

# Using Database in Android

---

# What is Database?

- A database is a structured collection of data that is organized and stored in a way that allows efficient storage, retrieval, and manipulation of information.
- They provide a structured way to store and organize data, ensuring data integrity, consistency, and security.
- Databases enable users and applications to perform operations like querying, inserting, updating, and deleting data while maintaining the overall integrity of the data.

# Database vs File

- Files store data in a less structured manner. They often store unstructured or semi-structured data, such as text, images, or documents. While files can be organised into folders and directories, they lack the relational structure of databases.
- Databases store data in a structured manner, typically using tables, rows, and columns. Each column represents a specific data attribute, and each row represents a record. This structured format allows for efficient querying, sorting, and filtering of data.

# Database basics - Table

- A table is a collection of related data organized into rows and columns. Each row represents a record, and each column represents a specific attribute of the data.
- Example: Table name: Employees

| EmployeeID | FirstName | LastName | Department | Salary  |
|------------|-----------|----------|------------|---------|
| 1          | John      | Smith    | HR         | \$60000 |
| 2          | Jane      | Doe      | Marketing  | \$55000 |
| 3          | Michael   | Johnson  | Sales      | \$70000 |
| 4          | Emily     | Williams | IT         | \$65000 |

← Row

↑  
Column

# Database Basic – Primary Key

- A primary key is a unique identifier for each row in a table. It ensures that each record is uniquely identified and helps maintain data integrity.

Primary key



| EmployeeID | FirstName | LastName | Department | Salary  |
|------------|-----------|----------|------------|---------|
| 1          | John      | Smith    | HR         | \$60000 |
| 2          | Jane      | Doe      | Marketing  | \$55000 |
| 3          | Michael   | Johnson  | Sales      | \$70000 |
| 4          | Emily     | Williams | IT         | \$65000 |

# Database basic – Foreign Key

- Foreign Key: A foreign key is a column in one table that references the primary key in another table. It establishes relationships between tables and ensures data consistency.
- Let's extend the previous "Employees" example to include a second table that demonstrates the use of a foreign key relationship. In this example, we'll add a "Departments" table and establish a relationship between the two tables using a foreign key.

# Database basic – Foreign Key (Cont..)

**Table Name: Departments**

| DepartmentID | DepartmentName |
|--------------|----------------|
| 1            | HR             |
| 2            | Marketing      |
| 3            | Sales          |
| 4            | IT             |

DepartmentID – Primary key

**Table Name: Employees**

| EmployeeID | FirstName | LastName | DepartmentID | Salary  |
|------------|-----------|----------|--------------|---------|
| 1          | John      | Smith    | 1            | \$60000 |
| 2          | Jane      | Doe      | 2            | \$55000 |
| 3          | Michael   | Johnson  | 3            | \$70000 |
| 4          | Emily     | Williams | 4            | \$65000 |

EmployeeID – Primary Key

DepartmentID – foreign key

# Android SQLite

- Android SQLite is a lightweight relational database management system (DBMS) that is included as a part of the Android operating system.
- It allows Android app developers to store and manage structured data locally within their applications.
- SQLite databases in Android are primarily used to store data that needs to be accessed and manipulated offline, such as user preferences, cached data, or small datasets.



# SQLite syntax – Create table

- create a table named "Students" with columns for "id" and "name"

```
CREATE TABLE table_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
    ...  
);
```

```
CREATE TABLE Students (  
    id INTEGER,  
    name TEXT  
);
```

```
CREATE TABLE Students (  
    id INTEGER PRIMARY KEY,  
    name TEXT  
);
```

With Primary key

```
CREATE TABLE Students (  
    id INTEGER PRIMARY KEY AUTOINCREMENT,  
    name TEXT  
);
```

With Auto increment

# SQLite syntax - INSERT

- To insert data into an SQLite table, you use the INSERT INTO statement. Here's how you would insert a new row of data into a table named "Students" with "id" and "name" columns

```
INSERT INTO Students (id, name)
VALUES (1, 'John Doe');
```

