Now let's talk about Room

Room is a persistence library, part of the Android Jetpack.

Here is the video from Android Developers Channel:

Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

Room is now considered as a better approach for data persistence than SQLiteDatabase. It makes it easier to work with SQLiteDatabase objects in your app, decreasing the amount of **boilerplate** code and verifying SQL queries at compile time.

Why use Room?

- Compile-time verification of SQL queries. each @Query and @Entity is checked at the compile time, that preserves your app from crash issues at runtime and not only it checks the only syntax, but also missing tables.
- Boilerplate code
- Easily integrated with other Architecture components (like LiveData)

Major problems with SQLite usage are

- There is no compile-time verification of raw SQL queries. For example, if you write a SQL query with a wrong column name that does not exist in real database then it will give exception during run time and you can not capture this issue during compile time.
- As your schema changes, you need to update the affected SQL queries manually. This process can be time-consuming and error-prone.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects (POJO).

Room vs SQLite

Room is an ORM, Object Relational Mapping library. In other words, Room will map our database objects to Java objects. Room provides an abstraction layer over SQLite to allow fluent database access while harnessing the full power of SQLite.

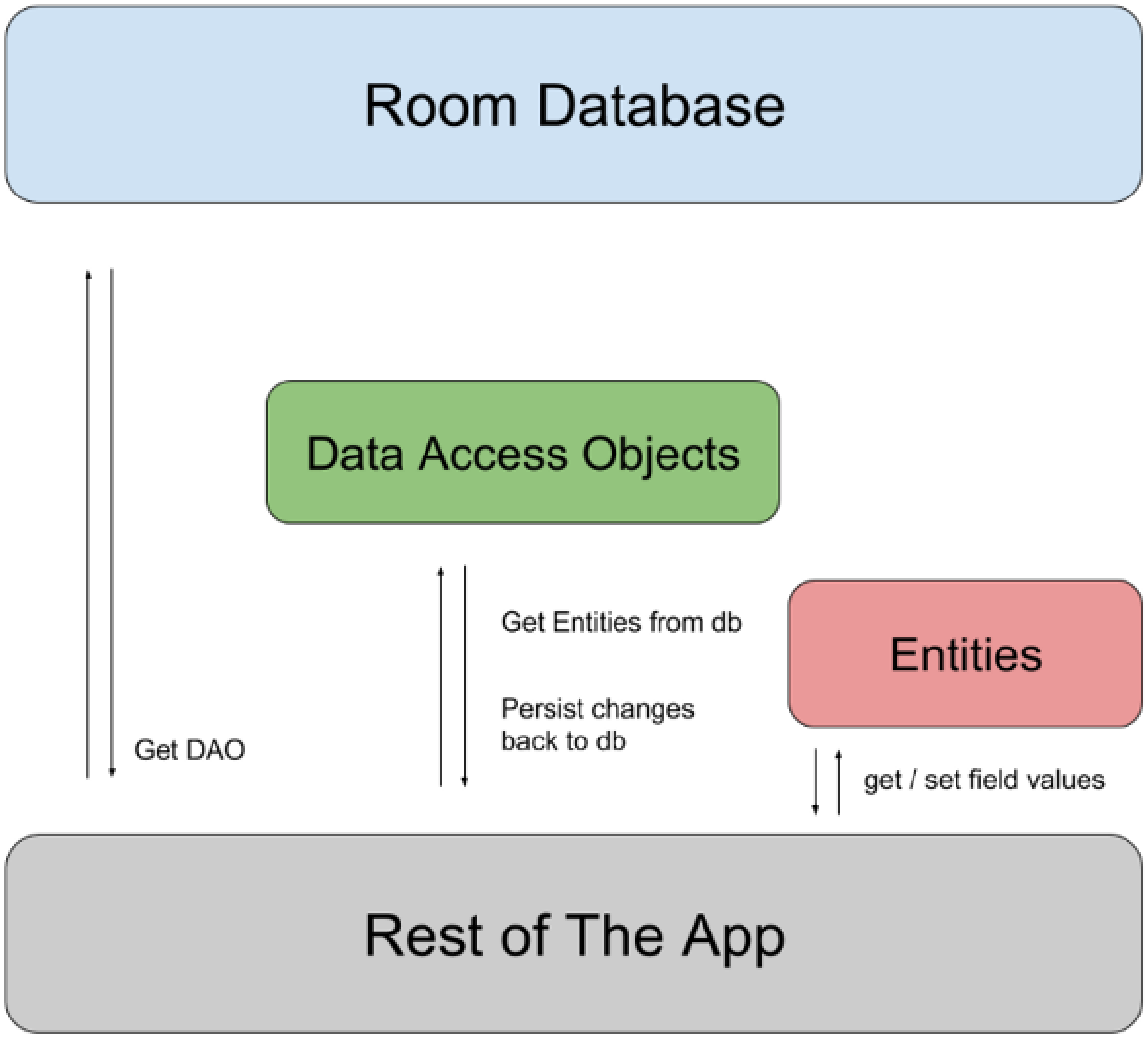
Difference between **SQLite** and **Room** persistence library:-

- In the case of SQLite, There is no compile-time verification of raw SQLite queries. But in Room, there is SQL validation at compile time.
- You need to use lots of boilerplate code to convert between SQL queries and Java data objects. But, Room maps our database objects to Java Object without boilerplate code.

