

Level/Subject : **D3/Programming 6**  
Topic : SQLite Database in Android apps  
Week : 4th  
Activity : Using Database and Room library in Android apps using Android Studio  
Alocated time : 240 mins labs  
Deliverables : Project folder  
Due date : end of week

---

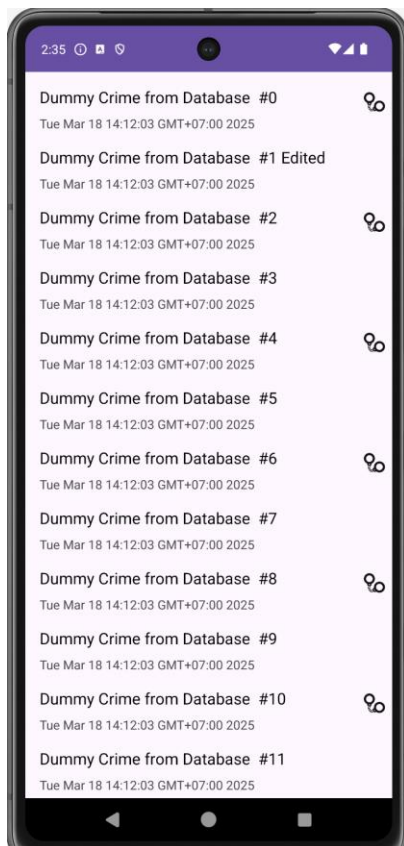
### Competency :

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

### Example Practice Task:

Create Android apps with database using Android Studio.

1. Almost every application needs a place to save data for the long term. Here you will implement a database for CriminalIntent and seed it with dummy data. Then you will update the app to pull crime data from the database and display it in the crime list.

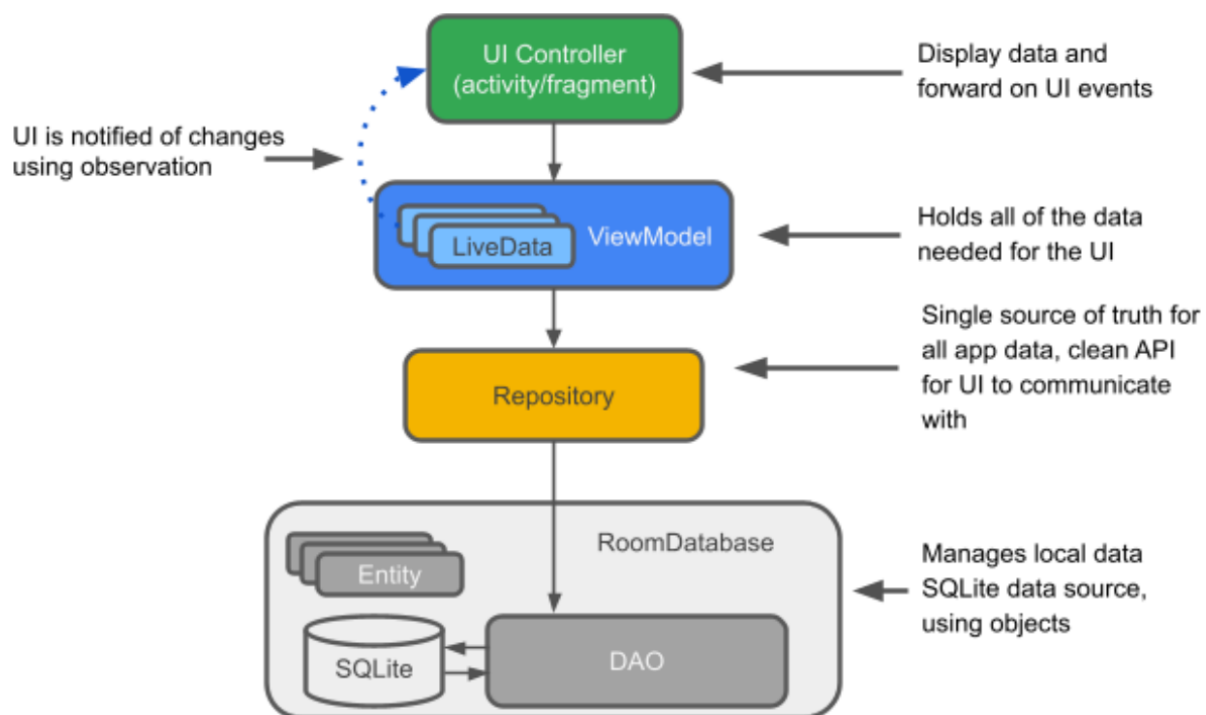


2. you learned how to persist transient UI state data across rotation and process death using `ViewModel` and saved instance state. These two approaches are great for small amounts of data tied to the UI. However, these approaches should not be used for storing non-UI data or data that is not tied to an activity or fragment instance and needs to persist regardless of UI state.
3. Instead, store your non-UI app data either locally (on the filesystem or in a database, as you will do for `CriminalIntent`) or on a web server.

## Room Architecture Component Library

4. You can learn about Room Architecture here:

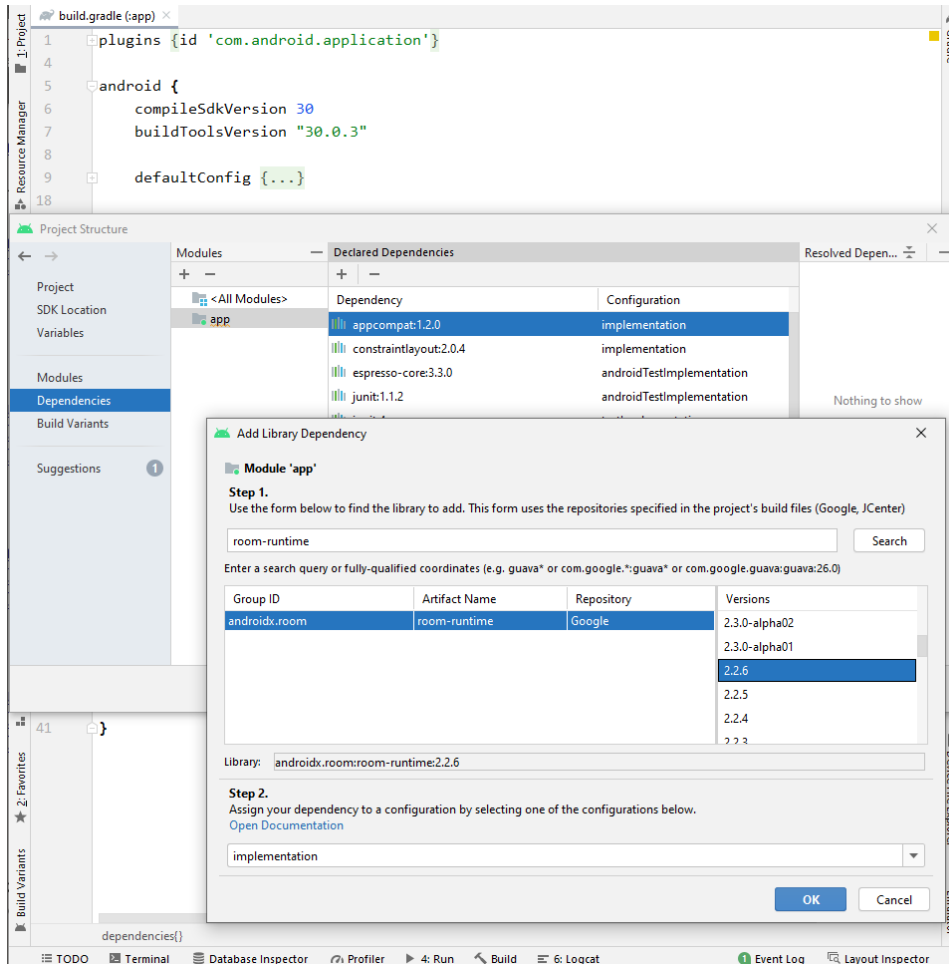
<https://developer.android.com/codelabs/android-room-with-a-view#0>

