Level/Subject	:	**D3/Programming 6**
Topic	:	Database & Navigation in Android apps
Week	:	5
Activity	:	Using Database and Navigation in Android apps using Android Studio
Alocated time	:	120 mins labs
Deliverables	:	Project folder
Due date	:	end of week

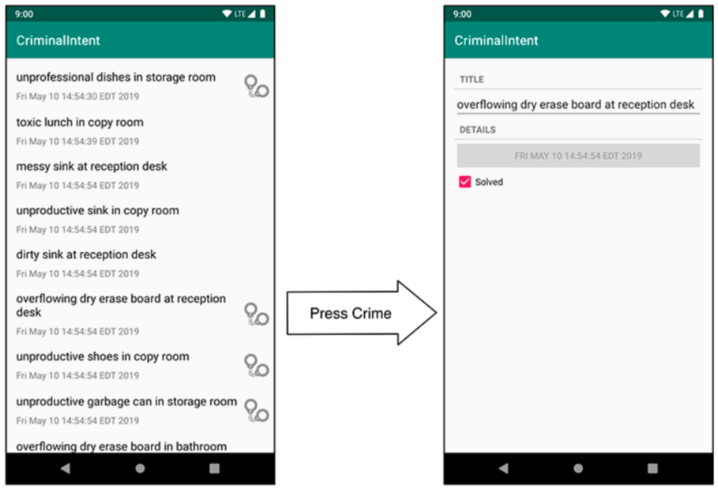
Competency :

Student expected to be able to create Android apps with SQLite database and Room library and Navigation using Android Studio IDE.

Example Practice Task:

Create Android apps with database & navigation using Android Studio.

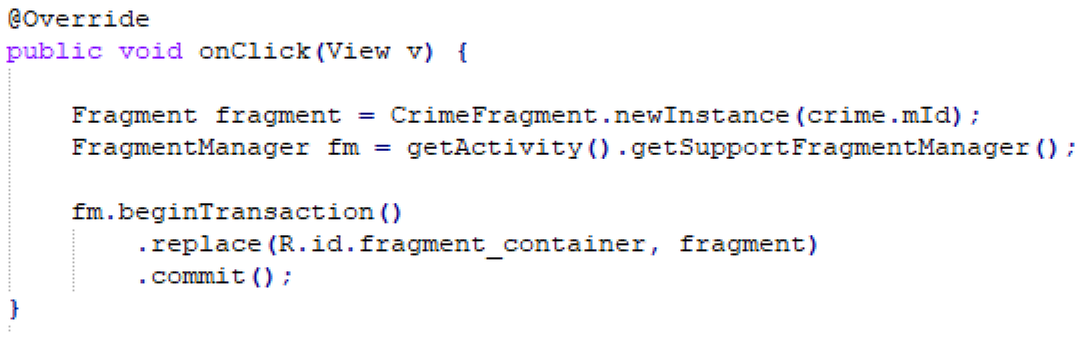
1. In this practice, you will get the list and the detail parts of CriminalIntent working together. When a user presses an item in the list of crimes, `MainActivity` will swap out `CrimeListFragment` with a new instance of `CrimeFragment`. `CrimeFragment` will display the details for the crime that was pressed


9:00
CriminalIntent
unprofessional dishes in storage room
Fri May 10 14:54:30 EDT 2019
toxic lunch in copy room
Fri May 10 14:54:39 EDT 2019
messy sink at reception desk
Fri May 10 14:54:54 EDT 2019
unproductive sink in copy room
Fri May 10 14:54:54 EDT 2019
dirty sink at reception desk
Fri May 10 14:54:54 EDT 2019
overflowing dry erase board at reception desk
Fri May 10 14:54:54 EDT 2019
unproductive shoes in copy room
Fri May 10 14:54:54 EDT 2019
unproductive garbage can in storage room
Fri May 10 14:54:54 EDT 2019
overflowing dry erase board in bathroom
Press Crime
9:00
CriminalIntent
TITLE
overflowing dry erase board at reception desk
DETAILS
FRI MAY 10 14:54:54 EDT 2019
Solved

2. To get this working you will learn how to implement navigation by having the hosting activity change out fragments in response to user actions. You will also learn how to pass data to a fragment instance using `fragment arguments`. Finally, you will learn how to use `LiveData transformations` to load immutable data in response to UI changes.

Single Activity: Fragment Boss

3. In **GeoQuiz**, you had one activity (`MainActivity`) start another activity (`CheatActivity`). In **CriminalIntent**, you are instead going to use a `single activity architecture`. An app that uses single activity architecture has one activity and multiple fragments. The activity's job is to swap fragments in and out in response to user events.
4. To implement the navigation from `CrimeListFragment` to `CrimeFragment` in response to the user pressing on a crime in the list, you might think to initiate a fragment transaction on the hosting activity's fragment manager in `CrimeListFragment`'s `CrimeHolder.onClick(View)`. This `onClick(View)` would get `MainActivity`'s `FragmentManager` and commit a fragment transaction that replaces `CrimeListFragment` with `CrimeFragment`.
5. The code in your `CrimeListFragment.CrimeHolder` would look like this:

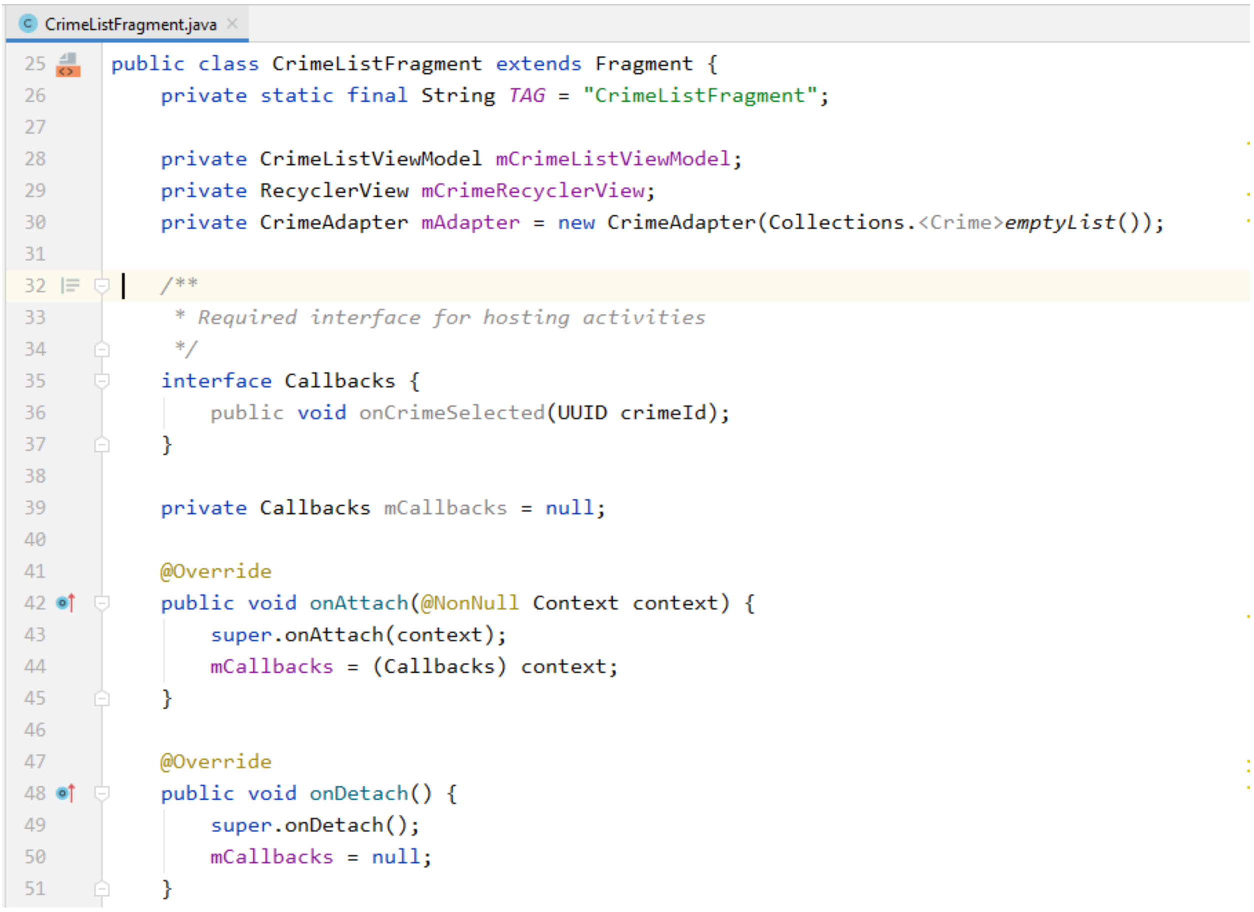


```
@Override
public void onClick(View v) {

    Fragment fragment = CrimeFragment.newInstance(crime.mId);
    FragmentManager fm = getActivity().getSupportFragmentManager();

    fm.beginTransaction()
        .replace(R.id.fragment_container, fragment)
        .commit();
}
```

6. This works, but it is not how stylish Android programmers do things. Fragments are intended to be standalone, composable units. If you write a fragment that adds fragments to the activity's `FragmentManager`, then that fragment is making assumptions about how the hosting activity works, and your fragment is no longer a standalone, composable unit.
7. For example, in the code above, `CrimeListFragment` adds a `CrimeFragment` to `MainActivity` and assumes that `MainActivity` has a `fragment_container` in its layout. This is business that should be handled by `CrimeListFragment`'s hosting activity instead of `CrimeListFragment`.
8. To maintain the independence of your fragments, you will **delegate work back to the hosting activity by defining callback interfaces** in your fragments. The hosting activities will implement these interfaces to perform fragment-bossing duties and layout-dependent behavior.

Fragment callback interfaces

9. To delegate functionality back to the hosting activity, a fragment typically defines a custom callback interface named `Callbacks`. This interface defines work that the fragment needs done by its boss, the hosting activity. Any activity that will host the fragment must implement this interface.
10. With a callback interface, a fragment is able to call methods on its hosting activity without having to know anything about which activity is hosting it.
11. Use a callback interface to delegate on-click events from `CrimeListFragment` back to its hosting activity. First, open `CrimeListFragment` and define a `Callbacks` interface with a single callback method. Add a `mCallbacks` member variable to hold an object that implements `Callbacks`. Override `onAttach(Context)` and `onDetach()` to set and unset the `mCallbacks` member variable.