- As your schema changes, you need to update the affected SQL queries manually. Room solves this problem.
- Room is built to work with LiveData and RxJava for data observation, while SQLite does not.

Room is awesome

Components of Room DB



Room has three main components of Room DB :

- Entity
- Dao
- Database

1. Entity

Represents a table within the database. Room creates a table for each class that has `@Entity` annotation, the fields in the class correspond to columns in the table. Therefore, the entity classes tend to be small model classes that don’t contain any logic.

Entities annotations

Before we get started with modeling our entities, we need to know some useful annotations and their attributes.

@Entity — every model class with this annotation will have a mapping table in DB

- foreignKeys — names of foreign keys
- indices — list of indicates on the table
- primaryKeys — names of entity primary keys
- tableName

@PrimaryKey — as its name indicates, this annotation points the primary key of the entity. autoGenerate — if set to true, then SQLite will be generating a unique id for the column

```
@PrimaryKey(autoGenerate = true)
```

@ColumnInfo — allows specifying custom information about column

```
@ColumnInfo(name = “column_name”)
```

@Ignore — field will not be persisted by Room

@Embeded — nested fields can be referenced directly in the SQL queries.

2. Dao

DAOs are responsible for defining the methods that access the database. In the initial SQLite, we use the `Cursor` objects. With Room, we don’t need all the `Cursor` related code and can simply define our queries using annotations in the `Dao` class.

3. Database

Contains the database holder and serves as the main access point for the underlying connection to your app’s persisted, relational data.

To create a database we need to define an abstract class that extends `RoomDatabase`. This class is annotated with `@Database`, lists the entities contained in the database, and the DAOs which access them.

The class that's annotated with `@Database` should satisfy the following conditions:

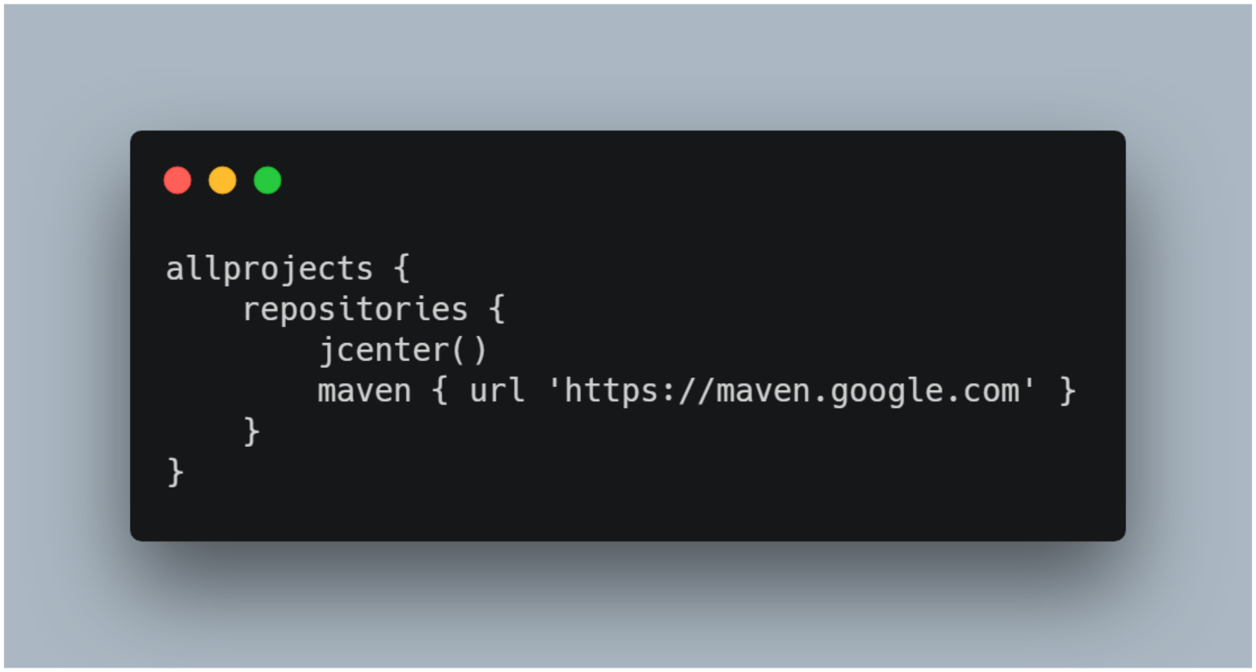
- Be an abstract class that extends `RoomDatabase`.
- Include the list of entities associated with the database within the annotation.
- Contain an abstract method that has 0 arguments and returns the class that is annotated with `@Dao`.