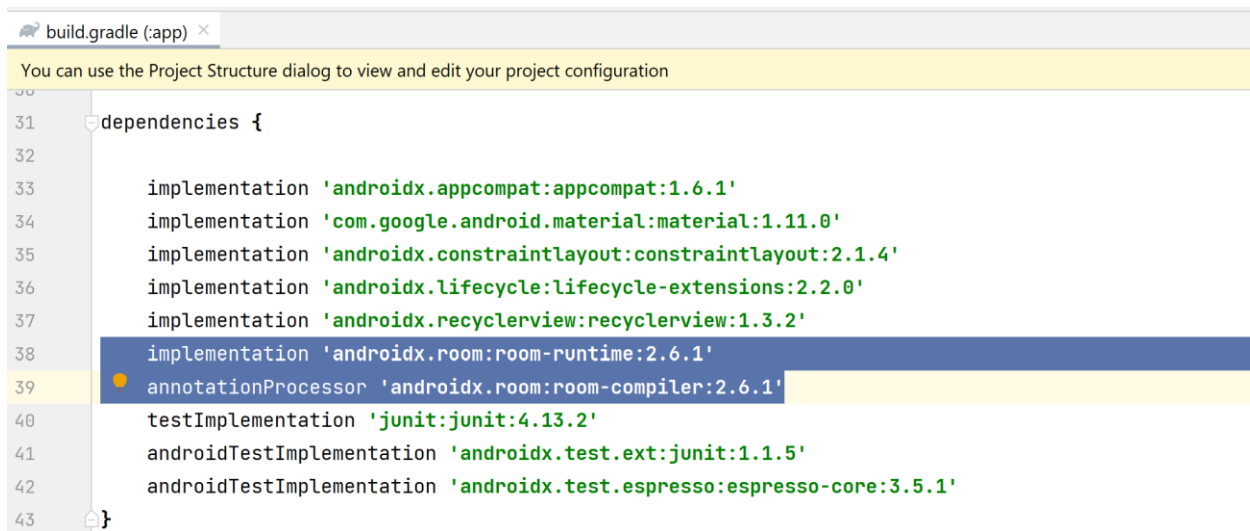


5. Room is a Jetpack architecture component library that simplifies database setup and access. It allows you to define your database structure and queries using annotated Java classes.
6. Room is composed of an API, annotations, and a compiler. The API contains classes you extend to define your database and build an instance of it. You use the annotations to indicate things like which classes need to be stored in the database, which class represents your database, and which class specifies the accessor functions to your

database tables. The compiler processes the annotated classes and generates the implementation of your database.

7. To use Room, you first need to add the dependencies it requires. Add the `room-runtime` and `room-compiler` dependencies to your `app/build.gradle` file.





```
build.gradle (.app) ×
You can use the Project Structure dialog to view and edit your project configuration

31 dependencies {
32
33     implementation 'androidx.appcompat:appcompat:1.6.1'
34     implementation 'com.google.android.material:material:1.11.0'
35     implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
36     implementation 'androidx.lifecycle:lifecycle-extensions:2.2.0'
37     implementation 'androidx.recyclerview:recyclerview:1.3.2'
38     implementation 'androidx.room:room-runtime:2.6.1'
39     annotationProcessor 'androidx.room:room-compiler:2.6.1'
40     testImplementation 'junit:junit:4.13.2'
41     androidTestImplementation 'androidx.test.ext:junit:1.1.5'
42     androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
43 }
```

8. Do not forget to sync your Gradle files. With your dependencies in place, you can move on to preparing your model layer for storage in the database.
9. Before you create a database, you have to decide what will be in that database. CriminalIntent stores a single list of crimes, so you will define one table named **crimes**

## Creating a Database

There are three steps to creating a database with Room:

- annotating your model class to make it a database entity
- creating the class that will represent the database itself
- creating a type converter so that your database can handle your model data

Room makes each of these steps straightforward

## Defining entities

10. Room structures the database tables for your application based on the entities you define. Entities are model classes you create, annotated with the `@Entity` annotation. Room will create a database table for any class with that annotation.
11. Since you want to store crime objects in your database, update `Crime` to be a Room entity. Open `Crime.java` and add the annotations:



```

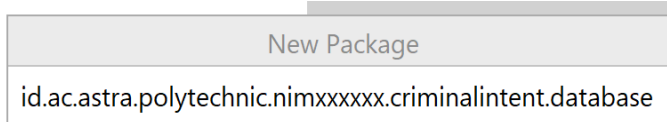
10  @Entity
11  public class Crime {
12
13      @PrimaryKey
14      @NonNull
15      private UUID id;
16      private String mTitle;
17      private Date mDate;
18      private boolean mSolved;
19
20      public Crime(){
21          id = UUID.randomUUID();
22          mTitle = "";
23          mDate = new Date();
24          mSolved = false;
25      }

```

12. The first annotation, `@Entity`, is applied at the class level. This entity annotation indicates that the class defines the structure of a table, or set of tables, in the database. In this case, each row in the table will represent an individual `Crime`. Each field member defined on the class will be a column in the table, with the name of the field member as the name of the column. The table that stores your crimes will have four columns: `id`, `mTitle`, `mDate`, and `mSolved`.
13. The other annotation you added is `@PrimaryKey`, which you added to the `id` field member. This annotation specifies which column in your database is the primary key. The primary key in a database is a column that holds data that is unique for each entry, or row, so that it can be used to look up individual entries. The `id` field member is unique for every `Crime`, so by adding `@PrimaryKey` to this field member you will be able to query a single crime from the database using its `id`.
14. `@NonNull` Denotes that a parameter, field, or method return value can never be null
15. Now that your `Crime` class is annotated, you can move on to creating your database class.

## Creating a database class

16. Entity classes define the structure of database tables. A single entity class could be used across multiple databases, should your app have more than one database. That case is not common, but it is possible. For this reason, an entity class is not used by Room to create a table unless you explicitly associate it with a database, which you will do shortly.
17. First, create a new package called `database` for your database-specific code. In the project tool window, right-click the `id.ac.astra.polytechnic.nimxxxxx.criminalintent` folder and choose `New → Package`. Name your new package `database`.



18. Now, create a new class called `CrimeDatabase` in the `database` package and define the class as shown below.

```
CrimeDatabase.java x
8  @Database(entities = {Crime.class}, version = 1)
9  public abstract class CrimeDatabase extends RoomDatabase {
10
11 }
```

19. The `@Database` annotation tells Room that this class represents a database in your app. The annotation itself requires two parameters. The first parameter is a list of entity classes, which tells Room which entity classes to use when creating and managing tables for this database. In this case, you only pass the `Crime` class, since it is the only entity in the app.
20. The second parameter is the version of the database. When you first create a database, the version should be 1. As you develop your app in the future, you may add new entities and new properties to existing entities. When this happens, you will need to modify your entities list and increment your database version to tell Room something has changed.
21. The database class itself is empty at this point. `CrimeDatabase` extends from `RoomDatabase` and is marked as abstract, so you cannot make an instance of it directly. You will learn how to use Room to get a database instance you can use later.

## Creating a type converter

22. Room uses SQLite under the hood. SQLite is an open source relational database, like MySQL or PostgreSQL. (SQL, short for Structured Query Language, is a standard language used for interacting with databases. People pronounce “SQL” as either “sequel” or as an initialism, “S-Q-L.”) Unlike other databases, SQLite stores its data in simple files you can read and write using the SQLite library. Android includes this SQLite library in its standard library, along with some additional helper classes.
23. Room makes using SQLite even easier and cleaner, serving as an object-relational mapping (or ORM) layer between your Java objects and database implementation. For the most part, you do not need to know or care about SQLite when using Room, but if you want to learn more you can visit [www.sqlite.org](http://www.sqlite.org), which has complete SQLite documentation.
24. Room is able to store primitive types with ease in the underlying SQLite database tables, but other types will cause issues. Your `Crime` class relies on the `Date` and `UUID` objects, which Room does not know how to store by default. You need to give the

database a hand so it knows how to store these types and how to pull them out of the database table correctly.