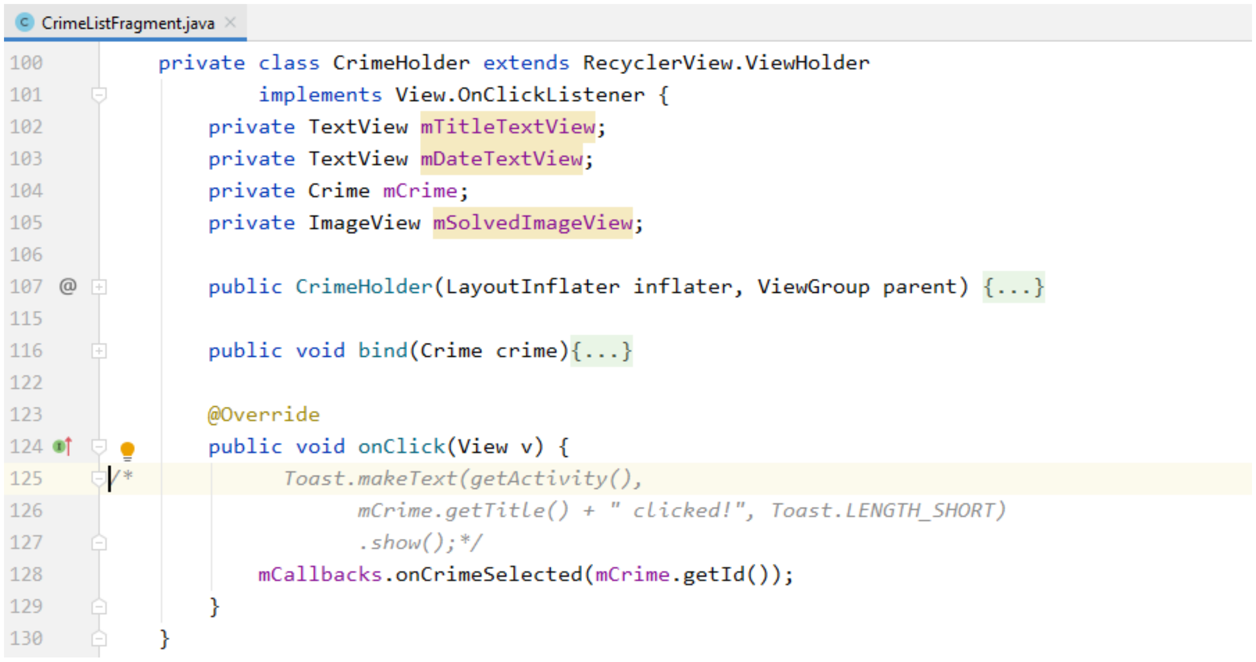


CrimeListFragment.java

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

    private CrimeListViewModel mCrimeListViewModel;
    private RecyclerView mCrimeRecyclerView;
    private CrimeAdapter mAdapter = new CrimeAdapter(Collections.<Crime>emptyList());

    /**
     * Required interface for hosting activities
     */
    interface Callbacks {
        public void onCrimeSelected(UUID crimeId);
    }

    private Callbacks mCallbacks = null;

    @Override
    public void onAttach(@NonNull Context context) {
        super.onAttach(context);
        mCallbacks = (Callbacks) context;
    }

    @Override
    public void onDetach() {
        super.onDetach();
        mCallbacks = null;
    }
```

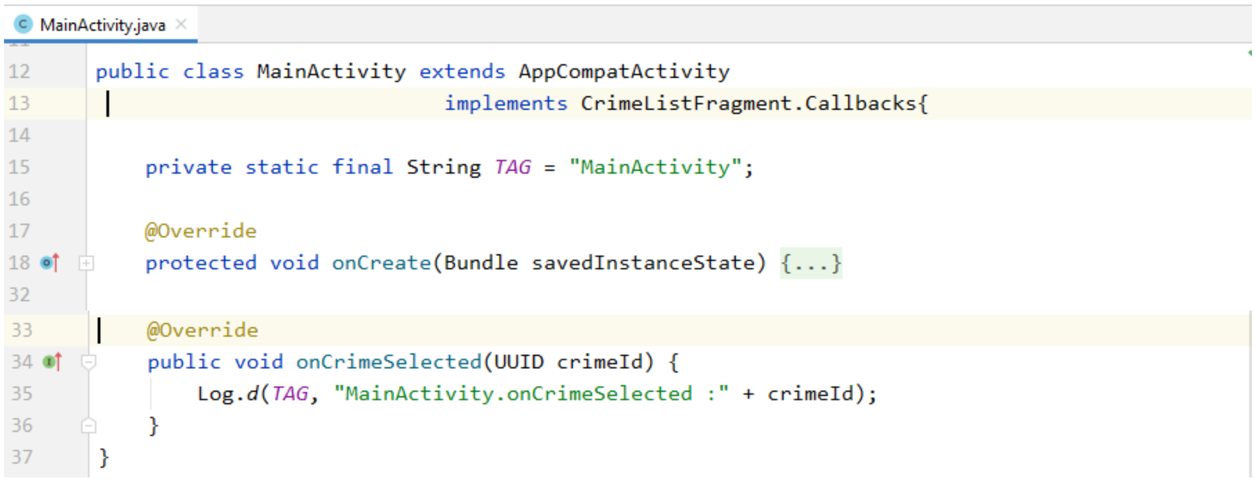
12. The `Fragment.onAttach(Context)` lifecycle method is called when a fragment is attached to an activity. Here you stash the `Context` argument passed to `onAttach(…)` in your `callbacks` property. Since `CrimeListFragment` is hosted in an activity, the `Context` object passed to `onAttach(…)` is the activity instance hosting the fragment.

13. Remember, `Activity` is a subclass of `Context`, so `onAttach(…)` passes a `Context` as a parameter, which is more flexible. Ensure that you use the `onAttach(Context)` signature for `onAttach(…)` and not the deprecated `onAttach(Activity)` method, which may be removed in future versions of the API.
14. Similarly, you set the variable to `null` in the corresponding waning lifecycle method, `Fragment.onDetach()`. You set the variable to `null` here because afterward you cannot access the activity or count on the activity continuing to exist.
15. Note that `CrimeListFragment` performs an unchecked cast of its activity to `CrimeListFragment.Callbacks`. This means that the hosting activity `must` implement `CrimeListFragment.Callbacks`. That is not a bad dependency to have, but it is important to document it.
16. Now `CrimeListFragment` has a way to call methods on its hosting activity. It does not matter which activity is doing the hosting. As long as the activity implements `CrimeListFragment.Callbacks`, everything in `CrimeListFragment` can work the same.
17. Next, update the click listener for individual items in the crime list so that pressing a crime notifies the hosting activity via the `Callbacks` interface. Call `onCrimeSelected(Crime)` in `CrimeHolder.onClick(View)`.


CrimeListFragment.java

```java
    private class CrimeHolder extends RecyclerView.ViewHolder
            implements View.OnClickListener {
        private TextView mTitleTextView;
        private TextView mDateTextView;
        private Crime mCrime;
        private ImageView mSolvedImageView;

        public CrimeHolder(LayoutInflater inflater, ViewGroup parent) {...}

        public void bind(Crime crime){...}

        @Override
        public void onClick(View v) {
            /*Toast.makeText(getActivity(),
                    mCrime.getTitle() + " clicked!", Toast.LENGTH_SHORT)
                    .show();*/
            mCallbacks.onCrimeSelected(mCrime.getId());
        }
    }
```

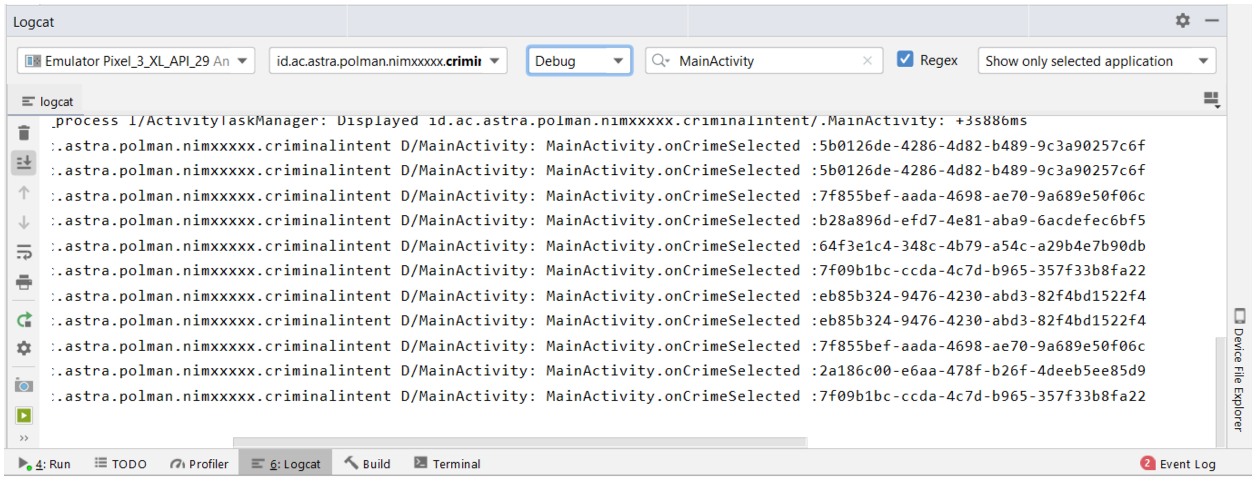
18. Finally, update `MainActivity` to implement `CrimeListFragment.Callbacks`. Log a debug statement in `onCrimeSelected(UUID)` for now.



MainActivity.java

```java
public class MainActivity extends AppCompatActivity
                            implements CrimeListFragment.Callbacks{

