Mobile Application Development (Finals) - Jaish Khan

1. Android and Android Studio

> **Android** → An open-source operating system, made by Google for mobile devices like smartphones or tablets.

IDE (Integrated Development Environment) → A software that provides the environment for writing, testing and debugging code.

- An IDE has features like a Visual Layout Editor, Code Autocompletion, Refactoring Tools, App Previewing and Debugging Tools.

> ⓘ **Android Studio**
>
> The official IDE for Native Android Development built on top of IntelliJ IDEA.

APK (Android Package Kit) → Android's native file format for packaging and distributing applications. It's essentially a compressed archive that contains all the necessary components of an Android application. *It contains Compiled code, Resources/Assets and Manifest file.*

1.1. Basics of Android

We can create Android Apps using these 2 languages together.

> **XML** → The markup language used to define structure/layout.
> **Java** → An object-oriented programming language used to define behaviour.

Thus, there are 2 main files in every android studio project: `MainActivity.java` and `activity_main.xml`.

1.1.1. App Mode

Apps that we create are by default in "Development Mode".
In software development, **Development Mode** and **Production Mode** serve distinct purposes. Development Mode is tailored for the development and debugging process, providing detailed error messages, stack traces, and debug logging. It enables developer options, activates performance monitoring tools, and supports hot reloading, allowing rapid iteration during development. However, this mode typically results in slower performance due to the overhead introduced by debugging features.

In contrast, **Production Mode** is optimized for end-user deployment. It focuses on efficient code execution, with minimal logging and disabled developer features to ensure better security and resource usage. Production Mode also reduces the application size, delivering enhanced performance and a streamlined user experience.

1. **Production Mode** → The state of an Android application when it is deployed and released to end-users.
2. **Development Mode** → The state during the app's development and testing phase. Debugging and Logging features are used.

1.1.2. Static or Dynamic Android Apps.

Static Apps	Dynamic Apps
Content Packaged with installation	Content Updated from servers
Limited or no server interaction	Regular data synchronization
Predictable behavior	Real-time updates
Offline functionality	Backend integration
Not user-generated	User-generated content
Lower maintenance requirements	Higher flexibility

- **Instance Variables** → Variables that are declared within a class and outside a function.
- **Callback Function** → A function that can be passed as an argument to another function.

1.1.3. Stages of Mobile App Development

1. Planning
2. Design
3. Development
4. Testing
5. Deployment
6. Maintenance

1.2. Attributes

1.2.1 Basic Attributes

1. `id` → A unique name given to an element, to be able to reference it (in the java file).
    - The `findViewById(R.id.name)` method is used to specify the id in Java.

2. `text` → Visible text inside of **TextView**, **Button** etc. elements.
    - `textSize` → For font size (preferably in `sp`).
    - `textColor` → For font color (using a Hex value or Color resource).
3. `background` → Sets the background color or drawable.
4. `hint` → Placeholder text inside **EditText** elements.
5. `inputType` → Specifies the type of the input (like textPassword, number etc).
6. `src` → To define the source image of an ImageView.
7. `visibility` → Can be visible, invisible or gone.

Sizing → `layout_width` and `layout_height` are used to define the width and height of the element itself. They can have these values:

1. A specific size (preferably in `dp`) like 480dp, 700dp etc.
2. `match_parent` → This element now takes up all the available space in its parent.
3. `wrap_content` → This element only takes up as much space, as is required by its content.

Units → Android Studio has these two unique units:

1. `sp` → scale-independent pixel: which is preferred for fonts.
2. `dp` → density-independent pixel: which is used for everything else (margins/paddings).

1.2 Alignment Attributes

We can use these attributes in XML to align elements.

1. `gravity` → To align the content inside this element.
2. `layout_gravity` → To align this element inside its parent.
3. `padding` → To define the space between this element's boundary and content.
4. `layout_margin` → To define the space outside this element's boundary.
5. `scaleType` → Used to scale images to the bounds of an ImageView.