At runtime, you can acquire an instance of `Database` by calling `Room.databaseBuilder()` or`Room.inMemoryDatabaseBuilder()`.

Implementation of Room

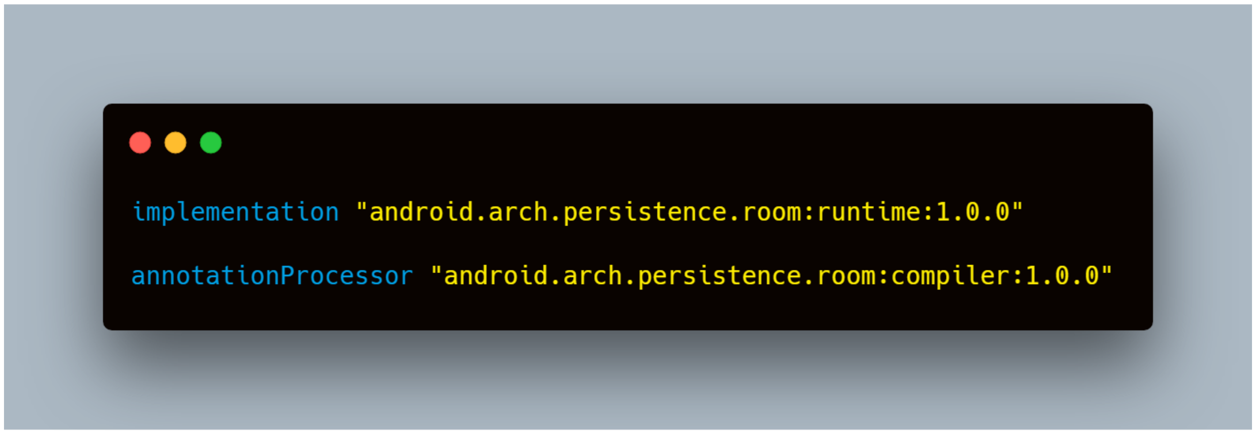
Step 1: Add the Gradle dependencies

1. To add it to your project, open the project level `build.gradle` file and add the highlighted line as shown below:



```
allprojects {
    repositories {
        jcenter()
        maven { url 'https://maven.google.com' }
    }
}
```

2. Open the `build.gradle` file for **your app or module** and add dependencies:

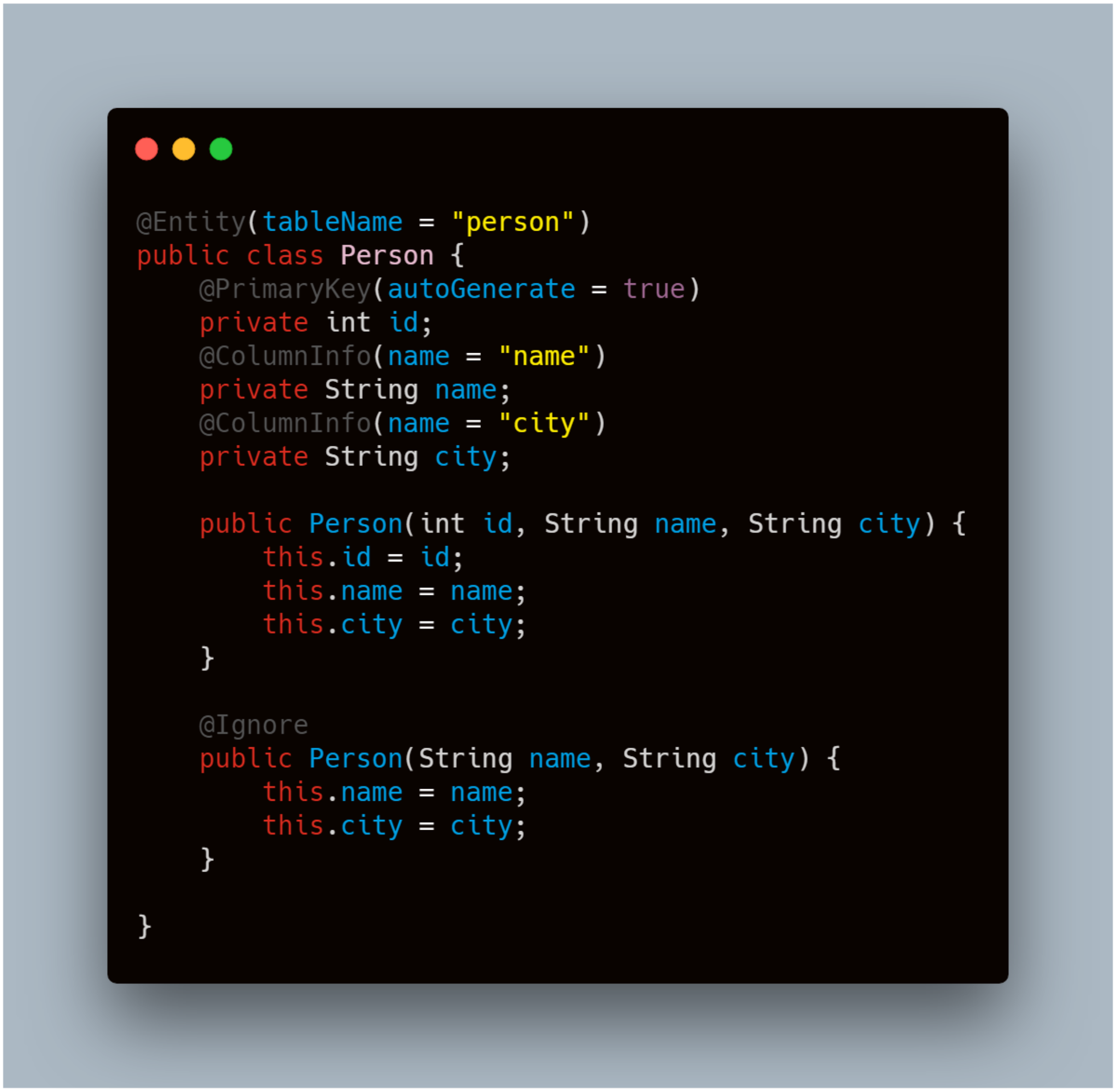


```
implementation "android.arch.persistence.room:runtime:1.0.0"

annotationProcessor "android.arch.persistence.room:compiler:1.0.0"
```