    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    @Override
    public void onCrimeSelected(UUID crimeId) {
        Log.d(TAG, "MainActivity.onCrimeSelected :" + crimeId);
    }
}
```

19. Run CriminalIntent. Search or filter Logcat to view `MainActivity`'s log statements. Each time you press a crime in the list you should see a log statement indicating the click event was propagated from `CrimeListFragment` to `MainActivity` through `Callbacks.onCrimeSelected(UUID)`.



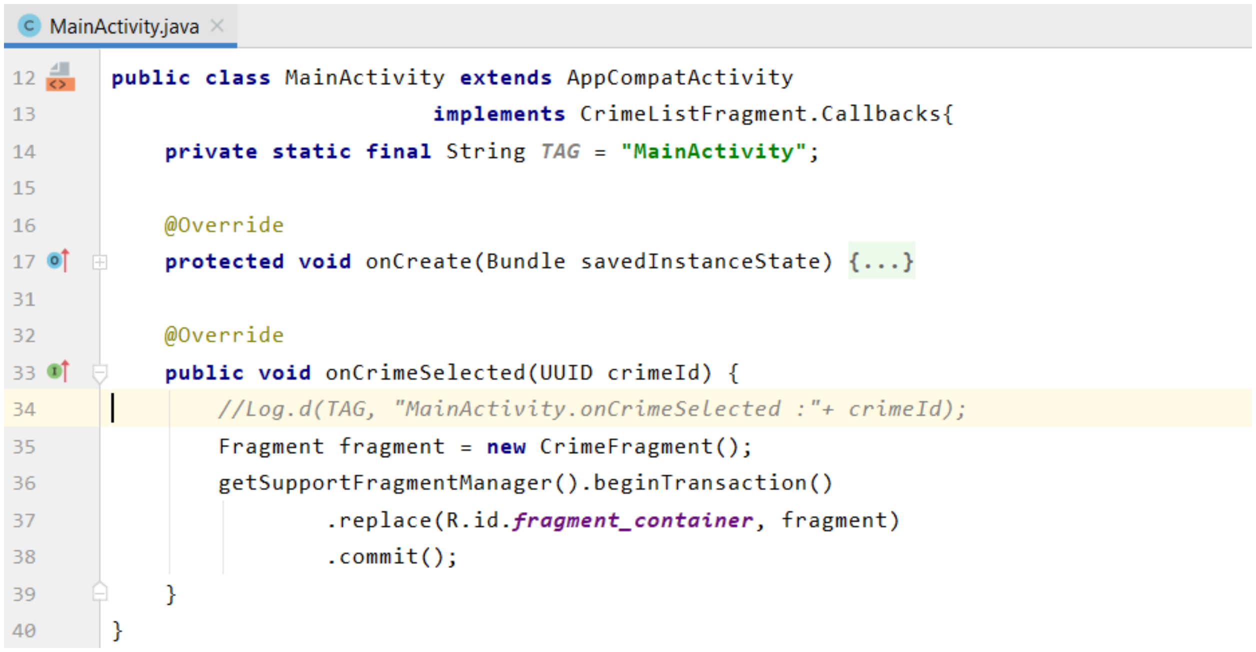
Logcat

Emulator Pixel_3_XL_API_29 An | id.ac.astra.polman.nimxxxxx.crimi | Debug | MainActivity | Regex ✓ | Show only selected application

```
_process I/ActivityTaskManager: Displayed id.ac.astra.polman.nimxxxxx.criminalintent/.MainActivity: +3s886ms
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :5b0126de-4286-4d82-b489-9c3a90257c6f
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :5b0126de-4286-4d82-b489-9c3a90257c6f
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :7f855bef-aada-4698-ae70-9a689e50f06c
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :b28a896d-efd7-4e81-aba9-6acdefec6bf5
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :64f3e1c4-348c-4b79-a54c-a29b4e7b90db
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :7f09b1bc-ccda-4c7d-b965-357f33b8fa22
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :eb85b324-9476-4230-abd3-82f4bd1522f4
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :eb85b324-9476-4230-abd3-82f4bd1522f4
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :7f855bef-aada-4698-ae70-9a689e50f06c
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :2a186c00-e6aa-478f-b26f-4deeb5ee85d9
:.astra.polman.nimxxxxx.criminalintent D/MainActivity: MainActivity.onCrimeSelected :7f09b1bc-ccda-4c7d-b965-357f33b8fa22
```

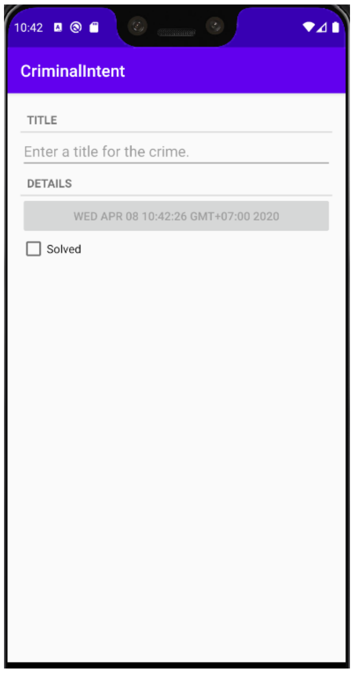
Replacing a fragment

20. Now that your callback interface is wired up correctly, update `MainActivity`'s `onCrimeSelected(UUID)` to swap out the `CrimeListFragment` with an instance of `CrimeFragment` when the user presses a crime in `CrimeListFragment`'s list. For now, ignore the crime ID passed to the callback.

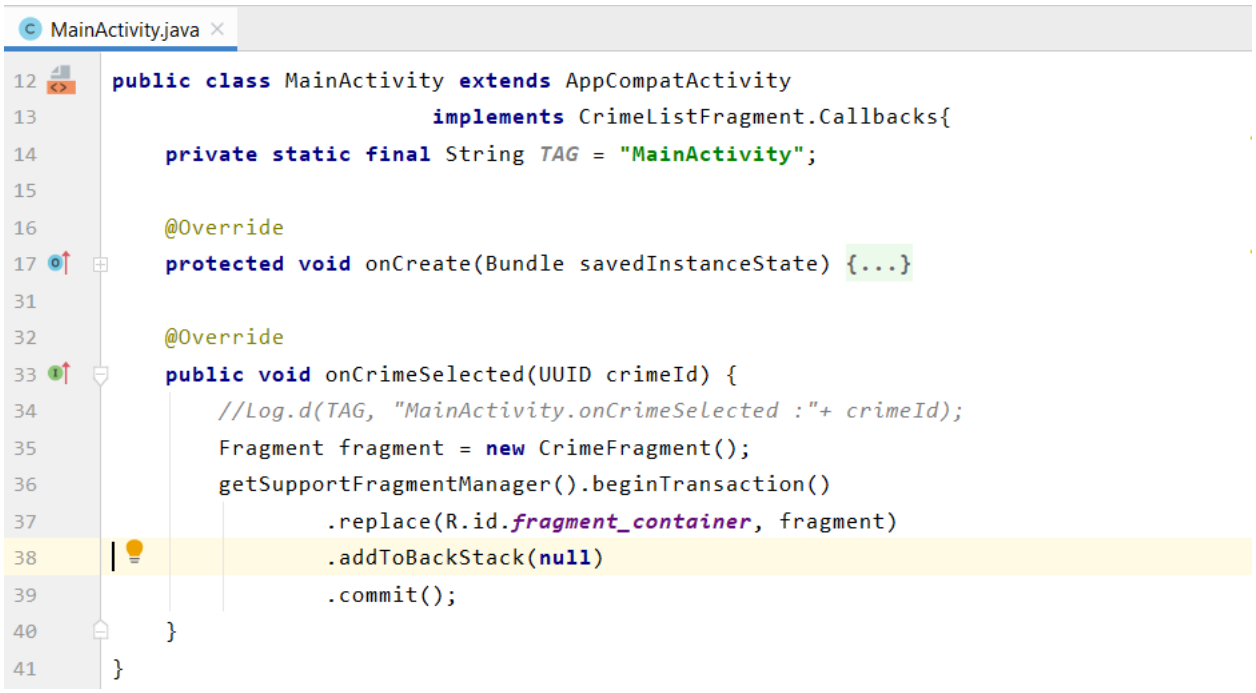

MainActivity.java

```
public class MainActivity extends AppCompatActivity
                        implements CrimeListFragment.Callbacks{
    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    @Override
    public void onCrimeSelected(UUID crimeId) {
        //Log.d(TAG, "MainActivity.onCrimeSelected :"+ crimeId);
        Fragment fragment = new CrimeFragment();
        getSupportFragmentManager().beginTransaction()
                .replace(R.id.fragment_container, fragment)
                .commit();
    }
}
```

21. `FragmentTransaction.replace(Int, Fragment)` replaces the fragment hosted in the activity (in the container with the integer resource ID specified) with the new fragment provided. If a fragment is not already hosted in the container specified, the new fragment is added, just as if you had called `FragmentTransaction.add(Int, fragment)`.
22. Run CriminalIntent. Press a crime in the list. You should see the crime detail screen appear


10:42
CriminalIntent
TITLE
Enter a title for the crime.
DETAILS
WED APR 08 10:42:26 GMT+07:00 2020
Solved

23. For now, the crime detail screen is empty, because you have not told `CrimeFragment` which `Crime` to display. You will populate the detail screen shortly. But first, you need to file down a remaining sharp edge in your navigation implementation.
24. Press the Back button. This dismisses `MainActivity`. This is because the only item in your app's back stack is the `MainActivity` instance that was launched when you launched the app.
25. Users will expect that pressing the Back button from the crime detail screen will bring them back to the crime list. To implement this behavior, add the replace transaction to the back stack.


MainActivity.java

```java
public class MainActivity extends AppCompatActivity
                         implements CrimeListFragment.Callbacks{
    private static final String TAG = "MainActivity";