1.3. Logging and Debugging

> **Logging** allows developers to record information about the application's execution including errors and warnings. **Debugging** is process of finding and fixing errors.

Logcat is a tool inside of Android Studio that displays the system's as well as the application's log messages. It is a valuable tool for debugging.

Compile-Time Errors	Run-Time Errors
An error that occurs during the compilation of an Android application.	An error that occurs during the execution of an Android application.

Compile-Time Errors	Run-Time Errors
Detected by the compiler before the application execution.	Detected by the runtime environment while the app is running.

> **How to solve or see runtime errors?**
>
> Developers can use debugging tools (logcat, debugger etc) to solve runtime errors in Android Studio.
>
> - Check "Logcat" output for error messages.
> - Analyze "Stack Trace" to pinpoint error locations.
> - Use "Debugger" to see variable states.

- **Exceptions** → Unexpected events or errors that occur during execution.
- **Try-Catch** → Keywords used to handle exceptions. The code inside of the `try` block is executed and if any exceptions are generated then the code inside of the `catch` block is executed.

1.4. Assets

Drawable Folder → Contains all the assets that we use in our Android project.

Android allows us to have many different types of assets (static resources) in our app.

1. **Cliparts** → Premade vector images provided by Android.
2. **XML Shapes** → We can also draw our own vector images using XML.
3. **Gradients** → Smooth color transitions which can be defined as XML.
4. **Images** → Can be `.png`, `.jpg` or `.webp`.
5. **HTML Files** → To view them using WebView.

We can attach resources (placed in the `res` folder's subfolders) by using `@folder/resource_file_name` inside of attributes like `src` or `color`.

```
//setImageResource() Sets a drawable resource as the content of the
ImageView.
ImageView imageView = findViewById(R.id.imageView);
imageView.setImageResource(R.drawable.your_image);
// When you have an image in your res/drawable folder and you want to set
it directly by its resource ID.

//setImageDrawable() Sets a drawable object as the content of the
ImageView.
ImageView imageView = findViewById(R.id.imageView);
Drawable drawable = getResources().getDrawable(R.drawable.your_image,
```

```
getApplicationContext().getTheme());
imageView.setImageDrawable(drawable);
// When you need more flexibility, such as setting a drawable that you've
created or modified programmatically.
```

1.5. App Previewing

There are two ways of previewing an app that we have built in Android Studio:

1. **AVD** (Android Virtual Device) → Emulates an android device, allowing developers to run and test their app inside of Android Studio.
2. **ADB** (Android Debug Bridge) → Enables installing the app onto your own Android device to be able to test them. (requires enabling of Developer Options → USB Debugging).

> AVD is advantageous compared to ADB because it allows us to test for multiple different devices and screen sizes (on the same PC).

1.6. Database

> An organized collection of data.

Types of Databases

1. *SQL* (Relational) Databases ⟶ Stores data in the form of tables with rows/columns.
   - Examples: SQLite, MySQL, PostgresSQL, Oracle.
2. *NoSQL* Databases ⟶ Stores data in the form of key-value pairs.
   - Examples: Firebase, MongoDB, Redis.

---

2. Views and ViewGroups

2.1. Views

> The basic building block of Android user interfaces. It is a simple rectangular UI element (also called **Widget**) that displays something to the user (like text, image, etc.) or responds to user interaction (like buttons, text fields, etc.) and used to build the user interface of the app.

They have equivalent classes, **with the same name**, in Java which allows them to be imported into the Java file.

Widget	Used for
TextView	Read-only text
EditText	Editable text field
Button	Clickable UI field
WebView	Displaying web content
ImageView	Displaying images

A `Toast` element, which is not a widget for "building the UI", is used for displaying temporary and breif messages on the screen.

Important Points

1. When using `WebView`
    1. We have to enable the internet permission in the `AndroidManifest.xml` file.
    2. We have to enable javascript using `setJavaScriptEnabled(True)` method.

Click Event Handling

There are two ways of handling click events in Android Java (usually on Buttons).