25. To tell Room how to convert your data types, you specify a `type converter`. A type converter tells Room how to convert a specific type to the format it needs to store in the database. You will need two functions, which you will annotate with `@TypeConverter`, for each type: One tells Room how to convert the type to store it in the database, and the other tells Room how to convert from the database representation back to the original type.
26. Create a class called `CrimeTypeConverters` in the `database` package and add two functions each for the `Date` and `UUID` types.

```
CrimeTypeConverters.java x
8  public class CrimeTypeConverters {
9      @TypeConverter
10     public Long fromDate(Date date){
11         if (date == null){
12             return null;
13         } else {
14             return date.getTime();
15         }
16     }
17
18     @TypeConverter
19     public Date toDate(Long millisSinceEpoch){
20         if (millisSinceEpoch == null){
21             return new Date();
22         } else {
23             return new Date(millisSinceEpoch);
24         }
25     }
26
27     @TypeConverter
28     public UUID toUUID(String uuid){
29         if (uuid == null){
30             return UUID.randomUUID();
31         } else {
32             return UUID.fromString(uuid);
33         }
34     }
35
36     @TypeConverter
37     public String fromUUID(UUID uuid){
38         if (uuid == null){
39             return null;
40         } else {
41             return uuid.toString();
42         }
43     }
44 }
```

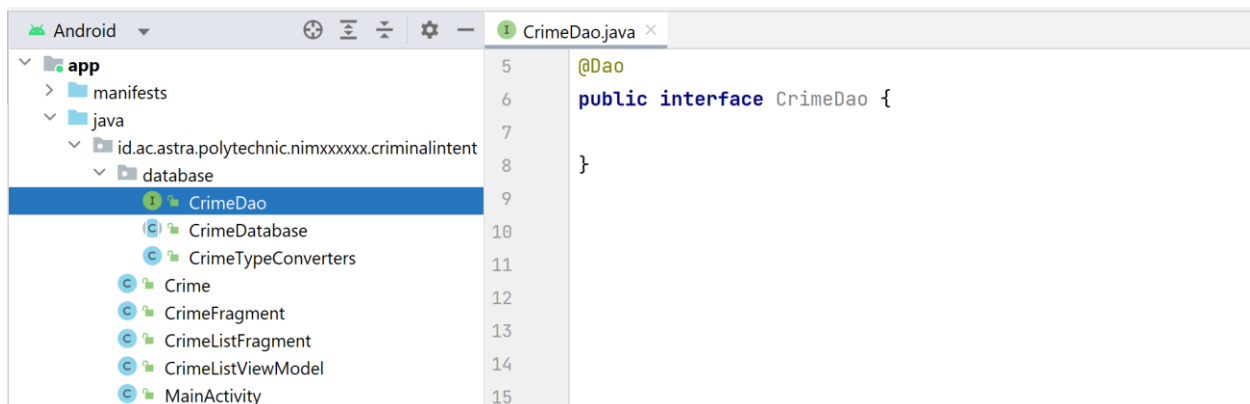
27. The first two functions handle the `Date` object, and the second two handle the UUIDs. Make sure you import the `java.util.Date` version of the `Date` class.
28. Declaring the converter functions does not enable your database to use them. You must explicitly add the converters to your database class.

```
CrimeDatabase.java x
9  @Database(entities = {Crime.class}, version = 1)
10 @TypeConverters(CrimeTypeConverters.class)
11 public abstract class CrimeDatabase extends RoomDatabase {
12
13 }
```

29. By adding the `@TypeConverters` annotation and passing in your `CrimeTypeConverters` class, you tell your database to use the functions in that class when converting your types.
30. With that, your database and table definitions are complete.

## Defining a Data Access Object

31. A database table does not do much good if you cannot edit or access its contents. The first step to interacting with your database tables is to create a data access object, or DAO. A DAO is an interface that contains functions for each database operation you want to perform. `CriminalIntent`'s DAO needs two query functions: one to return a list of all crimes in the database and another to return a single crime matching a given UUID.
32. Add a file named `CrimeDao.java` to the database package. In it, define an empty interface named `CrimeDao` annotated with Room's `@Dao` annotation.



The screenshot shows an IDE with the project structure on the left and the `CrimeDao.java` file open on the right. The project structure shows a package `id.ac.astra.polytechnic.nimxxxxx.criminalintent` with a sub-package `database`. The `database` package contains several files: `CrimeDao` (selected), `CrimeDatabase`, `CrimeTypeConverters`, `Crime`, `CrimeFragment`, `CrimeListFragment`, `CrimeListViewModel`, and `MainActivity`. The `CrimeDao.java` file is empty except for the `@Dao` annotation and the `public interface CrimeDao {` declaration.

```
CrimeDao.java x
5  @Dao
6  public interface CrimeDao {
7
8  }
9
10
11
12
13
14
15
```

33. The `@Dao` annotation lets Room know that `CrimeDao` is one of your data access objects. When you hook `CrimeDao` up to your database class, Room will generate implementations of the functions you add to this interface.



34. Speaking of adding functions, now is the time. Add two query functions to `CrimeDao`.

```
CrimeDao.java x
11 @Dao
12 public interface CrimeDao {
13
14     @Query("SELECT * from crime")
15     public List<Crime> getCrimes();
16
17     @Query("SELECT * from crime where id = :id")
18     public Crime getCrime(UUID id);
19
20 }
```

35. The `@Query` annotation indicates that `getCrimes()` and `getCrime(UUID)` are meant to pull information out of the database, rather than inserting, updating, or deleting items from the database. The return type of each query function in the DAO interface reflects the type of result the query will return.
36. The `@Query` annotation expects a string containing a SQL command as input. In most cases you only need to know minimal SQL to use Room, but if you are interested in learning more check out the SQL Syntax section at [www.sqlite.org](http://www.sqlite.org).
37. `SELECT * FROM crime` asks Room to pull all columns for all rows in the `crime` database table. `SELECT * FROM crime WHERE id=:id` asks Room to pull all columns from only the row whose `id` matches the ID value provided.
38. With that, the `CrimeDao` is complete, at least for now.
39. Next, you need to register your DAO class with your database class. Since the `CrimeDao` is an interface, Room will handle generating the concrete version of the class for you. But for that to work, you need to tell your database class to make an abstract method to expose the DAO.

```
CrimeDatabase.java x
9 @Database(entities = {Crime.class}, version = 1)
10 @TypeConverters(CrimeTypeConverters.class)
11 abstract public class CrimeDatabase extends RoomDatabase {
12
13     public abstract CrimeDao crimeDao();
14 }
```

40. Now, when the database is created, Room will generate a concrete implementation of the DAO that you can access. Once you have a reference to the DAO, you can call any of the functions defined on it to interact with your database.

## Accessing the Database Using the Repository Pattern

41. To access your database, you will use the `repository` pattern recommended by Google in its Guide to App Architecture ([developer.android.com/jetpack/docs/guide](https://developer.android.com/jetpack/docs/guide)).
42. A `repository` class encapsulates the logic for accessing data from a single source or a set of sources. It determines how to fetch and store a particular set of data, whether locally in a database or from a remote server. Your UI code will request all the data from the repository, because the UI does not care how the data is actually stored or fetched. Those are implementation details of the repository itself.
43. Since `CriminalIntent` is a simpler app, the repository will only handle fetching data from the database.
44. Create a class called `CrimeRepository` in the `id.ac.astra.polytechnic.nimxxxxx.criminalintent` package and define a companion object/[singleton](#) in the class.

```
CrimeRepository.java
5 public class CrimeRepository {
6     private static CrimeRepository INSTANCE;
7
8     private CrimeRepository(Context context){
9
10    }
11
12    public static void initialize(Context context){
13        if (INSTANCE == null){
14            INSTANCE = new CrimeRepository(context);
15        }
16    }
17
18    public static CrimeRepository get(){
19        return INSTANCE;
20    }
21 }
```

45. `CrimeRepository` is a singleton. This means there will only ever be one instance of it in your app process.
46. A singleton exists as long as the application stays in memory, so storing any properties on the singleton will keep them available throughout any lifecycle changes in your activities and fragments. Be careful with singleton classes, as they are destroyed when Android removes your application from memory. The `CrimeRepository` singleton is not a solution for long-term storage of data. Instead, it gives the app an owner for the crime data and provides a way to easily pass that data between controller classes.
47. To make `CrimeRepository` a singleton, you add two functions to its companion object. One initializes a new instance of the repository, and the other accesses the repository. You also mark the constructor as private to ensure no components can go rogue and create their own instance.