Step 2: Create a Model Class

Room creates a table for each class annotated with `@Entity`; the fields in the class correspond to columns in the table. Therefore, the entity classes tend to be small model classes that don't contain any logic. Our `Person` class represents the model for the data in the database. So let's update it to tell Room that it should create a table based on this class:

- Annotate the class with `@Entity` and use the `tableName` property to set the name of the table.
- Set the primary key by adding the `@PrimaryKey` annotation to the correct fields — in our case, this is the ID of the User.
- Set the name of the columns for the class fields using the `@ColumnInfo(name = "column_name")` annotation. Feel free to skip this step if your fields already have the correct column name.
- If multiple constructors are suitable, add the `@Ignore` annotation to tell Room which should be used and which not.

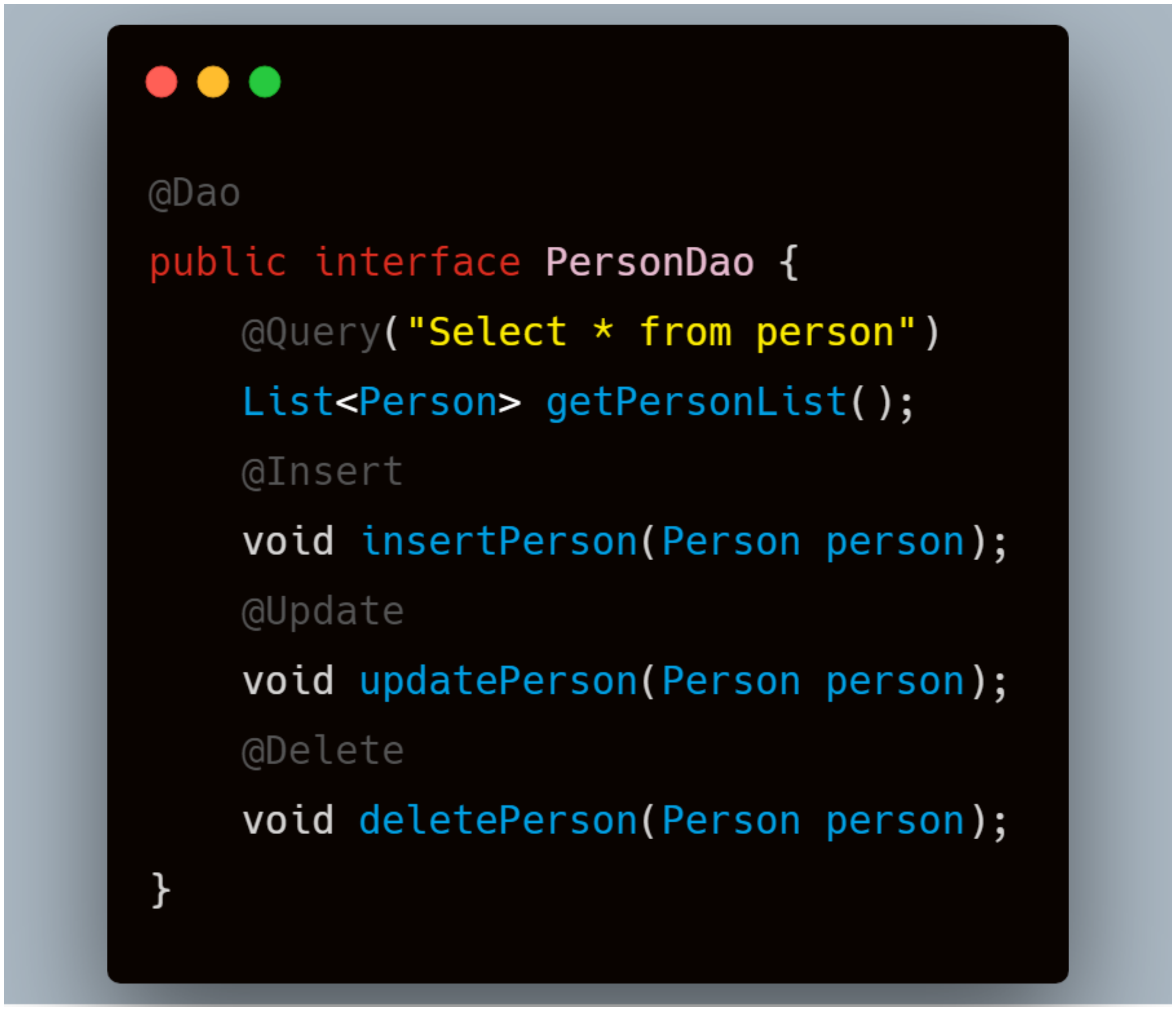


```java
@Entity(tableName = "person")
public class Person {
    @PrimaryKey(autoGenerate = true)
    private int id;
    @ColumnInfo(name = "name")
    private String name;
    @ColumnInfo(name = "city")
    private String city;

    public Person(int id, String name, String city) {
        this.id = id;
        this.name = name;
        this.city = city;
    }

    @Ignore
    public Person(String name, String city) {
        this.name = name;
        this.city = city;
    }

}
```