- If id is auto generated, you can omit it:

```
INSERT INTO Students (name)
VALUES ('Jane Smith');
```

# SQLite syntax - Update

- Suppose you have the following initial data:

| id | name       |
|----|------------|
| 1  | John Doe   |
| 2  | Jane Smith |

- And you want to update the name of the student with "id" 2 to "Jane Johnson". The SQL UPDATE statement would look like this:

```
UPDATE Students  
SET name = 'Jane Johnson'  
WHERE id = 2;
```

# SQLite syntax - Delete

- Delete all rows:

```
DELETE FROM Students
```

- Delete a specific row:

```
DELETE FROM Students  
WHERE id = 2;
```

# SQLite syntax - Select

- Select statement allows you to query the database and retrieve specific rows and columns of data based on your criteria.
- Suppose you have a table named "Students" with the following data:

| id | name        |
|----|-------------|
| 1  | John Doe    |
| 2  | Jane Smith  |
| 3  | Alice Brown |

- Select all data: 

```
SELECT * FROM Students;
```
- Select a specific column: 

```
SELECT name FROM Students;
```
- Select with condition: 

```
SELECT * FROM Students  
WHERE id > 1;
```

# Android database management

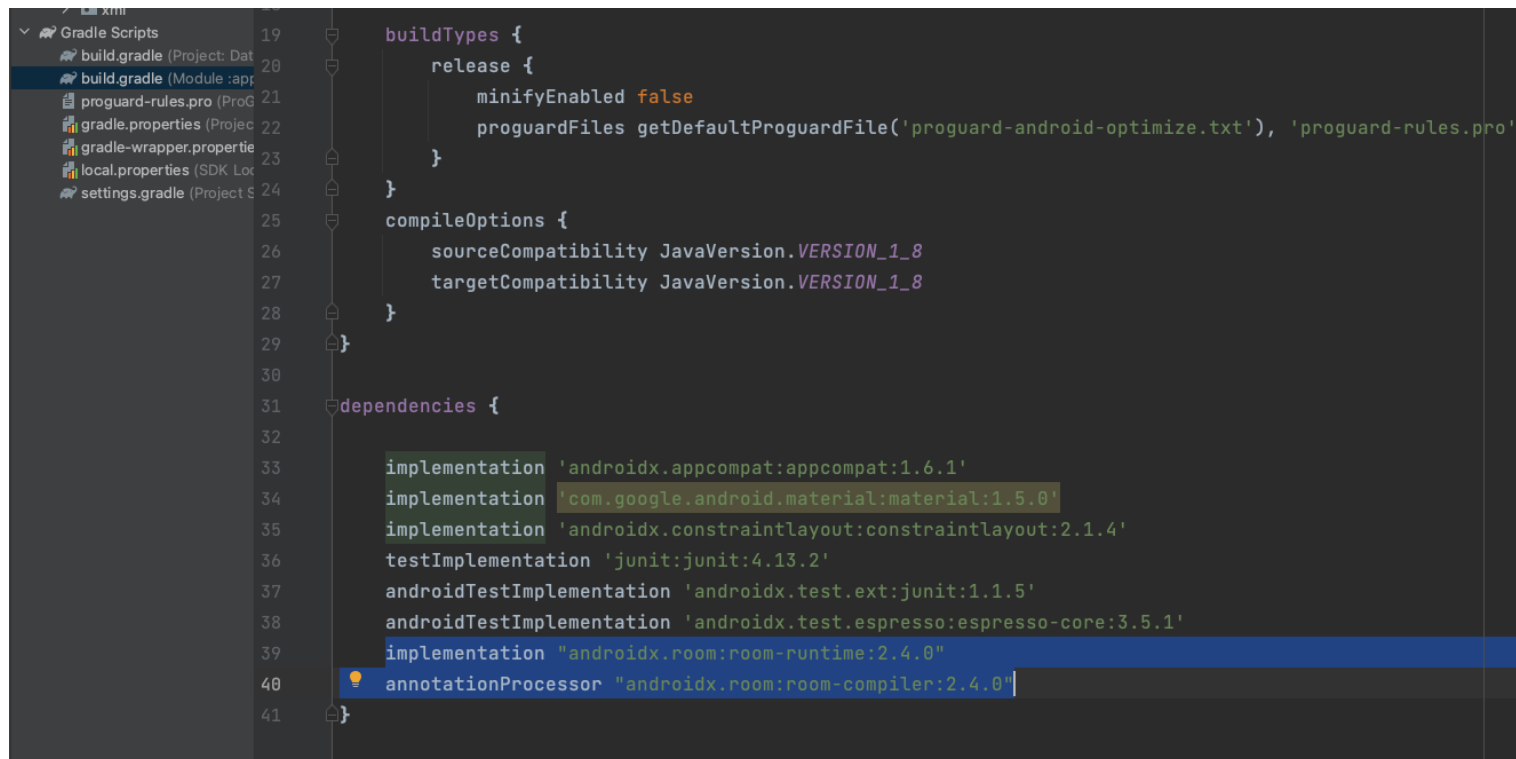
- SQLite
  - SQLite provides a direct API for working with databases. You interact with it using SQL statements and manage database creation, tables, indexes, and queries manually.
  - You write raw SQL queries to interact with the database.
- Android Room (Recommended):
  - Android Room is a higher-level abstraction built on top of SQLite. It provides a simplified and more convenient way to work with databases.
  - Room uses annotations to define entities (tables) and DAOs (Data Access Objects). It generates code for database operations at compile time.
  - Room generates type-safe queries at compile time using annotations and abstract methods in DAO interfaces.