48. To do work as soon as your application is ready, you can create an `Application` subclass. This allows you to access lifecycle information about the application itself.

49. Create a class called `CriminalIntentApplication` that extends `Application` and override `Application.onCreate()` to set up the repository initialization.

```
CriminalIntentApplication.java x
5  public class CriminalIntentApplication extends Application {
6
7      @Override
8      public void onCreate() {
9          super.onCreate();
10         CrimeRepository.initialize(this);
11     }
12 }
```

50. Similar to `Activity.onCreate(...)`, `Application.onCreate()` is called by the system when your application is first loaded in to memory. That makes it a good place to do any kind of one-time initialization operations.

51. The application instance does not get constantly destroyed and re-created, like your activity or fragment classes. It is created when the app launches and destroyed when your app process is destroyed. The only lifecycle function you will override in `CriminalIntent` is `onCreate()`.

52. In a moment, you are going to pass the application instance to your repository as a `Context` object. This object is valid as long as your application process is in memory, so it is safe to hold a reference to it in the repository class.

53. But in order for your application class to be used by the system, you need to register it in your manifest. Open `AndroidManifest.xml` and specify the `android:name` property to set up your application.

```
AndroidManifest.xml x
1  <?xml version="1.0" encoding="utf-8"?>
2  <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:tools="http://schemas.android.com/tools">
4
5      <application
6          android:name=".CriminalIntentApplication"
7          android:allowBackup="true"
8          android:dataExtractionRules="@xml/data_extraction_rules"
9          android:fullBackupContent="@xml/backup_rules"
10         android:icon="@mipmap/ic_launcher"
```

54. With the application class registered in the manifest, the OS will create an instance of `CriminalIntentApplication` when launching your app. The OS will then call `onCreate()` on the `CriminalIntentApplication` instance. Your

CrimeRepository will be initialized, and you can access it from your other components.

55. Next, add two member variables on your CrimeRepository to store references to your database and DAO objects.

```
CrimeRepository.java x
11
12 public class CrimeRepository {
13     private static final String DATABASE_NAME = "crime-database";
14
15     private static CrimeRepository INSTANCE;
16     private CrimeDatabase mDatabase;
17     private CrimeDao mCrimeDao;
18
19     @ private CrimeRepository(Context context){
20         mDatabase = Room.databaseBuilder(
21             context.getApplicationContext(),
22             CrimeDatabase.class,
23             DATABASE_NAME
24         ).build();
25
26         mCrimeDao = mDatabase.crimeDao();
27     }
28
29     public static void initialize(Context context){
30         if (INSTANCE == null){
31             INSTANCE = new CrimeRepository(context);
32         }
33     }
34
35     @ public static CrimeRepository get(){
36         return INSTANCE;
37     }
38 }
```

56. `Room.databaseBuilder()` creates a concrete implementation of your abstract `CrimeDatabase` using three parameters. It first needs a `Context` object, since the database is accessing the filesystem. You pass in the application context because, as discussed above, the singleton will most likely live longer than any of your activity classes.
57. The second parameter is the database class that you want Room to create. The third is the name of the database file you want Room to create for you. You are using a private string constant defined in the same file, since no other components need to access it.

58. Next, fill out your `CrimeRepository` so your other components can perform any operations they need to on your database. Add a function to your repository for each function in your DAO.

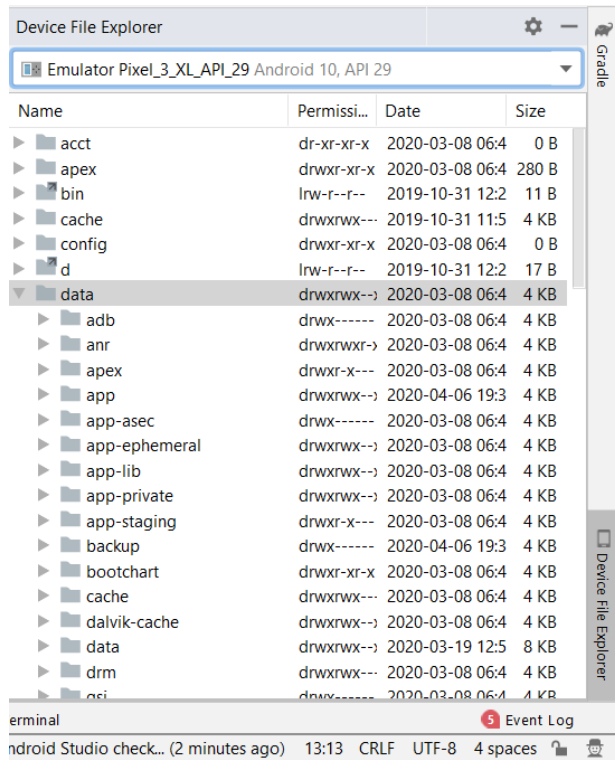
```
CrimeRepository.java x
12 public class CrimeRepository {
13     private static final String DATABASE_NAME = "crime-database";
14
15     private static CrimeRepository INSTANCE;
16     private CrimeDatabase mDatabase;
17     private CrimeDao mCrimeDao;
18
19     @ private CrimeRepository(Context context){...}
28
29     public static void initialize(Context context){...}
34
35     @ public static CrimeRepository get() { return INSTANCE; }
38
39     public List<Crime> getCrimes(){
40         return mCrimeDao.getCrimes();
41     }
42
43     public Crime getCrime(UUID id){
44         return mCrimeDao.getCrime(id);
45     }
46 }
```

59. Since Room provides the query implementations in the DAO, you call through to those implementations from your repository. This helps keep your repository code short and easy to understand.
60. This may seem like a lot of work for little gain, since the repository is just calling through to functions on your `CrimeDao`. But fear not; you will be adding functionality soon to encapsulate additional work the repository needs to handle.

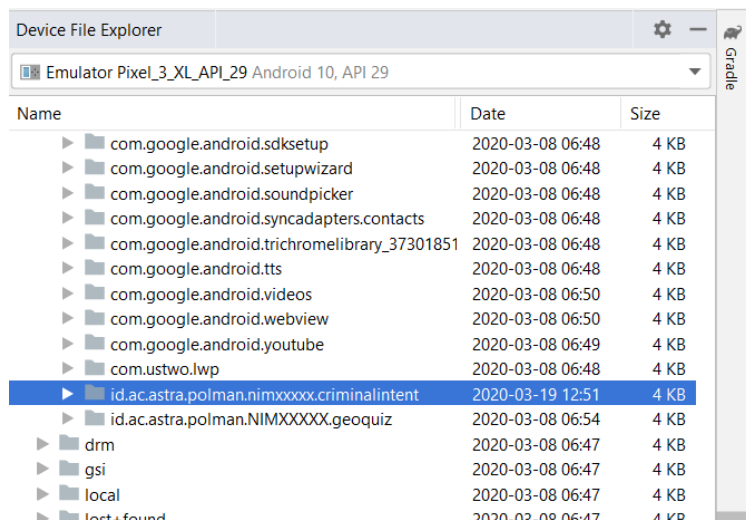
## Data

61. Each application on an Android device has a directory in the device's sandbox. Keeping files in the sandbox protects them from being accessed by other applications or even the prying eyes of users (unless the device has been rooted, in which case the user can get to whatever they like).
62. An application's sandbox directory is a child of the device's `data/data` directory named after the application package. For `CriminalIntent`, the full path to the sandbox

directory is data/data/  
id.ac.astra.polytechnic.nimxxxxx.criminalintent.