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    @Override
    public void onCrimeSelected(UUID crimeId) {
        //Log.d(TAG, "MainActivity.onCrimeSelected :"+ crimeId);
        Fragment fragment = new CrimeFragment();
        getSupportFragmentManager().beginTransaction()
                .replace(R.id.fragment_container, fragment)
                .addToBackStack(null)
                .commit();
    }
}
```

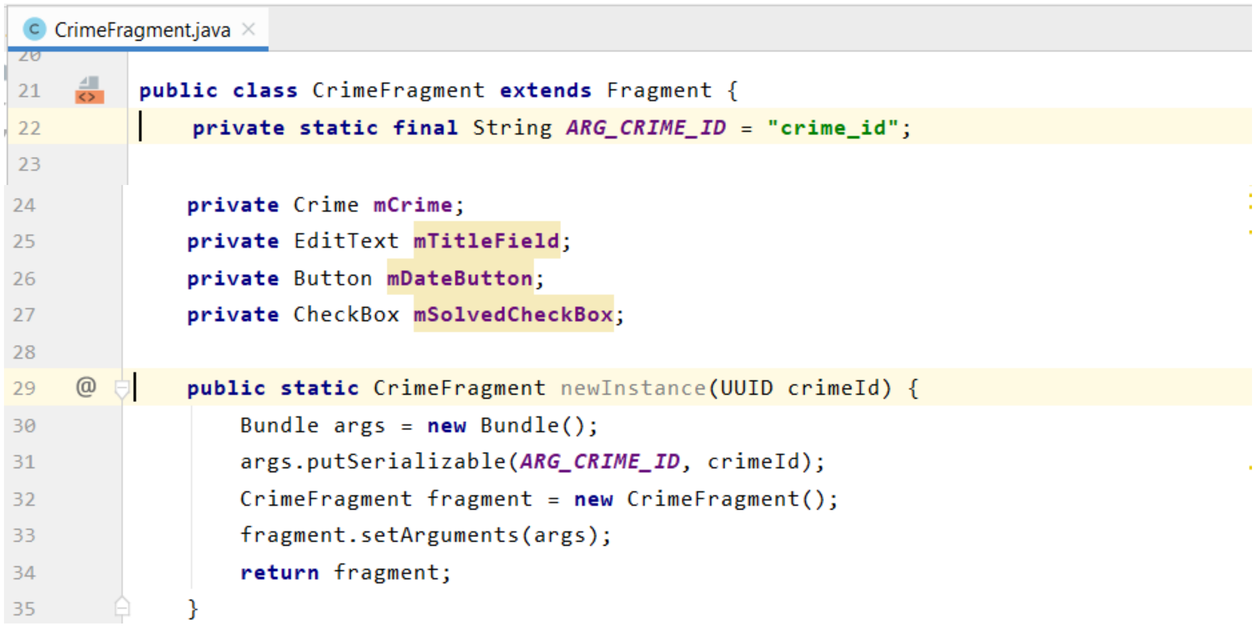
26. When you add a transaction to the back stack, this means that when the user presses the Back button the transaction will be reversed. So, in this case, `CrimeFragment` will be replaced with `CrimeListFragment`.
27. You can name the back stack state you are adding by passing a `String` to `FragmentTransaction.addToBackStack(String)`. Doing so is optional and, since you do not care about the name in this implementation, you pass `null`.
28. Run your app. Select a crime from the list to launch `CrimeFragment`. Press the Back button to go back to `CrimeListFragment`. Enjoy the simple pleasure of your app's navigation matching what users will expect.

Fragment Arguments

29. `CrimeListFragment` now notifies its hosting activity (`MainActivity`) when a crime is selected and passes along the ID of the selected crime.
30. That is all fine and dandy. But what you really need is a way to pass the selected crime ID from `MainActivity` to `CrimeFragment`. This way, `CrimeFragment` can pull data for that crime from the database and populate the UI with that data.
31. Fragment arguments offer a solution to this problem – they allow you to stash pieces of data someplace that belongs to the fragment. The "someplace" that belongs to a fragment is known as its `arguments bundle`. The fragment can retrieve data from the arguments bundle without relying on its parent activity or some other outside source.
32. Fragment arguments help you keep your fragment encapsulated. A well-encapsulated fragment is a reusable building block that can easily be hosted in any activity.
33. To create fragment arguments, you first create a `Bundle` object. This bundle contains key-value pairs that work just like the intent extras of an `Activity`. Each pair is known as an argument. Next, you use type-specific "put" methods of `Bundle` (similar to those of `Intent`) to add arguments to the bundle.
34. Every fragment instance can have a fragment arguments `Bundle` object attached to it.

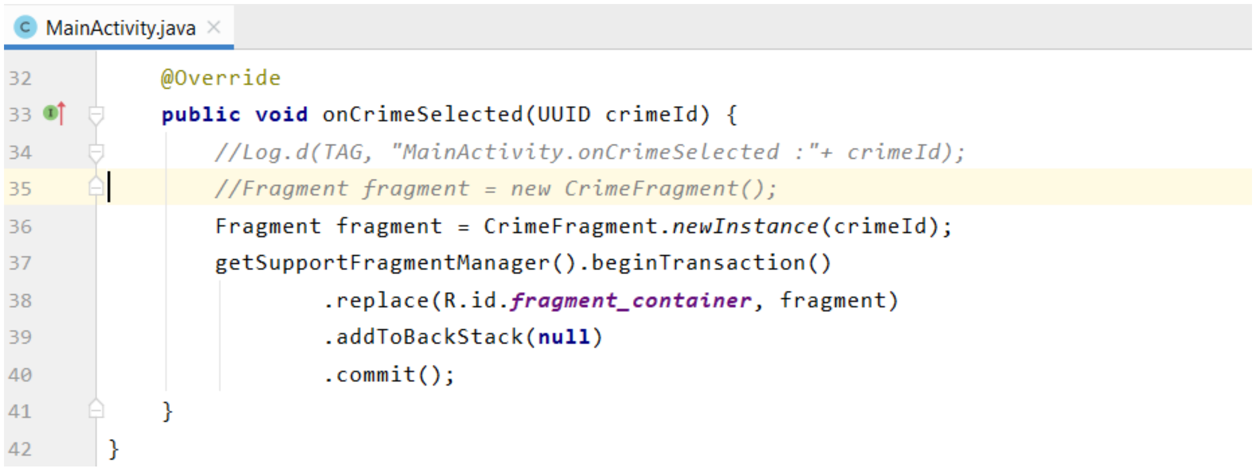
Attaching arguments to a fragment

35. 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.
36. To accomplish this, 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.
37. 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.
38. 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.



```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;
    }
```

39. Update `MainActivity` to call `CrimeFragment.newInstance(UUID)` when it needs to create a `CrimeFragment`. Pass along the `UUID` that was received in `MainActivity.onCrimeSelected(UUID)`.

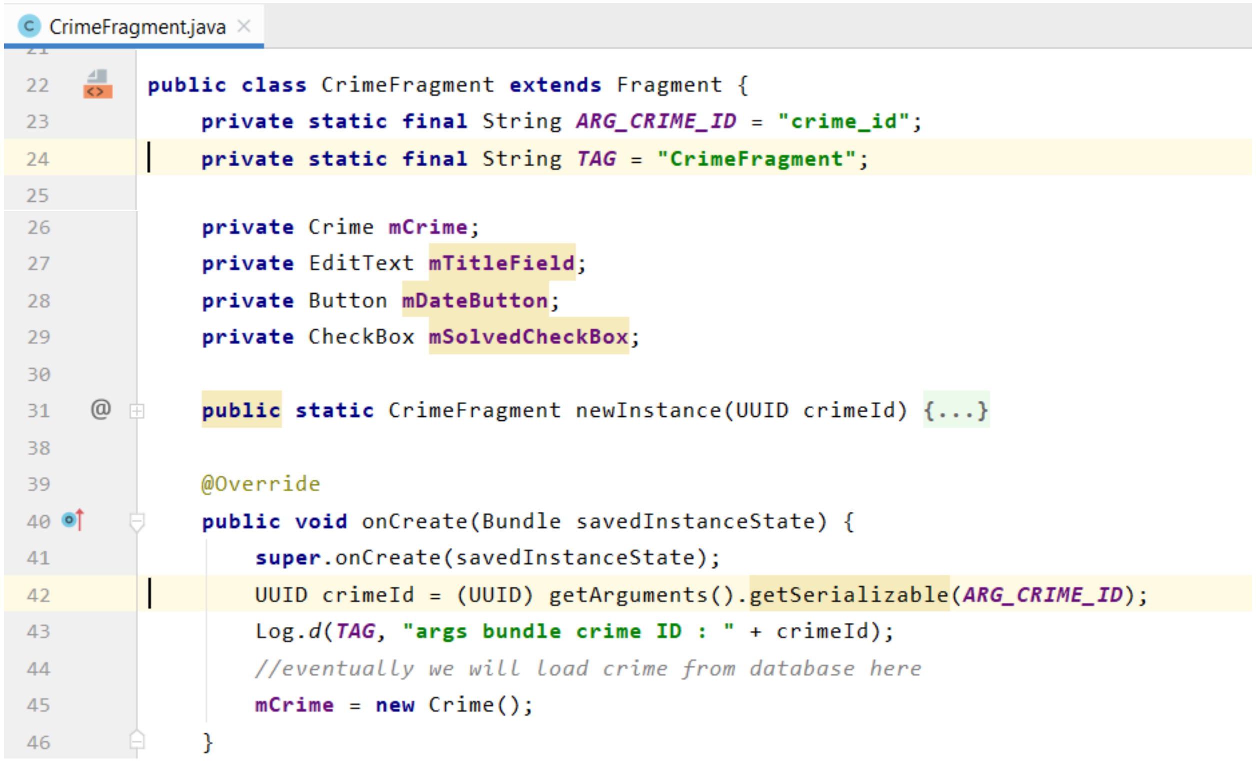


```java
    @Override
    public void onCrimeSelected(UUID crimeId) {
        //Log.d(TAG, "MainActivity.onCrimeSelected :"+ crimeId);
        //Fragment fragment = new CrimeFragment();
        Fragment fragment = CrimeFragment.newInstance(crimeId);
        getSupportFragmentManager().beginTransaction()
                .replace(R.id.fragment_container, fragment)
                .addToBackStack(null)
                .commit();
    }
}
```

40. Notice that the need for independence does not go both ways. `MainActivity` 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

41. To access a fragment's arguments, call the `Fragment` method named `getArguments`. Then use one of the type-specific "get" methods of `Bundle` to pull individual values out of the arguments bundle.
42. Back in `CrimeFragment.onCreate(…)`, retrieve the `UUID` from the fragment arguments. For now, log the ID so you can verify that the argument was attached as expected.