1. Using the `android:onClick="function"` attribute to specify a function in XML.
2. Using `setOnClickEventListener(view → function)` method to attach an event listener in Java.

You should never use the *first way* due to many reasons. Almost always use the **second way**.

Attaching Resources

Different widgets like `ImageView` and `WebView` require some resources to be specified. We can do that by placing our resources in the **drawable** folder (inside the *res* folder).

> We can then use methods like `setImageResource(R.drawable.image)` and `loadUrl("https://www.example.com")` to specify the resource for our ImageView and WebView elements, respectively.

2.1.1 CardView

> Provides a flexible and easy way to create a card-like layout. It is part of the Android Support Library and is used to display information in a card format with rounded corners and a shadow effect, giving a material design look and feel.

We can use these attributes:

- `cardCornerRadius`: Sets the radius of the card's corners.
- `cardElevation`: Sets the elevation (shadow) of the card.
- `cardBackgroundColor`: Sets the background color of the card.

2.2. ViewGroups

> An invisible container that defines the layout structure of the Activity and can hold multiple Views. They are also called *Layouts* (they're not the same).

> **Layouts** → subclass of ViewGroups can define how children are arranged on the screen and the structure of the UI.

In Android there are many different ViewGroups that can be used inside a Layout file (`activity_main.xml`):

1. **LinearLayout** → Arranges children in a single row (horizontal) or column (vertical).
2. **RelativeLayout** → Positions children relative to the each other or parent container.
3. **ConstraintLayout** → *The most flexible viewgroup*. Allows placing children relative to each other or parent using constraints.
4. **TableLayout** → Organizes children in rows and columns using `TableRows`.
5. **GridView** → Places children in a grid-like structure of rows and columns.
6. **FrameLayout** → Displays a single view or multiple views layered on top of each other.
7. **ScrollView** → A special type of FrameLayout as it only accepts a single view. We usually use a LinearLayout for this and it then allows scrolling through the children of the LinearLayout.
8. **HorizontalScrollView** → Same as ScrollView but it allows scrolling horizontally.

2.2.1. RecyclerView

> A flexible and efficient view group in Android for displaying large sets of data. It's an advanced version of the ListView and GridView but with more features and better performance.

- Allows for efficient "reusing of views" which improves performance and reduces memory consumption, especially for long lists.

Unlike ListView, RecyclerView requires an **Adapter** and a **Layout Manager** to handle data binding and layout arrangement, making it more flexible.

Adapter → Responsible for providing data to the RecyclerView and creating ViewHolder objects that represent individual items in the list. The adapter takes a collection of data (such as a list or array) and binds it to views in the RecyclerView (like individual rows or grid cells).

Importance of Adapter in RecyclerView

1. *Data Binding* → The key part that binds the data to the views inside the RecyclerView. Without the adapter, the RecyclerView would have no way of knowing what data to display.
2. *View Recycling and Performance* → Allows for efficient recycling of views, which helps optimize memory and performance for long lists.
3. *Modular and Maintainable Code* → Helps separate concerns, allowing data handling to be isolated from the layout and UI. This makes the code more modular and maintainable.

---

3. Screens

3.1. Activities

An **activity** is a "screen" for the user. Each activity has a XML layout file and a Java implementation file. We can also create new activities in Android and every activity we create generates two more files for it: one for Java and one for XML.