Step 3: Create Data Access Objects (DAOs)

DAOs are responsible for defining the methods that access the database.

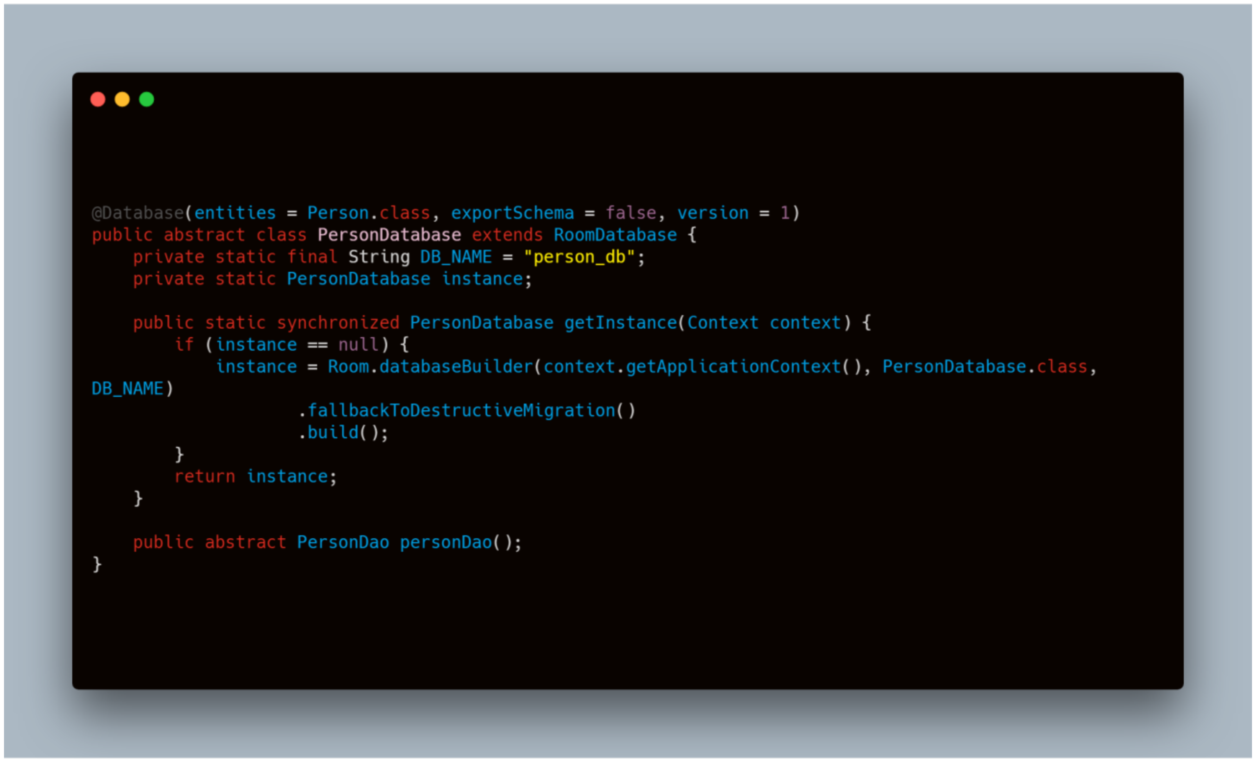
To create a DAO we need to create an interface and annotated with @Dao .



```
@Dao
public interface PersonDao {
    @Query("Select * from person")
    List<Person> getPersonList();
    @Insert
    void insertPerson(Person person);
    @Update
    void updatePerson(Person person);
    @Delete
    void deletePerson(Person person);
}
```