# What will we learn?

- Android Room
  - Because Google recommends to use Room
- I have attached a source code and old lecture slide on how to use the android.database and use raw sql queries. If you are interested, please check.

# Android Room – step by step

- Step 1: Add Dependencies
  - Add the necessary dependencies to your app's build.gradle file.
  - Press Sync now to sync the project after gradle change.



```
19 buildTypes {
20     release {
21         minifyEnabled false
22         proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
23     }
24 }
25 compileOptions {
26     sourceCompatibility JavaVersion.VERSION_1_8
27     targetCompatibility JavaVersion.VERSION_1_8
28 }
29 }
30
31 dependencies {
32     implementation 'androidx.appcompat:appcompat:1.6.1'
33     implementation 'com.google.android.material:material:1.5.0'
34     implementation 'androidx.constraintlayout:constraintlayout:2.1.4'
35     testImplementation 'junit:junit:4.13.2'
36     androidTestImplementation 'androidx.test.ext:junit:1.1.5'
37     androidTestImplementation 'androidx.test.espresso:espresso-core:3.5.1'
38     implementation 'androidx.room:room-runtime:2.4.0'
39     annotationProcessor 'androidx.room:room-compiler:2.4.0'
40 }
41 }
```



# Step 2: Define your Entity

- An entity is a class that represents a table in SQLite database.
- Each instance of the entity class corresponds to a row in the table, and the fields (properties) of the entity class represent the columns in the table.
- You define an entity using the **@Entity** annotation. This annotation tells Room that the class should be treated as a table in the database.
- You use the **@PrimaryKey** annotation to specify the primary key of the entity. The primary key uniquely identifies each row in the table.
- The fields of the entity class represent the columns in the table. You can annotate these fields with annotations like **@ColumnInfo** to specify column properties.