To create a New Activity,

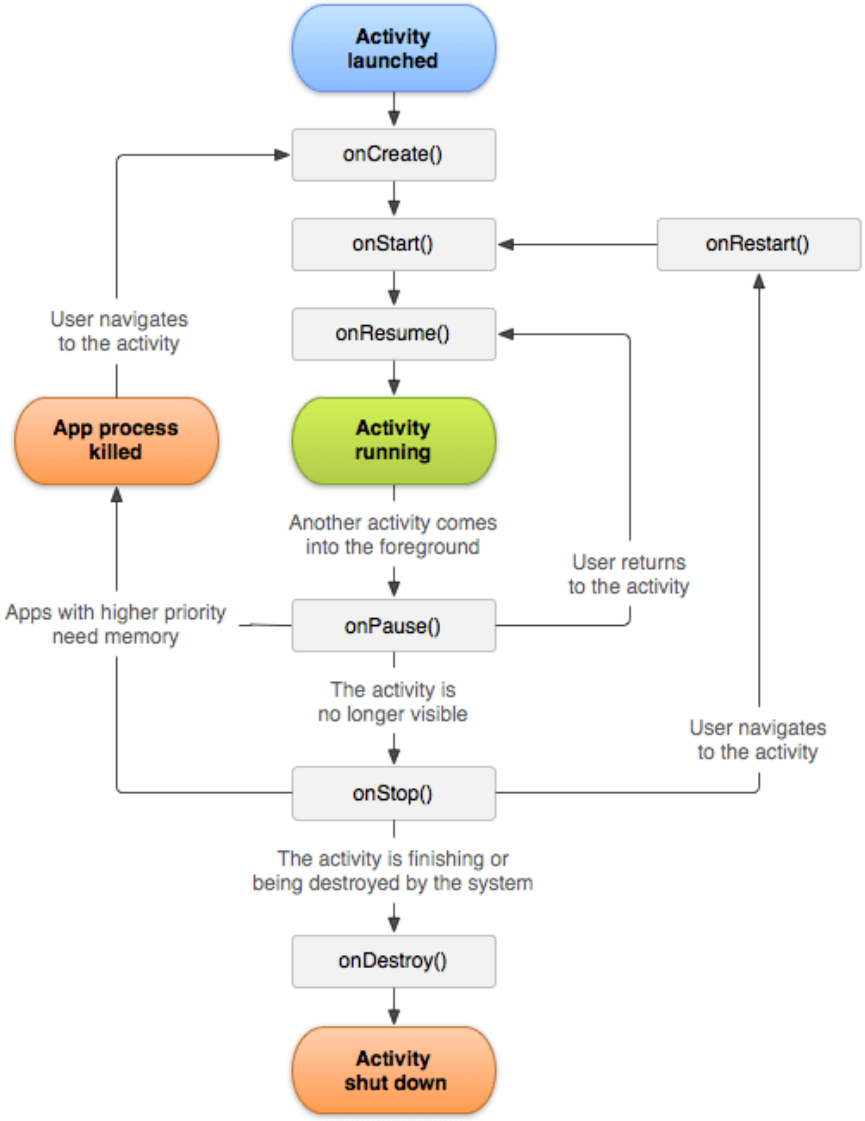
- Right-Click on the App folder.
- Select **Activity** > **New Activity** > **Empty Views Activity**.

3.1.1. Activity Life Cycle

Every activity has these 7 functions which make up the "Activity Life Cycle".

- The `super` keyword is used ensure the parent class executes its necessary setup code before your activity runs its own code.

Method	Runs When	Super Function
onCreate	activity is first created	`super.onCreate();`
onStart	activity is becoming visible to the user	`super.onStart();`
onResume	activity will start interacting with the user	`super.onResume();`
onPause	activity is not visible to the user.	`super.onPause();`
onStop	activity is no longer visible to the user.	`super.onStop();`
onRestart	after your activity is stopped, prior to start.	`super.onRestart();`
onDestroy	before the activity is destroyed.	`super.onDestroy();`



3.2. Intents

A fundamental concept of Android. They are basically like *messages* and are used for **navigation** and **message-passing**.

> The general syntax is to create an object of the `Intent` class and give the source activity and destination activity name to it. Then use the `startActivity()` method to execute the intent.

There are two main types of intents: Explicit and Implicit.