Step 4 — Create the database

To create a database we need to define an abstract class that extends `RoomDatabase`. This class is annotated with `@Database`, lists the entities contained in the database, and the DAOs which access them.



```java
@Database(entities = Person.class, exportSchema = false, version = 1)
public abstract class PersonDatabase extends RoomDatabase {
    private static final String DB_NAME = "person_db";
    private static PersonDatabase instance;

    public static synchronized PersonDatabase getInstance(Context context) {
        if (instance == null) {
            instance = Room.databaseBuilder(context.getApplicationContext(), PersonDatabase.class,
DB_NAME)
                    .fallbackToDestructiveMigration()
                    .build();
        }
        return instance;
    }

    public abstract PersonDao personDao();
}
```

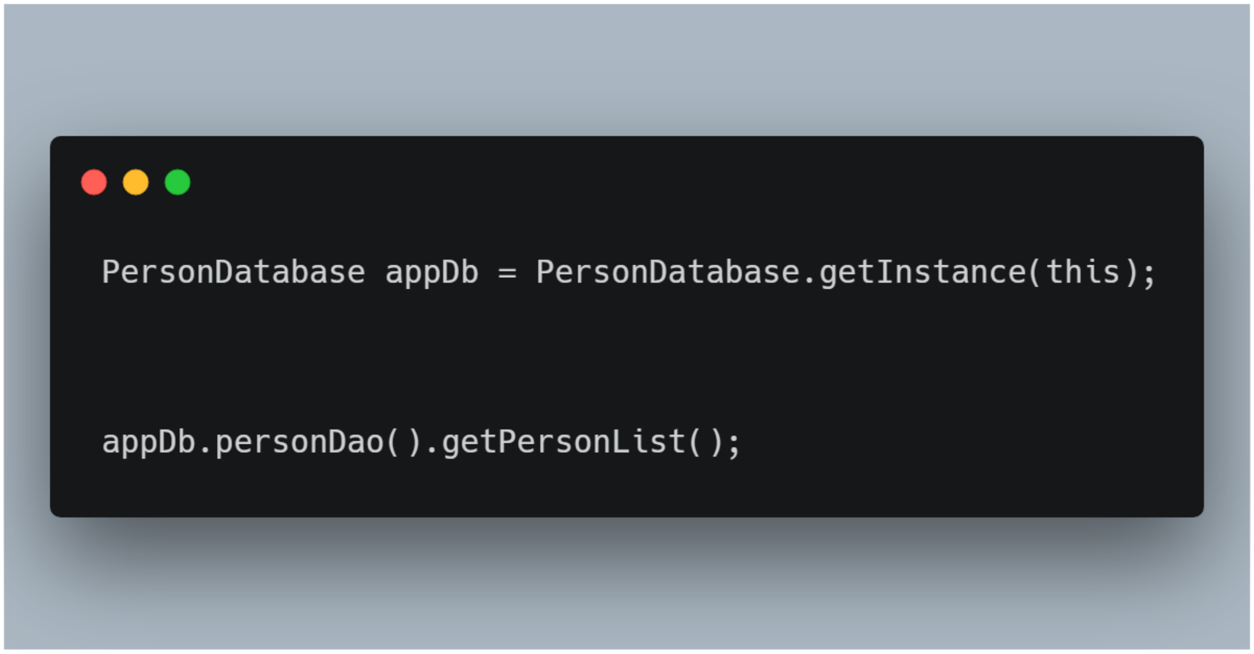
Step 4: Managing Data

To manage the data, first of all, we need to create an instance of the database.

then we can insert delete and update the database.