## Entity Class - example

```
import androidx.room.ColumnInfo;
import androidx.room.Entity;
import androidx.room.PrimaryKey;
```

```
@Entity(tableName = "students")
public class Student {
    @PrimaryKey(autoGenerate = true)
    private long id;

    @ColumnInfo(name = "student_name")
    private String name;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

# More about Entity annotations

- Auto-Generate Primary Key: You can use the **@PrimaryKey(autoGenerate = true)** annotation to create an auto-incrementing primary key.
- Composite Primary Key: If you need instances of an entity to be uniquely identified by a combination of multiple columns, you can define a composite primary key by listing those columns in the **primaryKey** property of @Entity:

```
@Entity(primaryKey = {"firstName", "lastName"})
public class User {
    public String firstName;
    public String lastName;
}
```

# More about Entity annotations

- By default, Room creates a column for each field that's defined in the entity. If an entity has fields that you don't want to persist, you can annotate them using **@Ignore**:

```
@Entity
public class User {
    @PrimaryKey
    public int id;

    public String firstName;
    public String lastName;

    @Ignore
    Bitmap picture;
}
```

- **Indices:** You can use the @Index annotation to create indices on columns to improve query performance.

Read more here: <https://developer.android.com/training/data-storage/room/defining-data>

## Step 3: Create your Data Access Object (DAO)

- Create an interface that defines the methods for accessing the database.
- DAO (Data Access Object) is an interface that defines methods for accessing the database.
- DAOs provide an abstraction layer that allows you to perform database operations without writing raw SQL queries. Instead, you define methods in the DAO interface, and Room generates the necessary SQL code for you at compile time.

# DAO annotations

- You define a DAO using the `@Dao` annotation. This annotation tells Room that the interface should be treated as a DAO.
- Each method in a DAO interface corresponds to a specific database operation, such as inserting, updating, deleting, or querying data.
- You use annotations like `@Insert`, `@Update`, `@Delete`, and `@Query` on DAO methods to specify the type of operation.

```
import androidx.room.Dao;

@Dao
public interface StudentDAO {
}
```

# DAO - Insert

- You can use the @Insert annotation to define methods that insert multiple entities into the database in a single operation.

```
@Dao
public interface StudentDAO {
    @Insert
    void insert(Student student);
    @Insert
    void insertMultiple(List<Student> students);
}
```

- The insert method is used to insert a single Student entity into the database.
- The insertMultiple method is used to insert a list of Student entities into the database. Note that the method parameter is of type List<Student>.

# DAO - Update

- You can update multiple entities in the database using the @Update annotation.

```
@Dao
public interface StudentDao {
    @Update
    void update(Student student);

    @Update
    void updateMultiple(List<Student> students);
}
```

- The update happens based on the primary key, i.e., id in student entity



# DAO - Delete

- @Delete annotation to define methods that delete entities from the database.

```
@Dao
public interface StudentDao {
    @Delete
    void delete(Student student);

    @Delete
    void deleteMultiple(List<Student> students);
}
```

- The delete happens based on the primary key, i.e., id in student entity

# Using Varargs - Variable-length argument lists

- The three dots (...) in the context of a method parameter is known as the "varargs" syntax
- It allows you to pass a variable number of arguments of the same type to a method.

```
@Dao
public interface StudentDAO {
    @Insert
    void insert(Student... student);
    @Update
    void update(Student... student);
    @Delete
    void delete(Student... student);
}
```

- The Student... students parameter indicates that you can pass any number of Student objects separated by commas or in an array. Internally, these objects are treated as an array of Student instances.

# Varargs example

- You can directly pass multiple arguments of the same type separated by commas.

```
void myMethod(int... numbers) {  
    // ...  
}  
  
myMethod(1, 2, 3); // Passing three integers
```

- You can also pass an array directly to a varargs parameter.

```
int[] numArray = {4, 5, 6};  
myMethod(numArray); // Passing an array
```

# DAO - Query

- **Basic Query:** You can define a simple query to retrieve data from the database using the `@Query` annotation. For example, let's say you want to retrieve all students from a "students" table:

```
@Dao
public interface StudentDao {
    @Query("SELECT * FROM students")
    List<Student> getAllStudents();
}
```

# DAO- Query with parameters

- You can pass parameters to your query by using placeholders in the query string and adding corresponding method parameters with the @Query annotation.

