P2_XXXXXXXXXX.
2. Zip the folder and submit it to the server.

Bibliography:

- Marsicano, et. al., "Android Programming – The Big Nerd Ranch", 5th Ed, 2022, Pearson Technology.