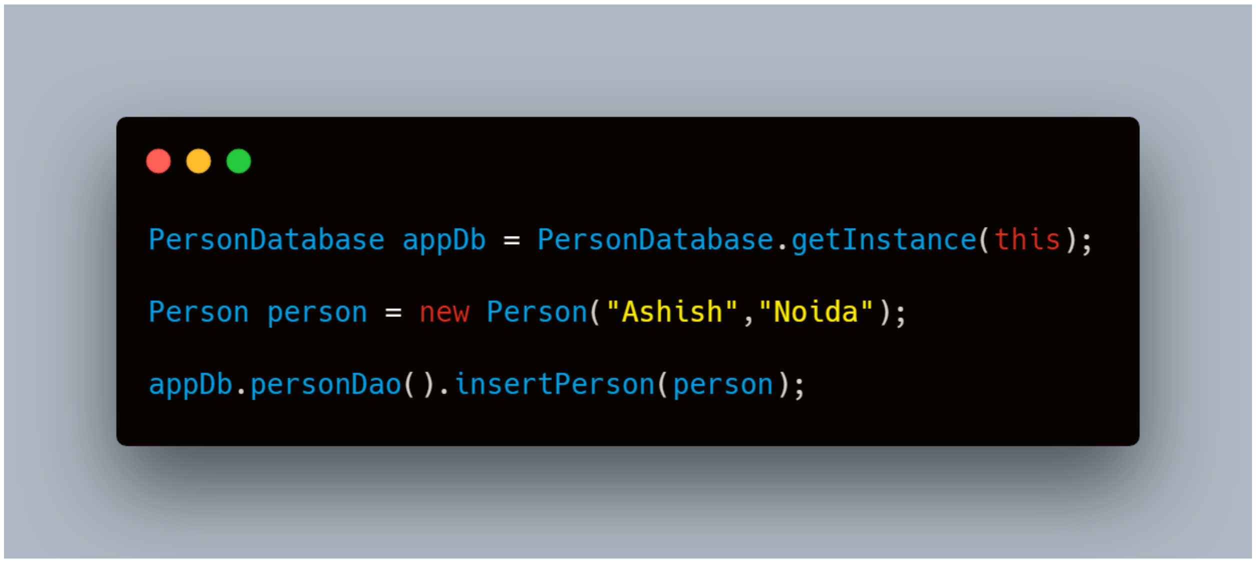
Make sure all the operation should execute on a different thread. In my case I use Executers (see here)

Query:



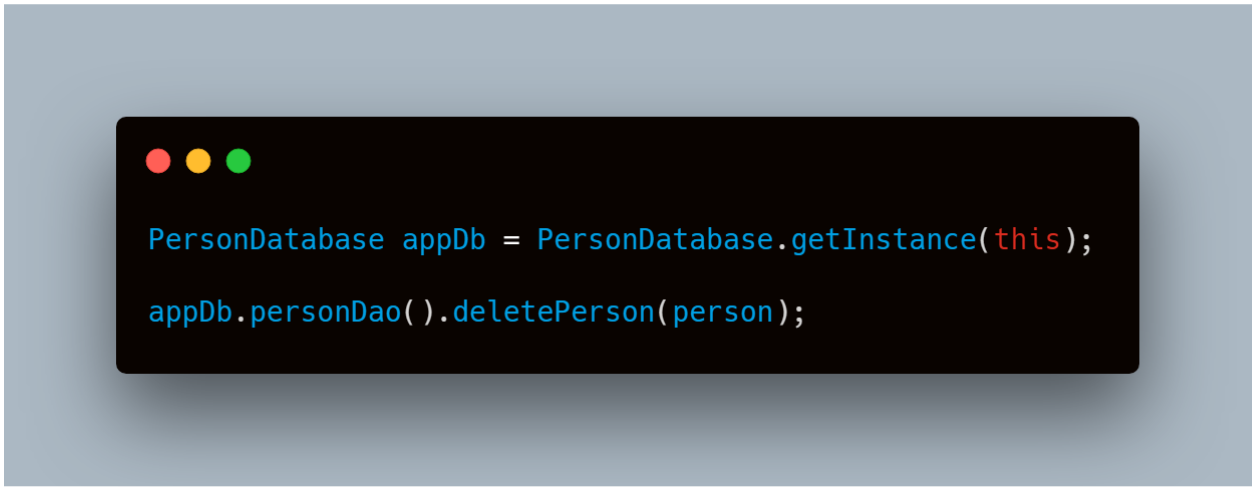
```
PersonDatabase appDb = PersonDatabase.getInstance(this);


appDb.personDao().getPersonList();
```

Insert:



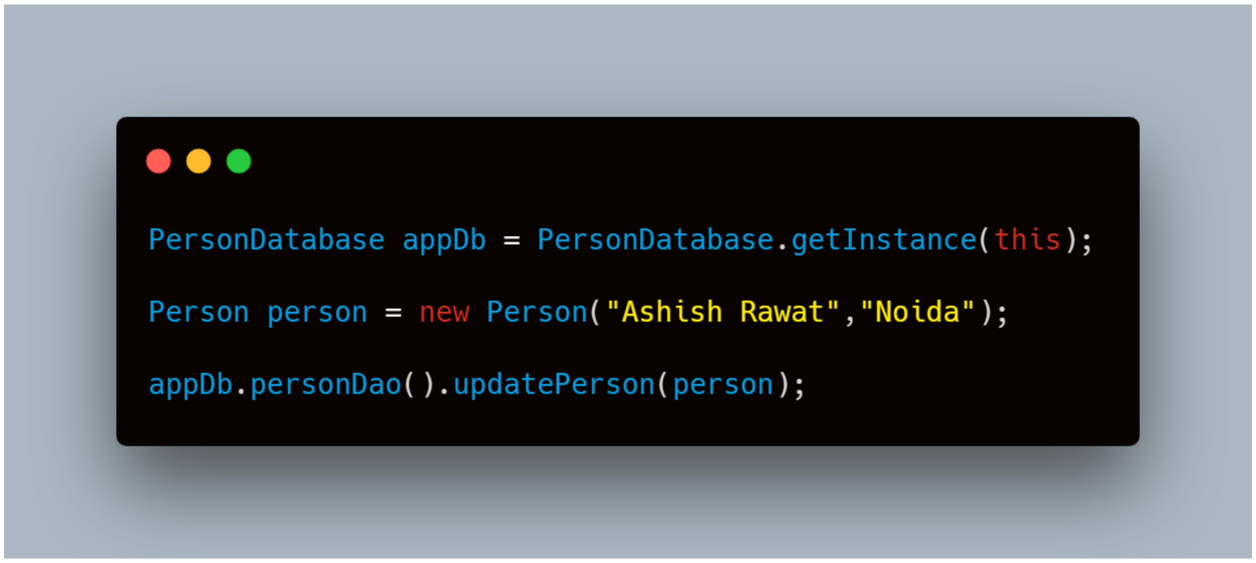
```
PersonDatabase appDb = PersonDatabase.getInstance(this);

Person person = new Person("Ashish","Noida");

appDb.personDao().insertPerson(person);
```

Delete:



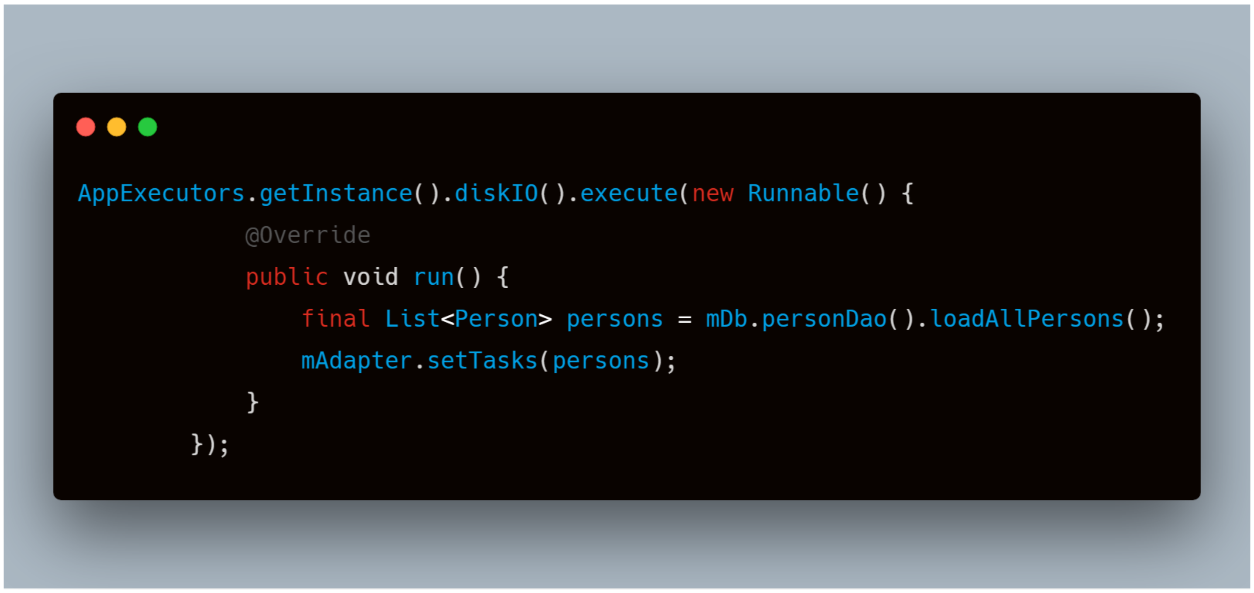
```
PersonDatabase appDb = PersonDatabase.getInstance(this);

appDb.personDao().deletePerson(person);
```

Update:



```
PersonDatabase appDb = PersonDatabase.getInstance(this);

Person person = new Person("Ashish Rawat","Noida");

appDb.personDao().updatePerson(person);
```

Executing with Executer



```
AppExecutors.getInstance().diskIO().execute(new Runnable() {
            @Override
            public void run() {
                final List<Person> persons = mDb.personDao().loadAllPersons();
                mAdapter.setTasks(persons);
            }
        });
```