63. To find the folder for CriminalIntent, open the data/data/ folder and find the subfolder with the name matching the package ID of the project.



com.geogebra.mr.geometry	2020-03-08 06:48	4 KB
com.ustwo.lwp	2020-03-08 06:48	4 KB
id.ac.astra.polman.nimxxxxx.criminalintent	2020-03-19 12:51	4 KB
cache	2020-03-19 12:51	4 KB
code_cache	2020-03-19 12:51	4 KB
databases	2020-04-06 19:46	4 KB
crime-database	2020-04-06 19:46	32 KB
crime-database-shm	2020-04-06 19:46	32 KB
crime-database-wal	2020-04-06 19:46	406.4 KB
id.ac.astra.polman.NIMXXXXX.geoquiz	2020-03-08 06:54	4 KB
drm	2020-03-08 06:47	4 KB
gsi	2020-03-08 06:47	4 KB
local	2020-03-08 06:47	4 KB
lost+found	2020-03-08 06:47	4 KB

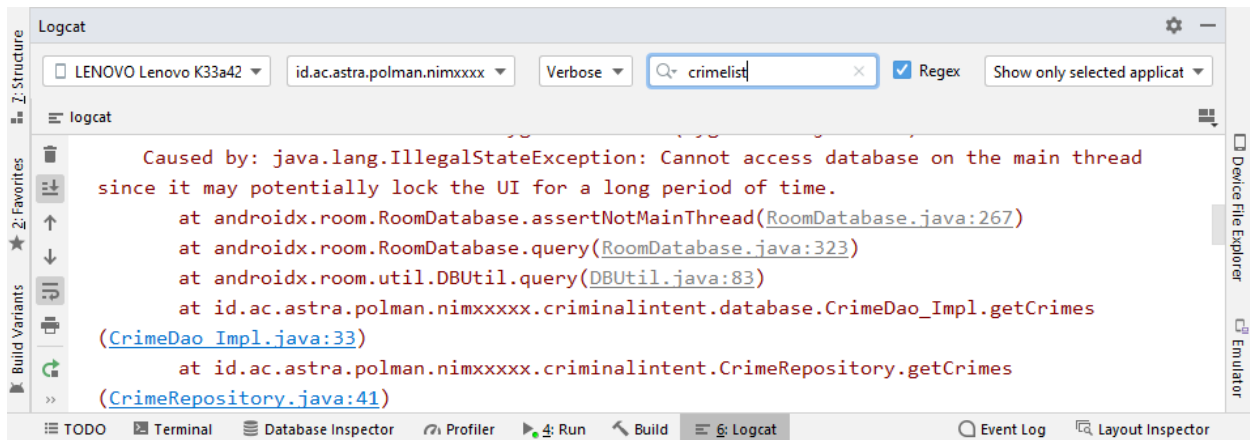
64. Currently, your `CrimeListViewModel` creates 100 fake crimes to display in the list. Remove the code that generates the fake crimes and replace it with a call to the `getCrimes()` function on your `CrimeRepository`.

```

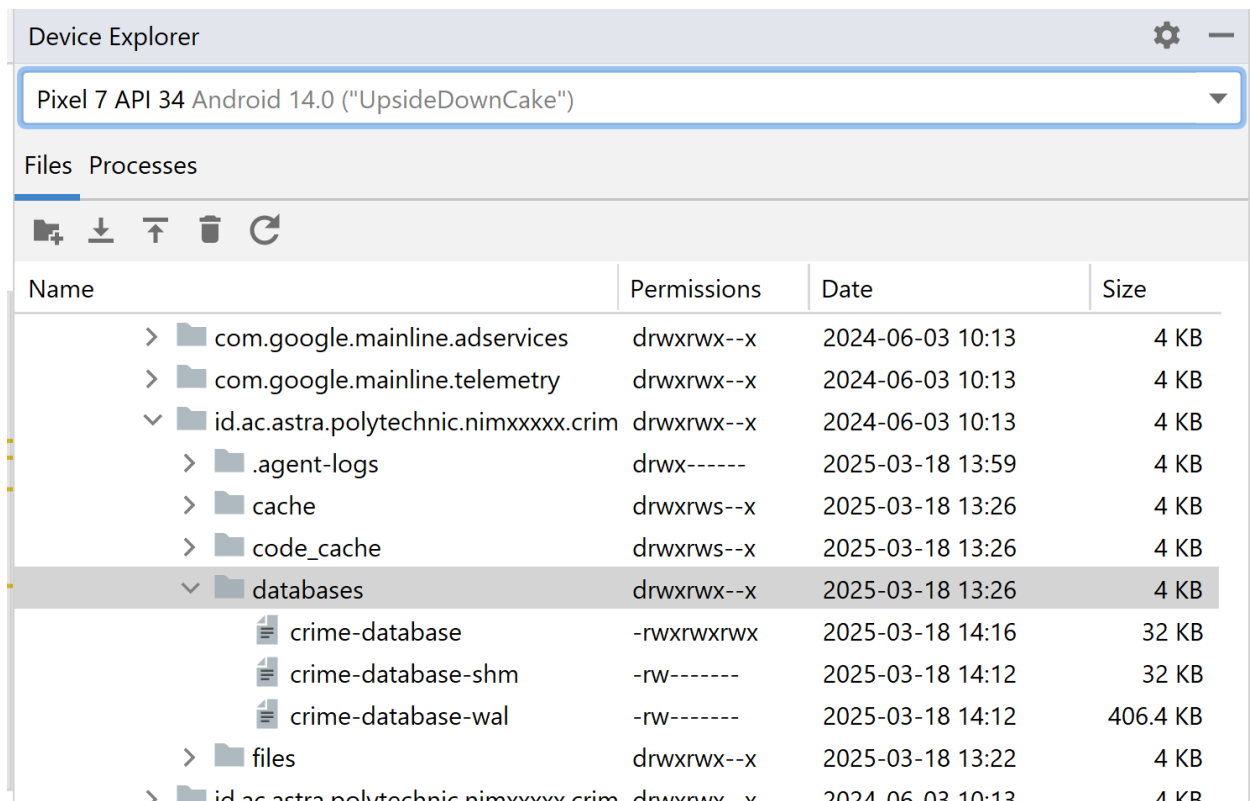
CrimeListViewModel.java
8   public class CrimeListViewModel extends ViewModel {
9
10      private List<Crime> mCrimes;
11      private CrimeRepository mCrimeRepository;
12
13      public CrimeListViewModel(){
14          mCrimeRepository = CrimeRepository.get();
15          mCrimes = mCrimeRepository.getCrimes();
16      }
17
18      public List<Crime> getCrimes(){
19          /*
20             if (mCrimes == null) {
21                 mCrimes = new ArrayList<>();
22                 loadCrimes();
23             }*/
24          return mCrimes;
25      }
26
27      /*
28         private void loadCrimes() {
29             // usually do asynchronous operation
30             for (int i = 0; i < 100; i++){
31                 Crime crime = new Crime();
32                 crime.setTitle("Crime #" + i);
33                 crime.setSolved(i % 2 == 0); //every other one
34                 mCrimes.add(crime);
35             }
36         }*/
37     }

```

65. Now, run your app to see the outcome. You may be surprised to find ... your app crashes.  
 66. Do not worry, this is the expected behavior. (At least, it is what we expected. Sorry.)  
 Take a look at the exception in Logcat to see what happened.