```
@Dao
public interface StudentDao {
    @Query("SELECT * FROM students WHERE name = :studentName")
    List<Student> getStudentsByName(String studentName);
}
```

```
@Dao
public interface StudentDao {
    @Query("SELECT * FROM students WHERE id = :studentId")
    Student getStudentById(long studentId);
}
```

# DAO Query with Multiple parameters

- You can also pass multiple parameters or reference the same parameter multiple times in a query.

```
@Query("SELECT * FROM user WHERE age BETWEEN :minAge AND :maxAge")
public User[] loadAllUsersBetweenAges(int minAge, int maxAge);

@Query("SELECT * FROM user WHERE first_name LIKE :search " +
        "OR last_name LIKE :search")
public List<User> findUserWithName(String search);
```

# DAO Query – Custom Object mapping

- You can map query results to custom objects using projections. This allows you to retrieve specific columns and construct custom objects from them.

```
public class NameTuple {  
    @ColumnInfo(name = "first_name")  
    public String firstName;  
  
    @ColumnInfo(name = "last_name")  
    @NonNull  
    public String lastName;  
}
```

```
@Query("SELECT first_name, last_name FROM user")  
public List<NameTuple> loadFullName();
```

- Read more: <https://developer.android.com/training/data-storage/room/accessing-data>

# The final StudentDao interface

```
@Dao
public interface StudentDAO {

    @Insert
    void insert(Student... student);

    @Update
    void update(Student... student);

    @Delete
    void delete(Student... student);

    @Query("SELECT * FROM students")
    List<Student> getAllStudents();

    @Query("SELECT * FROM students WHERE student_name = :studentName")
    List<Student> getStudentsByName(String studentName);

    @Query("SELECT * FROM students WHERE id = :studentId")
    Student getStudentByID(int studentId);
}
```



## Step 4: Create your database class

- You need to define class to hold the database. The database class must satisfy the following conditions:
  - The class must be annotated with a `@Database` annotation that includes an entities array that lists all of the data entities associated with the database.
  - The class must be an abstract class that extends `RoomDatabase`.
  - For each DAO class that is associated with the database, the database class must define an abstract method that has zero arguments and returns an instance of the DAO class.

```
import androidx.room.Database;
import androidx.room.RoomDatabase;