3.2.1. Explicit Intent

Where it is specified which component of which application will answer the intent. They are used for in-app actions like navigating to another acitivity/screen of the same app or starting a service to download a file in the background.

```
// MainActivity.java
Intent intent = new Intent(MainActivity.this, SecondActivity.class);
```

```
startActivity(intent);
```

3.2.2. Implicit Intent

Where the component isn't specified and instead a general action is declared, allowing other apps to handle the intent. They are used for navigating to other apps or retrieving information from other apps.

```
// For Opening Webpage
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setData(Uri.parse("https://www.example.com"));
startActivity(intent);

// For Sending Email
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, new String[]{"example@example.com"});
intent.putExtra(Intent.EXTRA_SUBJECT, "Subject");
intent.putExtra(Intent.EXTRA_TEXT, "Email message text");
startActivity(Intent.createChooser(intent, "Send Email"));
```

3.3.3. Extras

They are key-value pairs which can be attached using the various `putExtra()` methods and carry additional information to be passed on to another activity or app. We can also create a `Bundle` object to send all the extra data at once.

3.3.4. Actions

Android uses actions to specify the type of task to be performed upon an intent.

1. `ACTION_SEND` → Sending information generally like Text, Images and Files.
2. `ACTION_SENDTO` → Sending information specifically like Emails and Phone Numbers.
3. `ACTION_VIEW` → Used to show something; like Webpages and Images, to the user.
4. Intent, Intent Passing, Types of Intent (Explicit and Implicit Intent)
5. Extras (putExtra(),EXTRA_EMAIL , EXTRA_SUBJECT, EXTRA_TEXT)

3.3. Fragments

39. What is Fragment in Java (Android)?
40. Fragment Manager in Android (Common methods of FragmnentManager)
41. Steps to create fragments how to add and how to change them dynamically
42. Data passing in Fragments

> A modular section of an Activity that allows you to create more flexible UI designs by breaking the Activity layout into smaller components.

A Fragment represents a portion of the UI, and it can be added, removed, or replaced within an Activity while the Activity itself is still running. They can have:

- Their own layout and lifecycle
- Can be added, removed, or replaced dynamically.
- Can interact with other Fragments within the same Activity.

3.3.1. Fragment Manager

> Responsible for managing fragments within an activity. It allows you to perform various fragment operations, such as adding, removing, replacing, or finding fragments. It handles fragment transactions (like replacing or adding fragments) and ensures they are properly maintained within the activity.

Common Methods of FragmentManager

1. `beginTransaction()` → Starts a fragment transaction.
2. `add()` → Adds a fragment to the activity.
3. `replace()` → Replaces the existing fragment with a new one.
4. `remove()` → Removes a fragment from the activity.
5. `findFragmentByTag()` → Finds a fragment by its tag.
6. `commit()` → Commits the transaction and applies the changes.

3.3.2. Fragment Creation and Dynamic Updation.

1. In a New Project, Right-Click the App Folder into **New** > **Fragments** > **Blank Fragment** (you will get the two files for the fragment xml and java).
2. In `activity_main.xml`, Drag-and-Drop FragmentContainerView and give it an id.
3. Assuming that we have 3 buttons (One, Two, Three) in the `activity_main.xml` at the top and then one `FragmentContainerView` after them. Add this code on each of the button separately (using onclicklistener).

```
// Fragment Replacement Code
FragmentManager fragmentManager = getSupportFragmentManager();
fragmentManager.beginTransaction()
 .replace(R.id.fragmentContainerView, ChatsFragment.class, null)
 .commit();
```

---

4. Navigation Drawer

> A UI panel that shows your app's main navigation menu.

The most common feature offered by Android. It provides actions preferable to the users, for example changing the user profile, changing the settings of the application, etc.