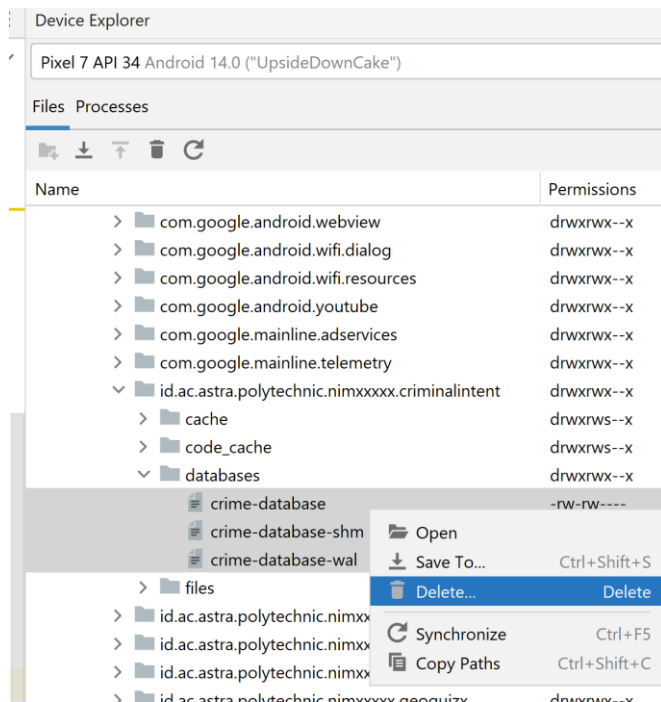


67. This error originates from the Room library. **Room is unhappy with your attempt to access the database on the main thread.** Throughout the remainder of this practice, you will learn about the threading model Android uses, as well as the purpose of the main thread and considerations for the type of work you run on the main thread. Additionally, **you will move your database interactions to a background thread.** That will make Room happy and get rid of the exception, which will in turn fix the crash.
68. For now, open the Device Explorer windows and expand our Apps folder database:



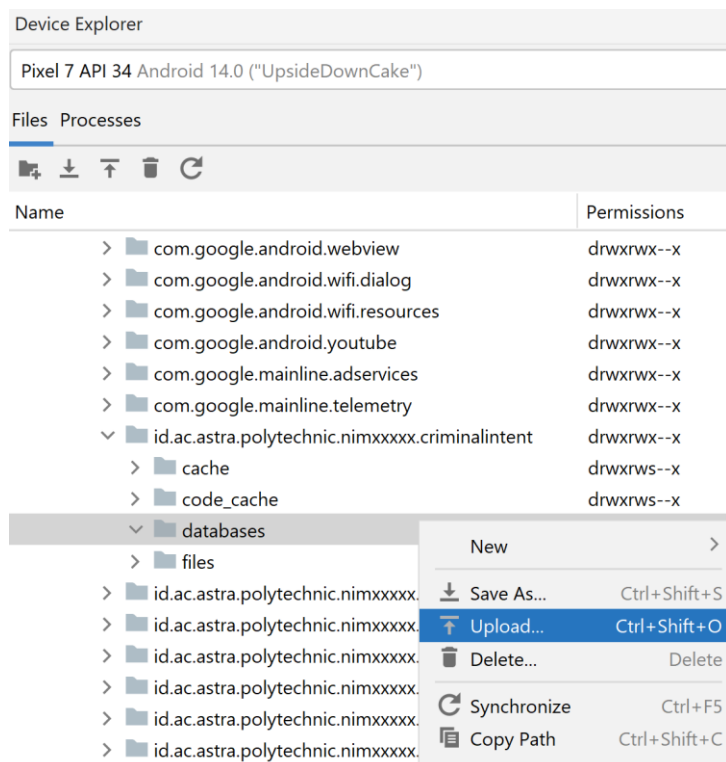
69. Select **all the files** on the databases folder, right click on all crime-database files, and choose **delete**.



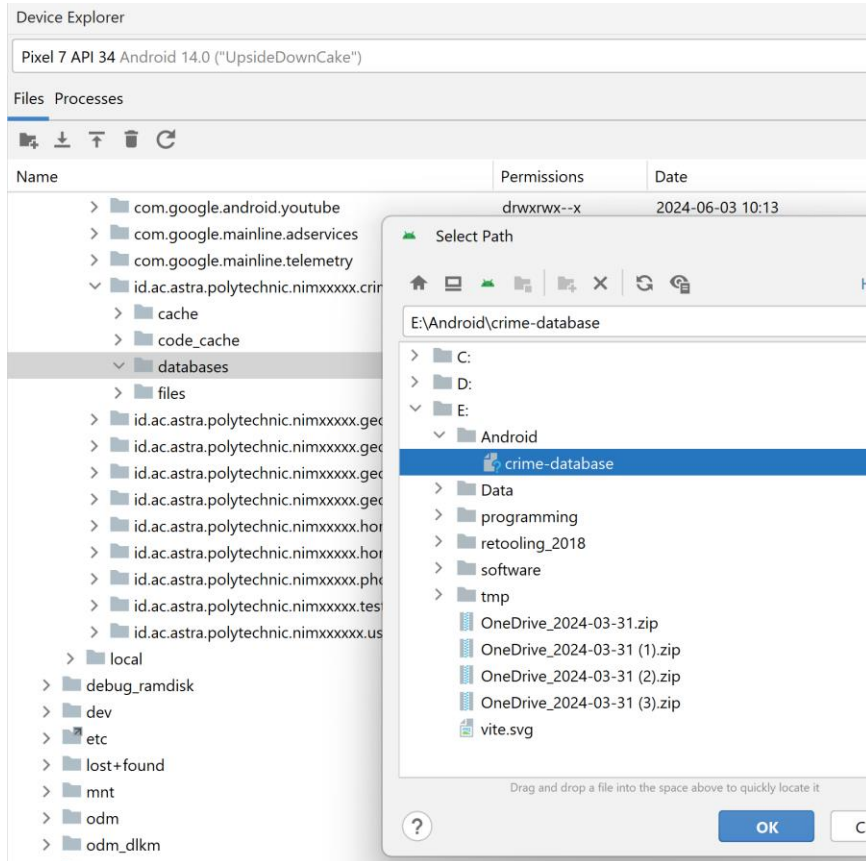


70. Now download the crime-database file from the server

71. And at databases folder, right click, and choose upload



72. Pick the crime-database file that you already downloaded, and upload it

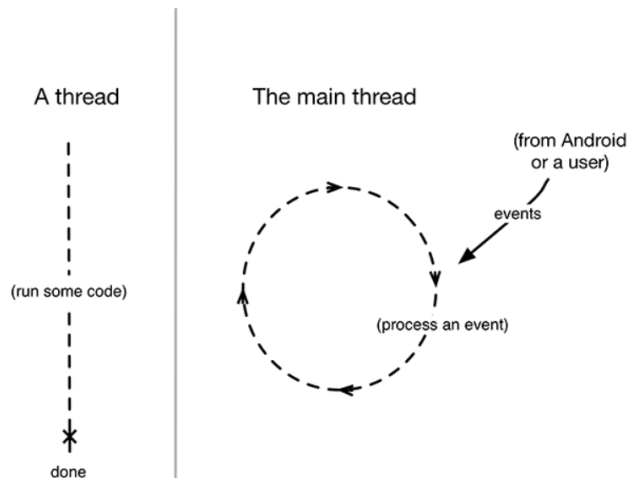


73. Klik OK.

74. Now we have a crime-database that already populated with 100 dummy data in our apps.

## Application Threads

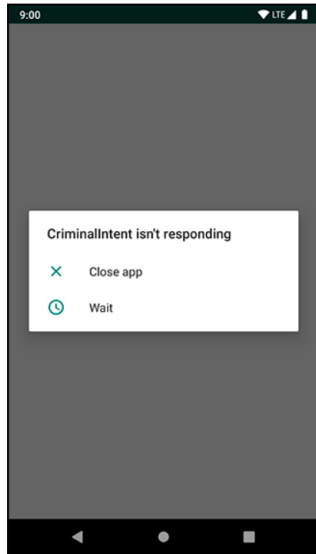
75. Reading from the database does not happen immediately. Because access can take so long, **Room disallows all database operations on the main thread**. If you try to violate this rule, Room will throw the `IllegalStateException` you have just seen.
76. Why? To understand that, you need to understand what a thread is, what the main thread is, and what the main thread does.
77. A thread is a single sequence of execution. Code running within a single thread will execute one step after another. Every Android app starts life with a `main` thread. The main thread, however, is not a preordained list of steps. Instead, it sits in an infinite loop and waits for events initiated by the user or the system. Then it executes code in response to those events as they occur.