```java
public class CrimeFragment extends Fragment {
    private static final String ARG_CRIME_ID = "crime_id";
    private static final String TAG = "CrimeFragment";

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

    public static CrimeFragment newInstance(UUID crimeId) {...}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        UUID crimeId = (UUID) getArguments().getSerializable(ARG_CRIME_ID);
        Log.d(TAG, "args bundle crime ID : " + crimeId);
        //eventually we will load crime from database here
        mCrime = new Crime();
    }
```

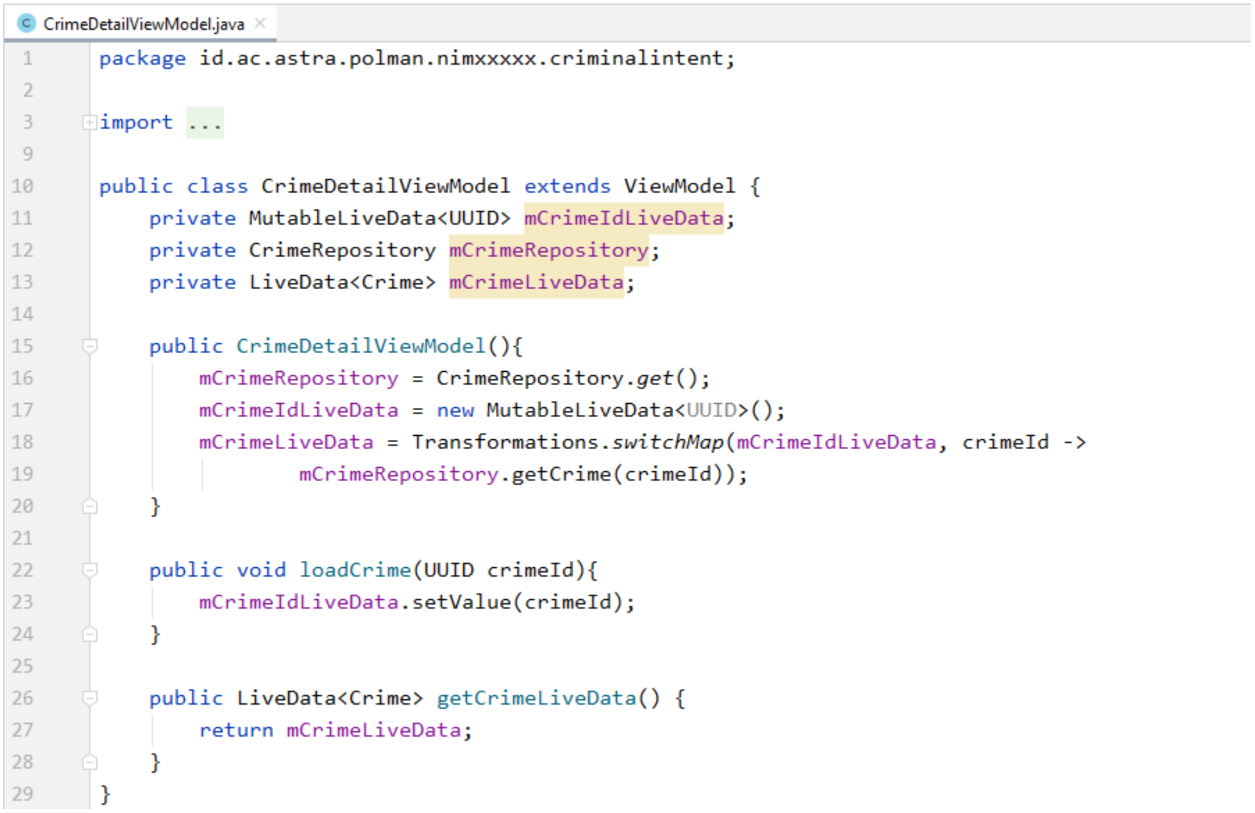
43. Run CriminalIntent. Click a crime on the list, and look at the LogCat. The app will behave the same, but you should feel all warm and fuzzy inside for passing the crime ID along to `CrimeFragment` while still maintaining `CrimeFragment`'s independence.

Using LiveData Transformations

44. Now that `CrimeFragment` has the crime ID, it needs to pull the crime object from the database so it can display the crime's data. Since this requires a database lookup that you do not want to repeat unnecessarily on rotation, add a `CrimeDetailViewModel` to manage the database query.
45. When `CrimeFragment` requests to load a crime with a given ID, its `CrimeDetailViewModel` should kick off a `getCrime(UUID)` database request.

When the request completes, `CrimeDetailViewModel` should notify `CrimeFragment` and pass along the crime object that resulted from the query.

46. Create a new class named `CrimeDetailViewModel` and expose a `LiveData` member variable to store and publish the `Crime` pulled from the database. Use `LiveData` to implement a relationship where changing the crime ID triggers a new database query.

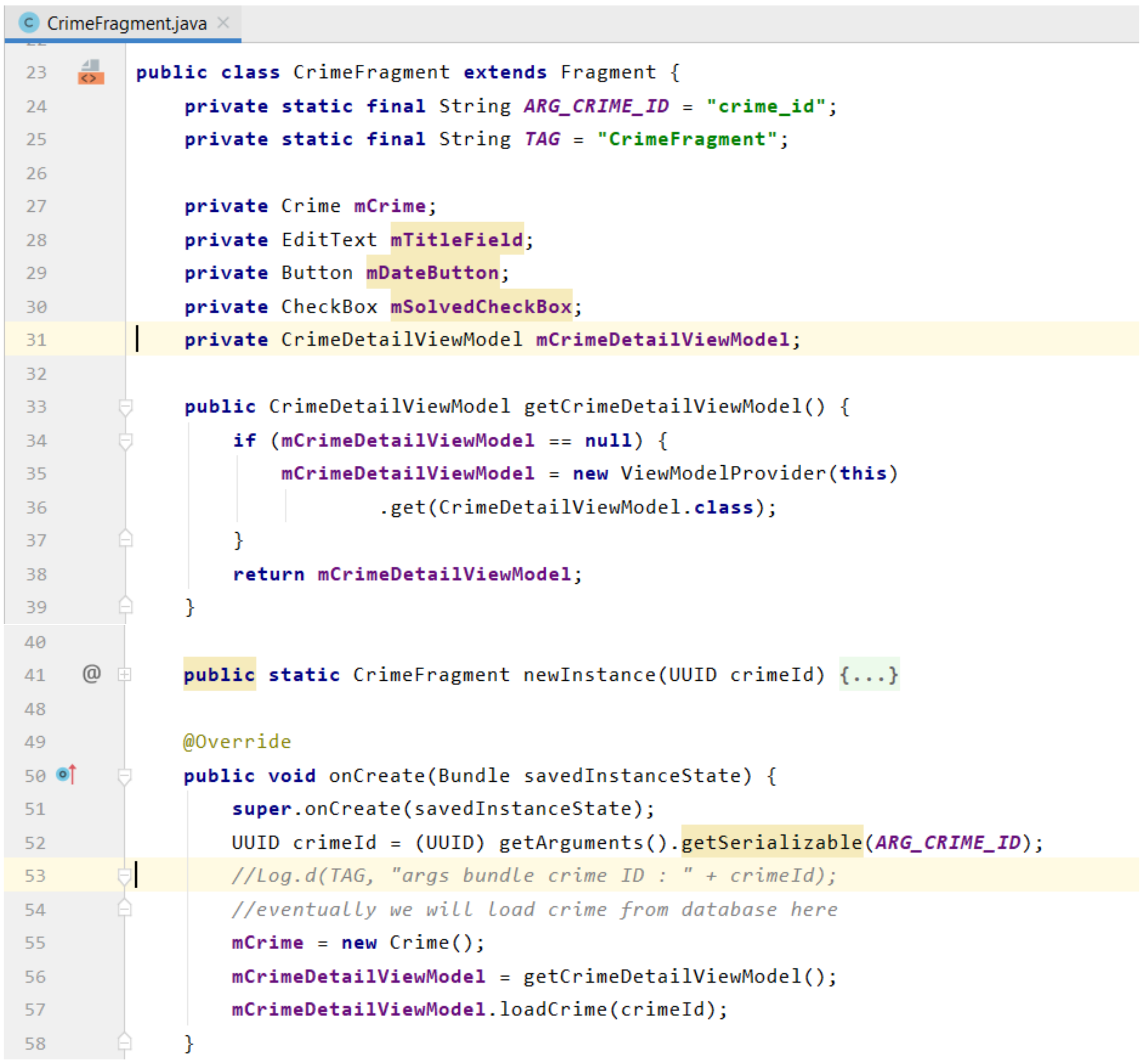


CrimeDetailViewModel.java

```java
package id.ac.astra.polman.nimxxxxx.criminalintent;

import ...