@Database(entities = {Student.class}, version = 1)
public abstract class StudentDataBase extends RoomDatabase {
    public abstract StudentDAO studentDAO();
}
```

# Step 5: Initialize Your Database class

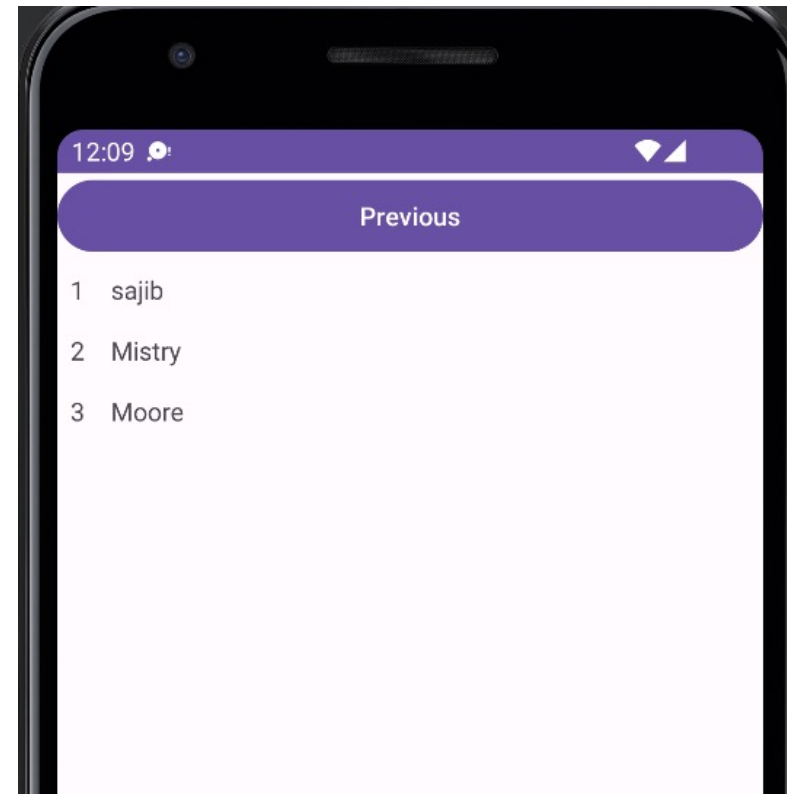
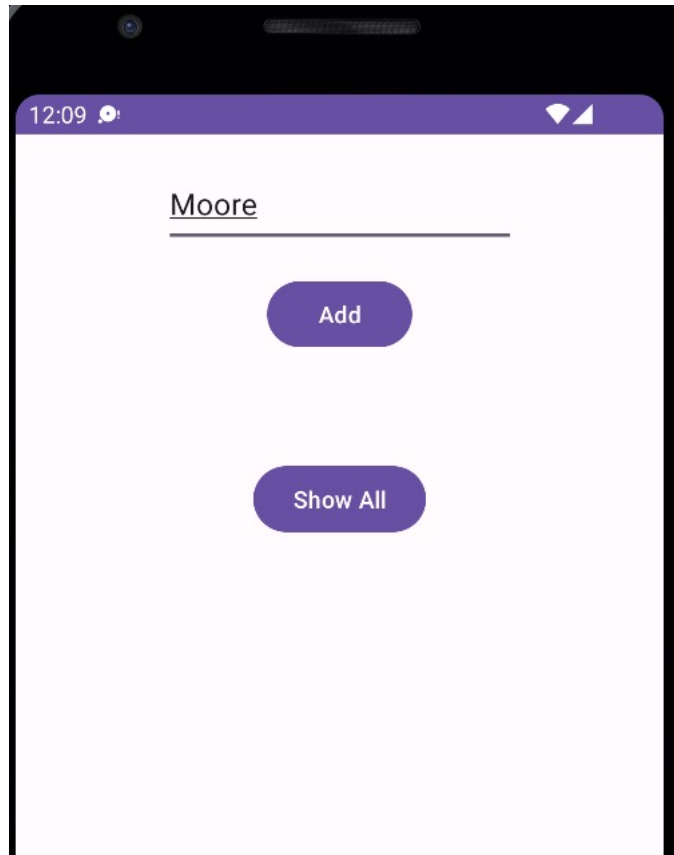
- Create a class to initialise your Database class object using Singleton pattern

```
2 usages
public class StudentDBInstance{
    3 usages
    private static StudentDataBase database;