4.1. Steps to create Navigation Drawer

1. Convert activity_main.xml to **DrawerLayout** and add `tools:openDrawer="start"`.
2. Right-Click layout folder, New > **Layout Resource File** to Create Header Layout `header_layout.xml` for the upper part of the drawer.
3. Right-Click res folder, **New** > **Android Resource Directory** and *ResourceType* of **Menu** to Create Navigation Menu `menu.xml`.
   - It uses the `item` elements to Create menu items with unique IDs.
4. Create Content Layout (`content_layout.xml`) for the content on the main screen.
   1. `Toolbar` element → To create the top toolbar (ImageButton is inside this).
   2. `ImageButton` element → To create the button (with an image) to open the drawer.
   3. `FragmentContainerView` element → To create a container to switch fragments.
5. Add header and menu as well as include the contentlayout files in activity_main.xml.

```
<androidx.drawerlayout.widget.DrawerLayout>
    <!-- Main Content -->
    <include layout="@layout/content_layout"/>

    <!-- Navigation Drawer -->
    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/header_layout"
        app:menu="@menu/nav_menu"/>
</androidx.drawerlayout.widget.DrawerLayout>
```

6. In the `MainActivity.java` file

```
// 1. Initialization
DrawerLayout drawerLayout = findViewById(R.id.drawer_layout);
NavigationView navigationView = findViewById(R.id.nav_view);
```

```
ImageButton menuButton = findViewById(R.id.menu_button);
menuButton.setOnClickListener(v→
drawerLayout.openDrawer(GravityCompat.START));

// 2. Navigation Handling
navigationView.setNavigationItemSelectedListener(item → {
        switch (item.getItemId()) {
                case R.id.nav_home:
                        switchFragment(new HomeFragment()); break;
                case R.id.nav_profile:
                        switchFragment(new ProfileFragment()); break;
        }
        drawerLayout.closeDrawer(GravityCompat.START);
        return true;
});

// 3. Fragment Switching
private void switchFragment(Fragment fragment) {
getSupportFragmentManager()
        .beginTransaction()
        .replace(R.id.fragment_container, fragment)
        .commit();
}

// 4. Back Button Implementation
public void onBackPressed() {
if (drawerLayout.isDrawerOpen(GravityCompat.START)) {
        drawerLayout.closeDrawer(GravityCompat.START);
} else { super.onBackPressed(); }
```

5. SQLite

> A lightweight, serverless, self-contained database engine, perfect for local storage in Android applications because it doesn't require a server to operate and is integrated into Android.

Benefits of SQLite → Lightweight and Efficient, Data Persistence, Uses SQL.

5.1. DBHelper and Constructor

> The `DBHelper` class is responsible for managing database creation, version management, and providing methods for interacting with the SQLite database.

To use it, Right-Click on the Java folder, Select New > Java Class.

- Name the class `DBHelper` and extend it from `SQLiteOpenHelper` (which is provided by Android to manage database creation and version management).

```
public class DBHelper extends SQLiteOpenHelper { /* Code */ };
```

> **Constructor** → The constructor of DBHelper is where we initialize the database, specifying the name and version. This constructor is called when the app is first launched, or when the database needs to be created or upgraded.

```
// Code for the Constructor of DBHelper
public DBHelper(@Nullable Context context) {
        super(context, DB_NAME, null, DB_VERSION);
}
```

- context → required for database creation and management.
- DB_NAME → Name of your database.
- DB_VERSION → Version of your database.
    - If you need to make changes to the structure of the database (like adding new tables or columns), you'll increment this version number.

5.2. SQLite Functions

1. **Creating the Database** → The onCreate method is called when the database is first created. You will define the structure of the database (i.e., the tables and columns).

```
@Override
public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE " + TABLE_NAME + " ("
```

```
            + COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT, "
            + COLUMN_TITLE + " TEXT, "
            + COLUMN_DESCRIPTION + " TEXT)");
}
```

- `execSQL()` method → Executes SQL statements.
  - In this example code we creating 3 columns id, title, and description.