78. Imagine that your app is an enormous shoe store and that you only have one employee – The Flash. (Who hasn't dreamed of that?) There are a lot of things to do in a store to keep the customers happy: arranging the merchandise, fetching shoes for customers, wielding the Brannock device. The Flash is so fast that everyone is taken care of in a timely fashion, even though there is only one guy doing all the work.
79. For this situation to work, The Flash cannot spend too much time doing any one thing. What if a shipment of shoes goes missing? Someone will have to spend a lot of time on the phone straightening it out. Your customers will get mighty impatient waiting for shoes while The Flash is on hold.
80. The Flash is like the main thread in your application. It runs all the code that updates the UI. This includes the code executed in response to different UI-related events – activity startup, button presses, and so on. (Because the events are all related to the UI in some way, **the main thread is sometimes called the UI thread.**)
81. The event loop keeps the UI code in sequence. It makes sure that none of these operations step on each other, while still ensuring that the code is executed in a timely fashion. So far, all of the code you have written has been executed on the main thread.

## Background threads

82. Database access is a lot like a phone call to your shoe distributor: It takes a long time compared to other tasks. During that time, the UI will be completely unresponsive, which might result in an application not responding, or ANR.
83. An ANR occurs when Android's watchdog determines that the main thread has failed to respond to an important event, like pressing the Back button. To the user, it looks like below



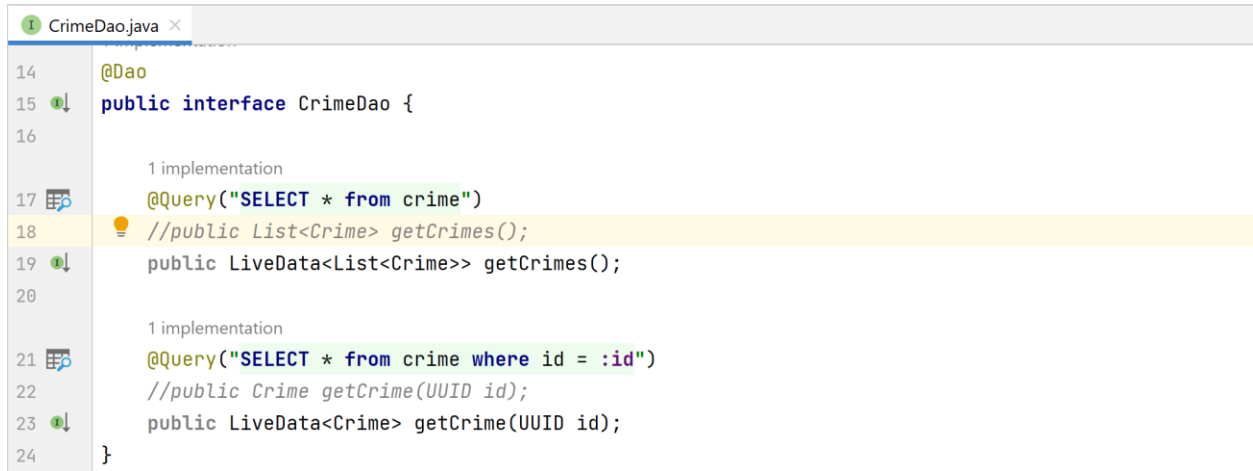
84. In your store, you would solve the problem by (naturally) hiring a second Flash to call the shoe distributor. In Android, you do something similar – you create a background thread and access the database from there.
85. There are two important rules to consider when you start adding background threads to your apps:
- All long-running tasks should be done on a background thread.** This ensures that your main thread is free to handle UI-related tasks to keep the UI responsive for your users.
  - The UI can only be updated from the main thread.** You will get an error if you try to modify the UI from a background thread, so you need to make sure any data generated from a background thread is sent to the main thread to update the UI.
86. There are many ways to execute work on a background thread on Android. You will learn how to make asynchronous network requests later.
87. For CriminalIntent, you will use two options to execute your database calls in the background. In this practice you will use `LiveData` to wrap your query data.

## Using LiveData

88. `LiveData` is a data holder class found in the `Jetpack lifecycle-extensions` library. Room is built to work with `LiveData`. Since you already included the `lifecycle-extensions` library in your `app/build.gradle` file, you have access to the `LiveData` class in your project.
89. `LiveData`'s goal is to simplify passing data between different parts of your application, such as from your `CrimeRepository` to the fragment that needs to display the crime data. `LiveData` also enables passing data between threads. This makes it a perfect fit for respecting the threading rules we laid out above.
90. When you configure queries in your Room DAO to return `LiveData`, Room will automatically execute those query operations on a background thread and then publish

the results to the `LiveData` object when the query is done. You can set your activity or fragment up to observe the `LiveData` object, in which case your activity or fragment will be notified on the main thread of results when they are ready.

91. In this practice, you will focus on using the cross-thread communication functionality of `LiveData` to perform your database queries. To begin, open `CrimeDao.java` and update the return type of your query functions to return a `LiveData` object that wraps the original return type.
92. While on it, we also add methods for inserting and deleting data.



```
14 @Dao
15 public interface CrimeDao {
16
17     1 implementation
18     @Query("SELECT * from crime")
19     //public List<Crime> getCrimes();
20     public LiveData<List<Crime>> getCrimes();
21
22     1 implementation
23     @Query("SELECT * from crime where id = :id")
24     //public Crime getCrime(UUID id);
25     public LiveData<Crime> getCrime(UUID id);
26 }
```

93. By returning an instance of `LiveData` from your DAO class, you signal Room to run your query on a background thread. When the query completes, the `LiveData` object will handle sending the crime data over to the main thread and notify any observers.

94. Next, update `CrimeRepository` to return `LiveData` from its query functions.

```
69
70 public static void initialize(Context context){
71     if (INSTANCE == null){
72         INSTANCE = new CrimeRepository(context);
73     }
74 }
75
76 public static CrimeRepository get(){
77     return INSTANCE;
78 }
79
80 //public List<Crime> getCrimes(){
81 public LiveData<List<Crime>> getCrimes(){
82     return mCrimeDao.getCrimes();
83 }
84
85 //public Crime getCrime(UUID id){
86 public LiveData<Crime> getCrime(UUID id){
87     return mCrimeDao.getCrime(id);
88 }
89 }
```

## Observing LiveData

95. To display the crimes from the database in the crime list screen, update `CrimeListFragment` to observe the `LiveData` returned from `CrimeRepository.getCrimes()`.

96. First, open `CrimeListViewModel.java` and rename the `mCrimes` field member so it is more clear what data the it holds.

```

CrimeListViewModel.java x
9      public class CrimeListViewModel extends ViewModel {
10
11          //private List<Crime> mCrimes;
12          private LiveData<List<Crime>> mCrimeListLiveData;
13          private CrimeRepository mCrimeRepository;
14
15          public CrimeListViewModel(){
16              mCrimeRepository = CrimeRepository.get();
17              //mCrimes = mCrimeRepository.getCrimes();
18              mCrimeListLiveData = mCrimeRepository.getCrimes();
19          }
20
21          /*      public List<Crime> getCrimes(){
22                  if (mCrimes == null) {
23                      mCrimes = new ArrayList<>();
24                      loadCrimes();
25                  }
26                  return mCrimes;
27              }*/
28
29          public LiveData<List<Crime>> getCrimes(){
30              return mCrimeListLiveData;
31          }
32
33          /*      private void loadCrimes() {
34                  // usually do asynchronous operation
35                  for (int i = 0; i < 100; i++){
36                      Crime crime = new Crime();
37                      crime.setTitle("Crime #" + i);
38                      crime.setSolved(i % 2 == 0); //every other one
39                      mCrimes.add(crime);
40                  }
41              }*/
42
43      }