public class CrimeDetailViewModel extends ViewModel {
    private MutableLiveData<UUID> mCrimeIdLiveData;
    private CrimeRepository mCrimeRepository;
    private LiveData<Crime> mCrimeLiveData;

    public CrimeDetailViewModel(){
        mCrimeRepository = CrimeRepository.get();
        mCrimeIdLiveData = new MutableLiveData<UUID>();
        mCrimeLiveData = Transformations.switchMap(mCrimeIdLiveData, crimeId ->
                mCrimeRepository.getCrime(crimeId));
    }

    public void loadCrime(UUID crimeId){
        mCrimeIdLiveData.setValue(crimeId);
    }

    public LiveData<Crime> getCrimeLiveData() {
        return mCrimeLiveData;
    }
}
```

47. The `mCrimeRepository` member variable stores a handle to the `CrimeRepository`. This is not necessary, but later on `CrimeDetailViewModel` will communicate with the repository in more than one place, so the member variable will prove useful at that point.
48. `mCrimeIdLiveData` stores the ID of the crime currently displayed (or about to be displayed) by `CrimeFragment`. When `CrimeDetailViewModel` is first created, the crime ID is not set. Eventually, `CrimeFragment` will call `CrimeDetailViewModel.loadCrime(UUID)` to let the `ViewModel` know which crime it needs to load.
49. Note that you explicitly defined `mCrimeLiveData`'s type as `LiveData<Crime>`. Since `mCrimeLiveData` is publicly exposed, you should ensure it is not exposed as a `MutableLiveData`. In general, `ViewModels` should never expose `MutableLiveData`.

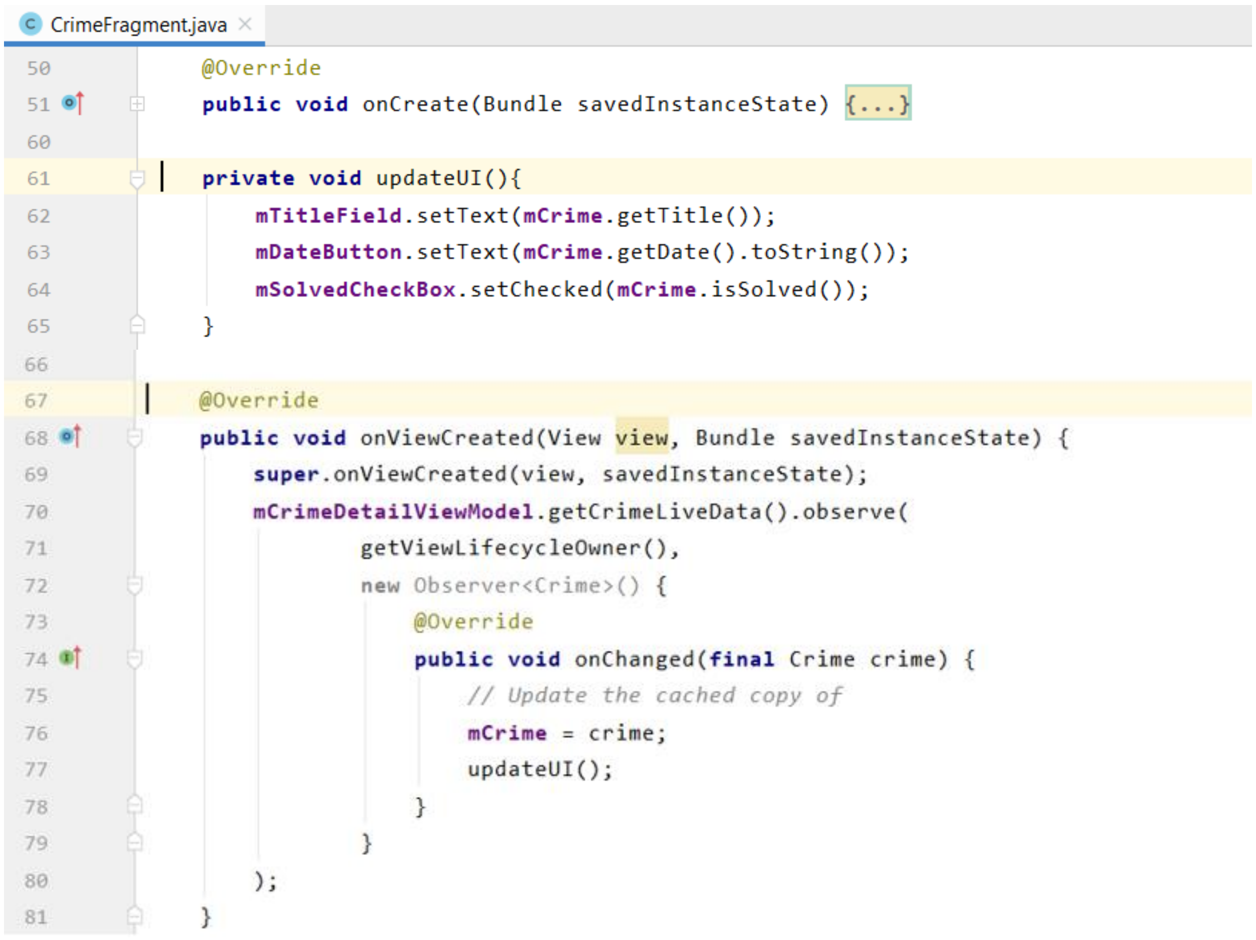
50. It may seem strange to wrap the crime ID in `LiveData`, since it is private to `CrimeDetailViewModel`. What within this `CrimeDetailViewModel`, you may wonder, needs to listen for changes to the private ID value?
51. The answer lies in the live data `Transformation` statement. A `live data transformation` is a way to set up a trigger-response relationship between two `LiveData` objects. A transformation method takes two inputs: a `LiveData` object used as a `trigger` and a `mapping method` that must return a `LiveData` object. The transformation method returns a new `LiveData` object, which we call the `transformation result`, whose value gets updated every time a new value gets set on the trigger `LiveData` instance.
52. The transformation result's value is calculated by executing the mapping method. The `value` property on the `LiveData` returned from the mapping method is used to set the `value` property on the live data transformation result.
53. Using a transformation this way means the `CrimeFragment` only has to observe the exposed `CrimeDetailViewModel.mCrimeLiveData` one time. When the fragment changes the ID it wants to display, the `ViewModel` just publishes the new crime data to the existing live data stream.
54. Open `CrimeFragment.java`. Associate `CrimeFragment` with `CrimeDetailViewModel`. Request that the `ViewModel` load the `Crime` in `onCreate(…)`.



CrimeFragment.java

```java
public class CrimeFragment extends Fragment {
    private static final String ARG_CRIME_ID = "crime_id";
    private static final String TAG = "CrimeFragment";

    private Crime mCrime;
    private EditText mTitleField;
    private Button mDateButton;
    private CheckBox mSolvedCheckBox;
    private CrimeDetailViewModel mCrimeDetailViewModel;

    public CrimeDetailViewModel getCrimeDetailViewModel() {
        if (mCrimeDetailViewModel == null) {
            mCrimeDetailViewModel = new ViewModelProvider(this)
                    .get(CrimeDetailViewModel.class);
        }
        return mCrimeDetailViewModel;
    }

    public static CrimeFragment newInstance(UUID crimeId) {...}

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        UUID crimeId = (UUID) getArguments().getSerializable(ARG_CRIME_ID);
        //Log.d(TAG, "args bundle crime ID : " + crimeId);
        //eventually we will load crime from database here
        mCrime = new Crime();
        mCrimeDetailViewModel = getCrimeDetailViewModel();
        mCrimeDetailViewModel.loadCrime(crimeId);
    }
```

55. Next, observe `CrimeDetailViewModel`'s `crimeLiveData` and update the UI any time new data is published.



CrimeFragment.java

```
    @Override
    public void onCreate(Bundle savedInstanceState) {...}

    private void updateUI(){
        mTitleField.setText(mCrime.getTitle());
        mDateButton.setText(mCrime.getDate().toString());
        mSolvedCheckBox.setChecked(mCrime.isSolved());
    }