2. **Upgrading the Database** → The onUpgrade method is called when the database version changes. If you modify the schema of the database (e.g., adding a new column or table), this method will handle the upgrade process.

```
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
        db.execSQL("DROP TABLE IF EXISTS " + TABLE_NAME);
        onCreate(db);
}
```

5.2. SQLite CRUD Code Snippets

> CRUD stands for Create, Read, Update and Delete.

```
// 1. Create Method
public boolean insert_data(String title, String description) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues values = new ContentValues();
        values.put(COLUMN_TITLE, title);
        values.put(COLUMN_DESCRIPTION, description);
        return db.insert(TABLE_NAME, null, values) =/= -1;
}

// 2. Read Method
public Cursor getData() {
        SQLiteDatabase db = this.getReadableDatabase();
        return db.rawQuery("SELECT * FROM " + TABLE_NAME, null);
}

// 3. Update Method
public boolean updateData(int id, String title, String description) {
        SQLiteDatabase db = this.getWritableDatabase();
        ContentValues contentValues = new ContentValues();
        contentValues.put(COLUMN_TITLE, title);
        contentValues.put(COLUMN_DESCRIPTION, description);
        String whereClause = COLUMN_ID + " = ?";
    String[] whereArgs = {String.valueOf(id)};
    int result = db.update(TABLE_NAME, values, whereClause, whereArgs);
    return result > 0; // Returns true if the update was successful
```

```
}

// 4. Delete Method
public boolean deleteData(int id) {
        SQLiteDatabase db = this.getWritableDatabase();
        String whereClause = COLUMN_ID + " = ?";
    String[] whereArgs = {String.valueOf(id)};
        int result = db.delete(TABLE_NAME, whereClause, whereArgs);
        return result > 0; // Returns true if the deletion was successful
}
```

5.3 SQLite CRUD Explanation

> The `getWritableDatabase()` and `getReadableDatabase()` methods are used to get a writable or readable instance of the database, respectively.

We use `ContentValues` class to create an object so we can provide the data to be inserted/updated.

1. The `insert_data()` method inserts new records (rows) into the database.
    - `db.insert()` method → Inserts the data into the table.
    - It returns true (positive value) if insertion is successful, else returns **-1**.
2. The `getData()` method gets all records (rows) from the database.
    - `db.rawQuery()` method → Executes a raw SQL query to retrieve all rows from the table. The method returns a **Cursor** object, which allows you to iterate through the results.
3. The `updateData()` method updates existing records (rows) in the database.
    - `db.update()` method → Updates a record in the database.
        - Takes 4 arguments: TableName, NewValues, WHERE clause and the id value for that row.
    - It returns positive value if the update was successful.
4. The `deleteData()` method deletes specified records (rows) from the database.
    - `db.delete()` method → Deletes a record/row from the database.
        - Takes 3 arguments: TableName, WHERE clause and the id value for that row.
    - It returns positive value if the update was successful.

---

6. Firebase

> Firebase is a Backend-as-a-Service (BaaS) platform provided by Google. It simplifies backend development by offering various ready-made services for app developers, allowing them to focus on building features rather than managing servers.

6.1. Firebase Services

Service	Function
Realtime Database	A cloud-hosted NoSQL database which provides real-time data synchronization to all connected clients.
Authentication	Used to manage users securely. Supports user authentication via email/password, phone numbers, and third-party providers (Google, Facebook, Twitter).
Cloud Firestore	A scalable and flexible NoSQL database for storing and syncing data. Best for more complex querying than the Realtime Database.
Cloud Storage	Handles file uploads and stores large files like images, videos, and audio.
Cloud Messaging	Used for sending push notifications to your app users.
Hosting	Serves static files like HTML, CSS, and JavaScript for web applications.
Crashalytics	Tracks and reports app crashes in real time, helping you debug issues.
Analytics	Offers insights into user behavior, such as screen views, active users, and events.
Functions	Allows you to run server-side code (functions) in response to events.
Remote Config	Enables app customization without requiring a new release.

- The data from the database can be synced at a time across all the clients such as android, web as well as IOS. The data in the database is stored in the JSON format and it updates in real-time with every connected client.

6.2. Firebase Configuration

1. **Visit Firebase Console**: Go to https://console.firebase.google.com/ and Create an account.
2. **Create a New Project**: `New Project → Empty Views Activity` in Android Studio.