```

97. Next, clean up CrimeListFragment to reflect the fact that CrimeListViewModel now exposes the LiveData returned from your repository. Update the onCreate (...) implementation, since it references crimeListViewModel.crimes, which no longer exists (and since you no longer need the logging you put in place there). In updateUI (), remove the reference to crimeListViewModel.crimes and add a parameter to accept a list of crimes as input.
98. Finally, remove the call to updateUI () from onCreateView (...). You will implement a call to updateUI () from another place shortly.

```

CrimeListFragment.java x
23 public class CrimeListFragment extends Fragment {
24     private static final String TAG = "CrimeListFragment";
25
26     private CrimeListViewModel mCrimeListViewModel;
27     private RecyclerView mCrimeRecyclerView;
28     private CrimeAdapter mAdapter;
29
30     //private void updateUI(){
31     private void updateUI(List<Crime> crimes){
32         //List<Crime> crimes = mCrimeListViewModel.getCrimes();
33         mAdapter = new CrimeAdapter(crimes);
34         mCrimeRecyclerView.setAdapter(mAdapter);
35     }
36
37     @Override
38     public void onCreate(@Nullable Bundle savedInstanceState) {
39         super.onCreate(savedInstanceState);
40         mCrimeListViewModel = new ViewModelProvider(this).get(CrimeListViewModel.class);
41         //Log.d(TAG, "Total crimes: " + mCrimeListViewModel.getCrimes().size());
42     }
43
44     @Nullable
45     @Override
46     public View onCreateView(@NonNull LayoutInflater inflater,
47                             @Nullable ViewGroup container,
48                             @Nullable Bundle savedInstanceState) {
49         View view = inflater.inflate(R.layout.fragment_crime_list, container, false);
50         mCrimeRecyclerView = (RecyclerView) view.findViewById(R.id.crime_recycler_view);
51         mCrimeRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));
52         //updateUI();
53         return view;
54     }

```

99. Now, update `CrimeListFragment` to observe the `LiveData` that wraps the list of crimes returned from the database. Since the fragment will have to wait for results from the database before it can populate the recycler view with crimes, initialize the recycler view adapter with an empty crime list to start. Then set up the recycler view adapter with the new list of crimes when new data is published to the `LiveData`.



```

CrimeListFragment.java x
23 public class CrimeListFragment extends Fragment {
24     private static final String TAG = "CrimeListFragment";
25
26     private CrimeListViewModel mCrimeListViewModel;
27     private RecyclerView mCrimeRecyclerView;
28     private CrimeAdapter mAdapter = new CrimeAdapter(Collections.<Crime>emptyList());
29
...
37     @Override
38     public void onCreate(@Nullable Bundle savedInstanceState) {
39         super.onCreate(savedInstanceState);
40         mCrimeListViewModel = new ViewModelProvider(this).get(CrimeListViewModel.class);
41         //Log.d(TAG, "Total crimes: " + mCrimeListViewModel.getCrimes().size());
42     }
43
44     @Nullable
45     @Override
46     public View onCreateView(@NonNull LayoutInflater inflater,
47                             @Nullable ViewGroup container,
48                             @Nullable Bundle savedInstanceState) {
49         View view = inflater.inflate(R.layout.fragment_crime_list, container, false);
50         mCrimeRecyclerView = (RecyclerView) view.findViewById(R.id.crime_recycler_view);
51         mCrimeRecyclerView.setLayoutManager(new LinearLayoutManager(getActivity()));
52         //updateUI();
53         mCrimeRecyclerView.setAdapter(mAdapter);
54         return view;
55     }
56
57     @Override
58     public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
59         super.onViewCreated(view, savedInstanceState);
60         mCrimeListViewModel.getCrimes().observe(
61             getViewLifecycleOwner(),
62             new Observer<List<Crime>>() {
63                 @Override
64                 public void onChanged(List<Crime> crimes) {
65                     //Update the cached copy of
66                     updateUI(crimes);
67                     Log.i(TAG, "Got crimes = " + crimes.size());
68                 }
69             }
70         );
71     }

```

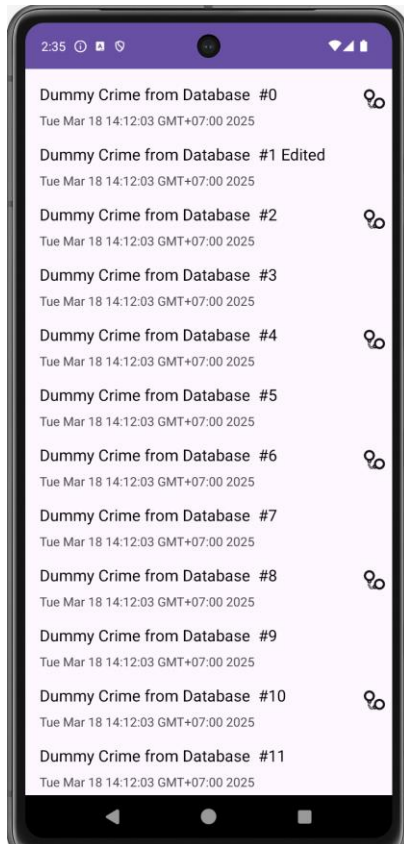
100. The `LiveData.observe(LifecycleOwner, Observer)` function is used to register an observer on the `LiveData` instance and tie the life of the observation to the life of another component, such as an activity or fragment.

101. The second parameter to the `observe(...)` function is an `Observer` implementation. This object is responsible for reacting to new data from the `LiveData`. In this case, the observer's code block is executed whenever the `LiveData`'s list of crimes gets updated. The observer receives a list of crimes from the `LiveData` and prints a log statement if the field member is not null.
102. If you never unsubscribe or cancel your `Observer` from listening to the `LiveData`'s changes, your `Observer` implementation might try to update your fragment's view when the view is in an invalid state (such as when the view is being torn down). And if you attempt to update an invalid view, your app can crash.
103. This is where the `LifecycleOwner` parameter to `LiveData.observe(...)` comes in. The lifetime of the `Observer` you provide is scoped to the lifetime of the Android component represented by the `LifecycleOwner` you provide. In the code above, you scope the observer to the life of the fragment's view.
104. As long as the `lifecycle owner` you scope your observer to is in a valid lifecycle state, the `LiveData` object will notify the observer of any new data coming in. The `LiveData` object will automatically unregister the `Observer` when the associated lifecycle is no longer in a valid state. Because `LiveData` reacts to changes in a lifecycle, it is called a `lifecycle-aware component`.
105. A lifecycle owner is a component that implements the `LifecycleOwner` interface and contains a `Lifecycle` object. A `Lifecycle` is an object that keeps track of an Android lifecycle's current state. (Recall that activities, fragments, views, and even the application process itself all have their own lifecycle.) The lifecycle states, such as `created` and `resumed`, are enumerated in `Lifecycle.State`. You can query a `Lifecycle`'s state using `Lifecycle.getCurrentState()` or register to be notified of changes in state.
106. The `AndroidX Fragment` is a lifecycle owner directly – `Fragment` implements `LifecycleOwner` and has a `Lifecycle` object representing the state of the fragment instance's lifecycle.
107. A fragment's view lifecycle is owned and tracked separately by `FragmentViewLifecycleOwner`. Each `Fragment` has an instance of `FragmentViewLifecycleOwner` that keeps track of the lifecycle of that fragment's view.
108. In the code above, you scope the observation to the fragment's view lifecycle, rather than the lifecycle of the fragment itself, by passing the `viewLifecycleOwner` to the `observe(...)` function. The fragment's view lifecycle, though separate from the lifecycle of the `Fragment` instance itself, mirrors the lifecycle of the fragment. It is possible to change this default behavior by retaining the fragment (which you will not do in `CriminalIntent`).
109. `Fragment.onViewCreated(...)` is called after `Fragment.onCreateView(...)` returns, signaling that the fragment's view hierarchy is in place. You observe the `LiveData` from `onViewCreated(...)` to ensure that your view is ready to display the crime data. This is also the reason you pass the `viewLifecycleOwner` to the `observe()` function, rather than the fragment itself. You only want to receive the list of crimes while your view is in a good state, so using the

view's lifecycle owner ensures that you will not receive updates when your view is not on the screen.

110. When the list of crimes is ready, the observer you defined prints a log message and sends the list to the `updateUI()` function to prepare the adapter.

111. With everything in place, run `CriminalIntent`. You should no longer see the crash you saw earlier. Instead, you should see the fake crimes from the database.



#####

### Notes:

1. Create folder PRG6\_M6\_P4\_XXXXXXXXXX.
2. Zip the folder and submit it to the server.

### Bibliography:

- Marsicano, et. al., “Android Programming – The Big Nerd Ranch”, 5<sup>th</sup> Ed, 2022, Pearson Technology.