    @Override
    public void onViewCreated(View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        mCrimeDetailViewModel.getCrimeLiveData().observe(
                getViewLifecycleOwner(),
                new Observer<Crime>() {
                    @Override
                    public void onChanged(final Crime crime) {
                        // Update the cached copy of
                        mCrime = crime;
                        updateUI();
                    }
                }
        );
    }
```

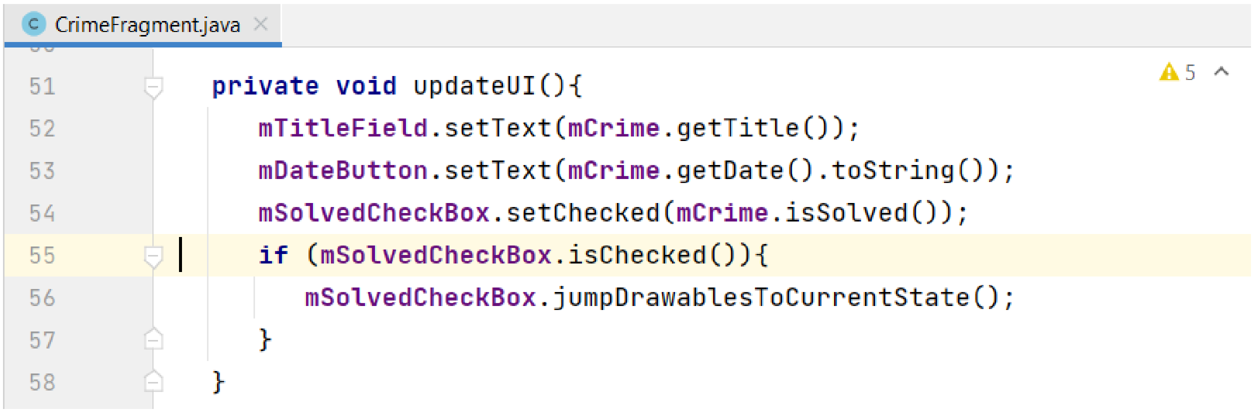
56. (Be sure to import `androidx.lifecycle.Observer`.)
57. You may have noticed that `CrimeFragment` has its own `Crime` state stored in its `mCrime` field member. The values in this `mCrime` field member represent the edits the user is currently making. The crime in `CrimeDetailViewModel.crimeLiveData` represents the data as it is currently saved in the database. `CrimeFragment` "publishes" the user's edits when the fragment moves to the stopped state by writing the updated data to the database.

58. **Run your app**. Press a crime in the list. If all goes as planned, you should see the crime detail screen appear, populated with data from the crime you pressed in the list

59. When `CrimeFragment` displays, you may see the checkbox animating to the checked state if you are viewing a crime marked as solved. This is expected, since the checkbox state gets set as the result of an asynchronous operation. The database query for the crime kicks off when the user first launches `CrimeFragment`. When the database query

completes, the fragment's `mCrimeDetailViewModel.mCrimeLiveData` observer is notified and in turn updates the data displayed in the widgets.

60. Clean this up by skipping over the animation when you programmatically set the checkbox checked state. You do this by calling `View.jumpDrawablesToCurrentState()`. Note that if the lag as the crime detail screen loads is unacceptable based on your app's requirements, you could pre-load the crime data into memory ahead of time (as the app launches, for example) and stash it in a shared place. For CriminalIntent, the lag is very slight, so the simple solution of skipping the animation is enough.



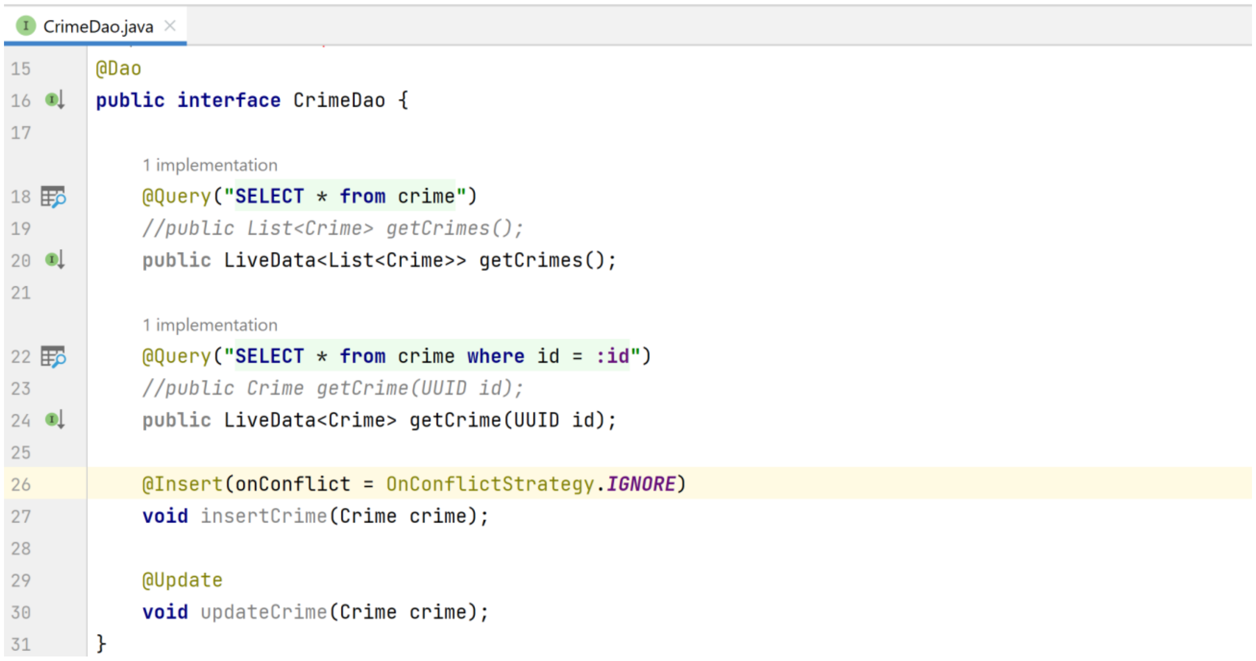
```java
private void updateUI(){
    mTitleField.setText(mCrime.getTitle());
    mDateButton.setText(mCrime.getDate().toString());
    mSolvedCheckBox.setChecked(mCrime.isSolved());
    if (mSolvedCheckBox.isChecked()){
        mSolvedCheckBox.jumpDrawablesToCurrentState();
    }
}
```

61. **Run your app again**. Press a solved crime in the list. Notice the checkbox no longer animates as the screen spins up. If you press the checkbox, the animation does happen, as desired.

62. Now, edit the crime's title. Press the Back button to go back to the crime list screen. Sadly, the changes you made were not saved. Luckily, this is easy to fix.

Updating the Database

63. The database serves as the single source of truth for crime data. In CriminalIntent, when the user leaves the detail screen, any edits they made should be saved to the database. (Other apps might have other requirements, like having a "save" button or saving updates as the user types.)
64. First, add a method to your `CrimeDao` to update an existing crime.



CrimeDao.java

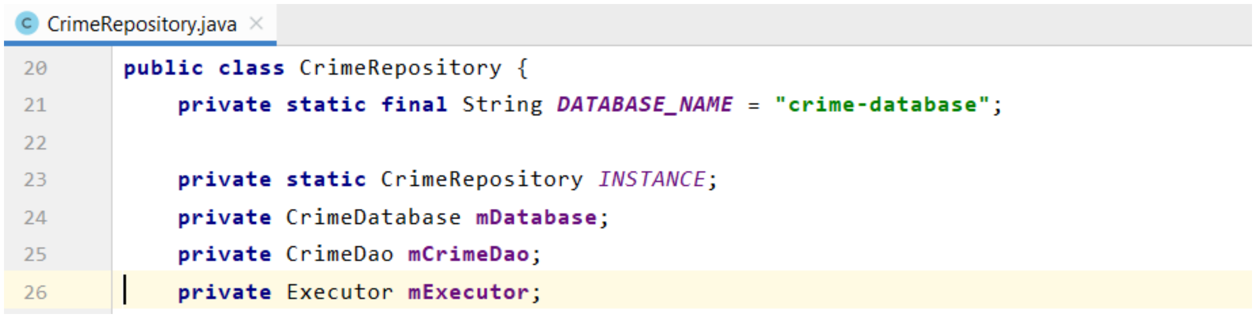
```
@Dao
public interface CrimeDao {

    1 implementation
    @Query("SELECT * from crime")
    //public List<Crime> getCrimes();
    public LiveData<List<Crime>> getCrimes();

    1 implementation
    @Query("SELECT * from crime where id = :id")
    //public Crime getCrime(UUID id);
    public LiveData<Crime> getCrime(UUID id);

    @Insert(onConflict = OnConflictStrategy.IGNORE)
    void insertCrime(Crime crime);

    @Update
    void updateCrime(Crime crime);
}
```

65. The annotations for these methods do not require any parameters. Room can generate the appropriate SQL command for these operations.
66. The `updateCrime()` method uses the `@Update` annotation. This method takes in a crime object, uses the ID stored in that crime to find the associated row, and then updates the data in that row based on the new data in the crime object.
67. The insert() method uses the `@Insert` annotation. The parameter is the crime you want to add to the database table.
68. Next, update the repository to call through to the new update DAO methods. Recall that Room automatically executes the database queries for `CrimeDao.getCrimes()` and `CrimeDao.getCrime(UUID)` on a background thread because those DAO methods return `LiveData`. In those cases, `LiveData` handles ferrying the data over to your main thread so you can update your UI.
69. You cannot, however, rely on Room to automatically run your insert and update database interactions on a background thread for you. Instead, you must execute those DAO calls on a background thread explicitly. A common way to do this is to use an `executor`.

Using an executor

70. An `Executor` is an object that references a thread. An executor instance has a method called `execute` that accepts a block of code to run. The code you provide in the block will run on whatever thread the executor points to.
71. You are going to create an executor that uses a new thread (which will always be a background thread). Any code in the block will run on that thread, so you can perform your database operations there safely.
72. You cannot implement an executor in the `CrimeDao` directly, because Room generates the method implementations for you based on the interface you define. Instead,

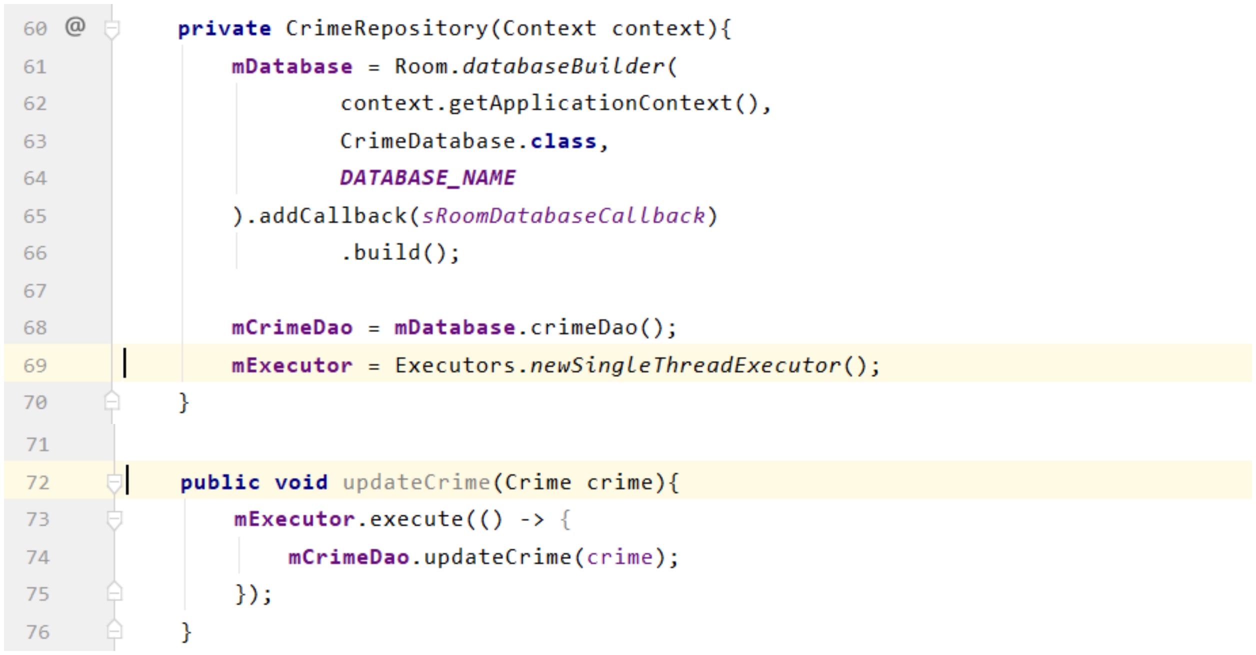
implement the executor in `CrimeRepository`. Add a property to the executor to hold a reference, then execute your insert and update methods using the executor.



CrimeRepository.java