    2 usages
    public static StudentDataBase getDatabase(Context context) {
        if (database == null) {
            database = Room.databaseBuilder(context,
                StudentDataBase.class, name: "app_database")
                .allowMainThreadQueries()
                .build();
        }
        return database;
    }
}
```

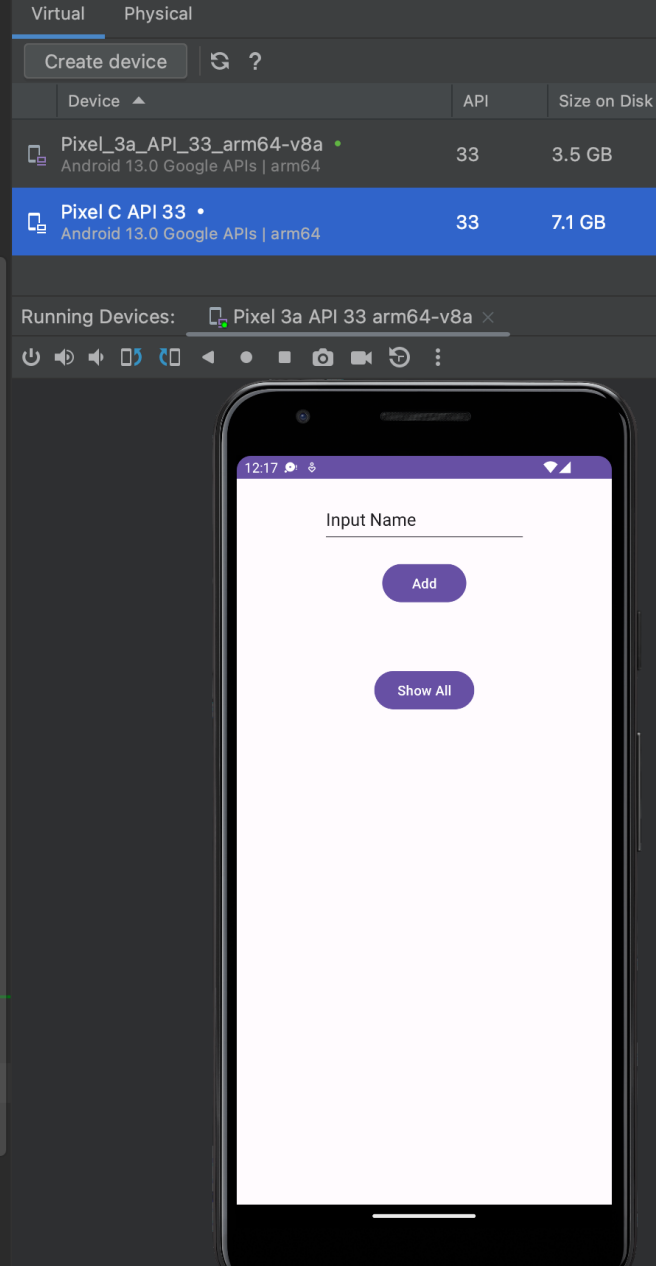
- Note: We are using `allowMainThreadQueries()`. It is not recommended. We will omit it when we learn multi-threading in Android in later lectures.

# Step 6: Use Room in Your Activities/Fragments



# MainActivity.java

```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
  
    StudentDAO studentDAO = StudentDBInstance.  
        getDatabase(getApplicationContext()).studentDAO();  
    EditText name = findViewById(R.id.studentName);  
    Button add = findViewById(R.id.addButton);  
    Button next = findViewById(R.id.nextActivity);  
  
    add.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            String studentName = name.getText().toString();  
            Student student = new Student();  
            student.setName(studentName);  
            studentDAO.insert(student);  
            Toast toast = Toast.makeText(context: MainActivity.this,  
                text: "A student is added", Toast.LENGTH_SHORT);  
            toast.show();  
        }  
    });  
  