3. **Connect App to Firebase**:
    1. In Android Studio, Go to `Tools → Firebase` (in the top bar).
    2. In the Realtime Database section, Click on "Get Started with Realtime Database".
    3. This opens a new screen where you have steps. Click on *Connect to Firebase*.
    4. This will open Firebase Console in your browser. Create the Firebase Project, Give it a name, and Connect it to your mobile app.
4. **Add Realtime Database**:
    1. In Android Studio, Click on *Add the Real Time Database SDK to your app*. Accept the changes to update your Gradle files.
    2. In Firebase Console, Go to Realtime Database and enable "Test Mode" to enable read/write operations.
5. **Add Authentication**
    1. Follow the same steps to add Authentication.
    2. In Firebase Console, Go to Authentication and enable "Email/Password Authentication".

6.3. Using Authentication and Real Time Database

1. **Create 3 Activities** → Login, Signup and Dashboard.
2. **Create Screen Layouts** → Create Login, Sign-Up, and Dashboard screens in XML and assign IDs to all views.
3. **Create a Class to Store Data** → Define a Java class with fields for user data like name, contact or email. Include getter and setter methods and an empty constructor for Firebase compatibility.
    - We have to create an entire object and send that entire object to Firebase, if we want to send multiple data.
4. **Implement Sign-Up Functionality** → In the `SignUpActivity`, use `createUserWithEmailAndPassword()` to create a user.
    - Save user data using a sanitized email.
    - Add a TextView (with Intent) to link back to the Login screen.
5. **Implement Login Functionality** → In the `LoginActivity`, use `signInWithEmailAndPassword()` for authentication.
    - On successful login, navigate to the Dashboard and pass the user's email.
6. **Fetch and Display User Data** → In the `DashboardActivity`, get user details from the Realtime Database using the email node. Add the data in the data model class and display the user's name.

6.4. Firebase Code Snippets

```
// 1. Initialize the Firebase Database
FirebaseDatabase database = FirebaseDatabase.getInstance();

// 2. Create Parent node in database with name of "users"
DatabaseReference users_ref = database.getReference("users");

// 3. Initialize the Firebase Authentication
FirebaseAuth auth = FirebaseAuth.getInstance();

// 4. Email Sanitization
String sanitizedEmail = email.getText().toString().replace(".", "_");
DatabaseReference ref = users_ref.child(sanitizedEmail);

// 5. User Creation
auth.createUserWithEmailAndPassword(email.getText().toString(),
password.getText().toString()).addOnCompleteListener(task → {
        if(task.isSuccessful()){
                ref.setValue(signupData).addOnCompleteListener(dbtask → {
                        /* If-Else and Toasts */
                });
        } else { /* Toast to show exception */ };
});

// 6. User Authentication
auth.signInWithEmailAndPassword(email.getText().toString(),
password.getText().toString()).addOnCompleteListener(task → {
        if(task.isSuccessful()){
                ref.setValue(signupData).addOnCompleteListener(dbtask → {
                        /* If-Else and Toasts */
                });
        } else { /* Toast to show exception */ };
});

// 7. Data Fetching (Not Important)
ref.addListenerForSingleValueEvent(new ValueEventListener() {
        public void onDataChange(@NonNull DataSnapshot snapshot) {
        if (snapshot.exists()) {
                // Convert the snapshot to an object
                DataModel user = snapshot.getValue(DataModel.class);

                /* Show data on the screen if user is not null */
         } else { /* Toast to show exception */ }

        public void onCancelled(@NonNull DatabaseError error) { /* Toast
*/ }
});
```

> Email Sanitization is the process of cleaning and validating email content to ensure it is safe and free from malicious code or harmful content.

- The `createUserWithEmailAndPassword()` and `signInWithEmailAndPassword()` functions are used to "create a new user" and "authenticate user for login", respectively, using email and password. They take two string parameters *email* and *password*.
- The `.addListenerForSingleValueEvent()` is a Firebase event listener that is used to read data from the database (just once).
    - Its `onDataChange()` and `onCancelled()` methods are executed for success and failure, respectively.