```
public class CrimeRepository {
    private static final String DATABASE_NAME = "crime-database";

    private static CrimeRepository INSTANCE;
    private CrimeDatabase mDatabase;
    private CrimeDao mCrimeDao;
    private Executor mExecutor;
```

…



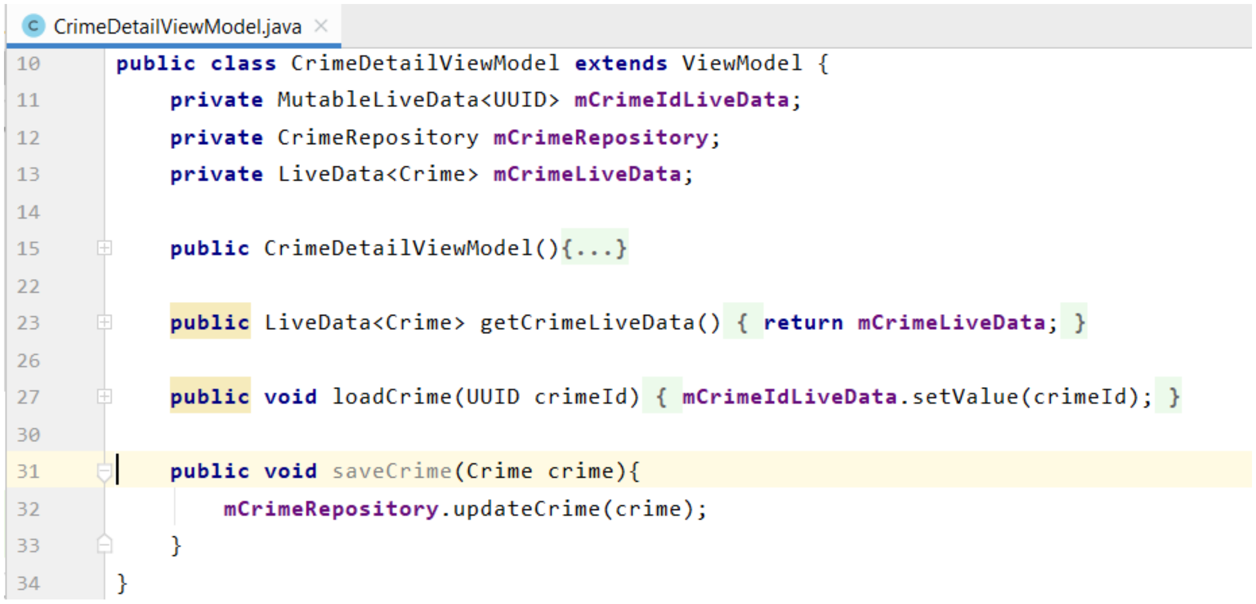
```
    private CrimeRepository(Context context){
        mDatabase = Room.databaseBuilder(
                context.getApplicationContext(),
                CrimeDatabase.class,
                DATABASE_NAME
        ).addCallback(sRoomDatabaseCallback)
                .build();

        mCrimeDao = mDatabase.crimeDao();
        mExecutor = Executors.newSingleThreadExecutor();
    }

    public void updateCrime(Crime crime){
        mExecutor.execute(() -> {
            mCrimeDao.updateCrime(crime);
        });
    }
```

73. The `newSingleThreadExecutor()` method returns an executor instance that points to a new thread. Any work you execute with the executor will therefore happen off the main thread.
74. `updateCrime()` wrap their calls to the DAO inside the `execute {}` block. This pushes these operations off the main thread so you do not block your UI.

Tying database writes to the fragment lifecycle

75. Last but not least, update your app to write the values the user enters in the crime detail screen to the database when the user navigates away from the screen.

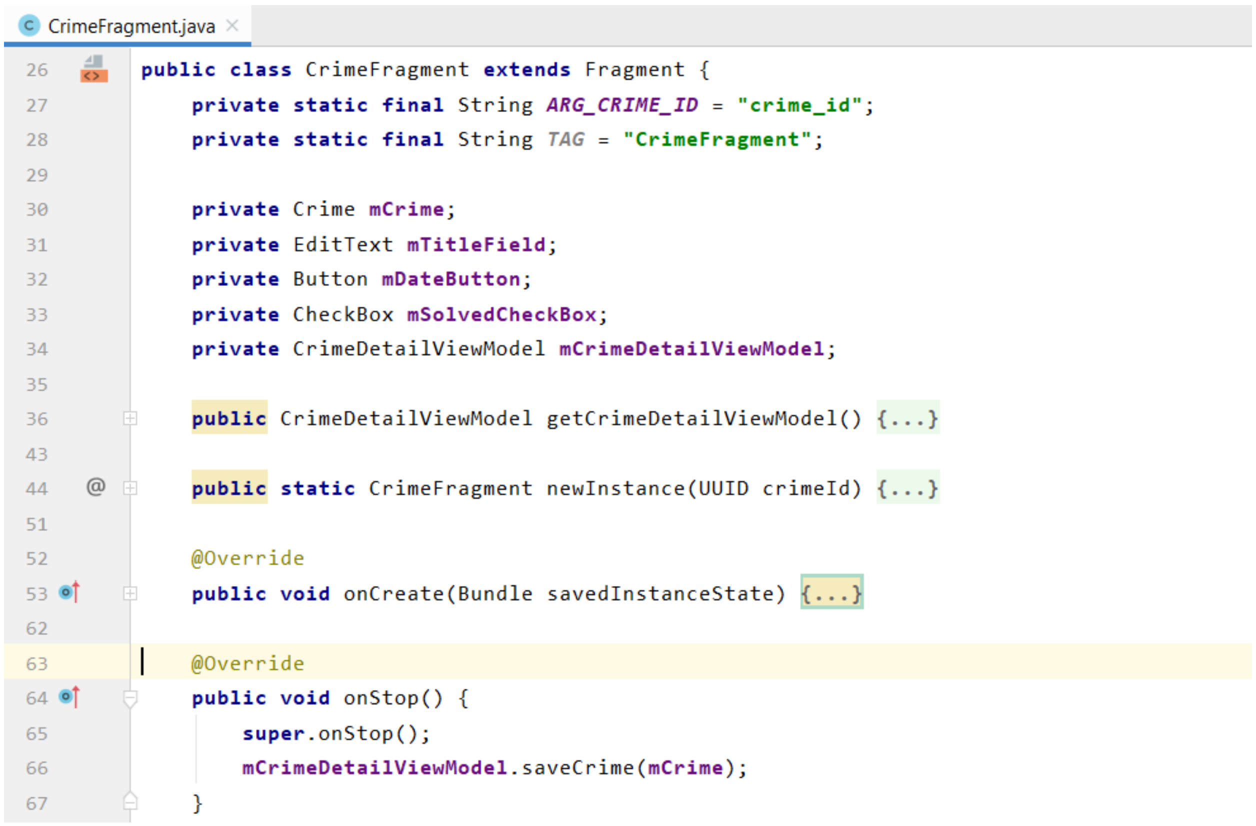
76. Open `CrimeDetailViewModel.java` and add a method to save a crime object to the database.



CrimeDetailViewModel.java

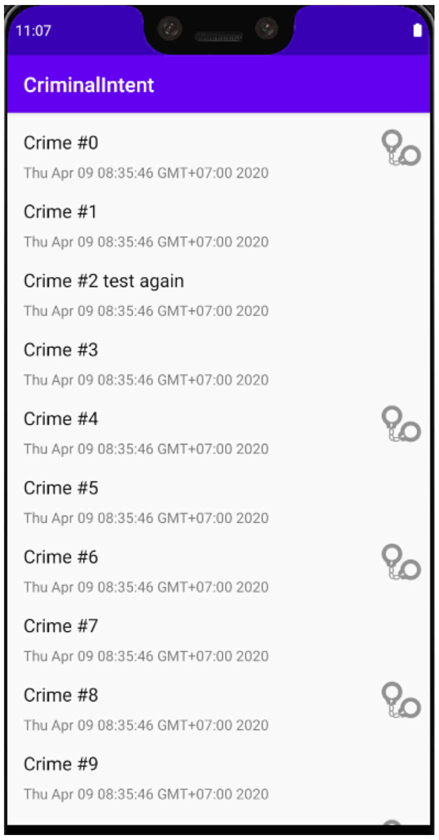
```java
public class CrimeDetailViewModel extends ViewModel {
    private MutableLiveData<UUID> mCrimeIdLiveData;
    private CrimeRepository mCrimeRepository;
    private LiveData<Crime> mCrimeLiveData;

    public CrimeDetailViewModel(){...}

    public LiveData<Crime> getCrimeLiveData() { return mCrimeLiveData; }

    public void loadCrime(UUID crimeId) { mCrimeIdLiveData.setValue(crimeId); }

    public void saveCrime(Crime crime){
        mCrimeRepository.updateCrime(crime);
    }
}
```

77. `saveCrime(Crime)` accepts a `Crime` and writes it to the database. Since `CrimeRepository` handles running the update request on a background thread, the database integration here is very simple.
78. Now, update `CrimeFragment` to save the user’s edited crime data to the database.


CrimeFragment.java

```
public class CrimeFragment extends Fragment {
    private static final String ARG_CRIME_ID = "crime_id";
    private static final String TAG = "CrimeFragment";

    private Crime mCrime;
    private EditText mTitleField;
    private Button mDateButton;
    private CheckBox mSolvedCheckBox;
    private CrimeDetailViewModel mCrimeDetailViewModel;

    public CrimeDetailViewModel getCrimeDetailViewModel() {...}

    public static CrimeFragment newInstance(UUID crimeId) {...}

    @Override
    public void onCreate(Bundle savedInstanceState) {...}

    @Override
    public void onStop() {
        super.onStop();
        mCrimeDetailViewModel.saveCrime(mCrime);
    }
```

79. `Fragment.onStop()` is called any time your fragment moves to the stopped state (that is, any time the fragment moves entirely out of view). This means the data will get saved when the user finishes the detail screen (such as by pressing the Back button). The data will also be saved when the user switches tasks (such as by pressing the Home button or using the overview screen). Thus, saving in `onStop()` meets the requirement of saving the data whenever the user leaves the detail screen and also ensures that, if the process is killed due to memory pressure, the edited data is not lost.
80. Run CriminalIntent and select a crime from the list. Change data in that crime. Press the Back button and pat yourself on the back for your most recent success: The changes you made on the detail screen are now reflected in the list screen. In the next practice, you will hook up the date button on the detail screen to allow the user to select the date when the crime occurred.


11:07

CriminalIntent

Crime #0
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #1
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #2 test again
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #3
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #4
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #5
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #6
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #7
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #8
Thu Apr 09 08:35:46 GMT+07:00 2020

Crime #9
Thu Apr 09 08:35:46 GMT+07:00 2020

###

Notes:

1. Create folder PRG6_M5_P1_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.