    next.setOnClickListener(new View.OnClickListener() {  
        @Override  
        public void onClick(View view) {  
            Intent intent = new Intent(packageContext: MainActivity.this,  
                NextActivity.class);  
            startActivity(intent);  
        }  
    });  
}
```

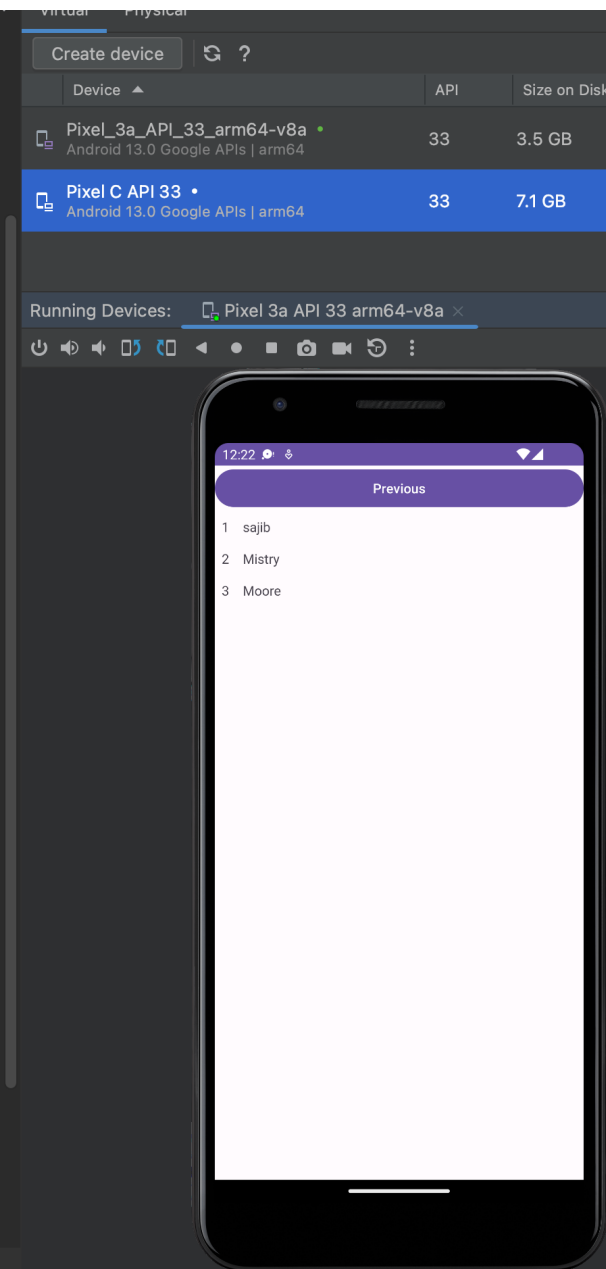


# NextActivity.java

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_next);
    StudentDAO studentDAO = StudentDBInstance.
        getDatabase(getApplicationContext()).studentDAO();
    LinearLayout rootLinearLayout = findViewById(R.id.rootLayout);
    List<Student> studentList = studentDAO.getAllStudents();
    for (Student student: studentList) {
        LinearLayout newLinearLayout = new LinearLayout(context: this);
        newLinearLayout.setOrientation(LinearLayout.HORIZONTAL);
        LinearLayout.LayoutParams params = new LinearLayout.
            LayoutParams(LinearLayout.LayoutParams.WRAP_CONTENT,
                LinearLayout.LayoutParams.WRAP_CONTENT);
        params.setMargins(left: 20, top: 20, right: 20, bottom: 20);

        TextView textView = new TextView(context: this);
        textView.setLayoutParams(params);
        textView.setText(String.valueOf(student.getId()));
        newLinearLayout.addView(textView);
        TextView textView1 = new TextView(context: this);
        textView1.setLayoutParams(params);
        textView1.setText(student.getName());
        newLinearLayout.addView(textView1);

        rootLinearLayout.addView(newLinearLayout);
    }
    Button prev = findViewById(R.id.prev);
    prev.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Intent i = new Intent(packageContext: NextActivity.this,
                MainActivity.class);
            startActivity(i);
        }
    });
}
```



# How to inspect your data

## Use App Inspection in Android Studio

The screenshot displays the App Inspection interface within Android Studio. The top bar shows the device configuration: Pixel 3a API 33 arm64-v8a > com.example.databaseroom. Below this, the 'Database Inspector' tab is active, showing a tree view of databases. The 'app\_database' is expanded, and the 'students' table is selected. The table data is displayed in a grid with columns 'id' and 'student\_name'. The data rows are:

| id | student_name |
|----|--------------|
| 1  | sajib        |
| 2  | Mistry       |
| 3  | Moore        |

The bottom of the screen shows the Android Studio toolbar with various icons for Version Control, Run, TODO, Problems, Terminal, App Inspection (active), Logcat, App Quality Insights, Services, Build, and Profiler.

# How to Wipe Data

- If you need to remove the database from your application, we may need to WIPE data from the emulator.

