1. **Write difference between Table Layout and Frame Layout.**

In Android, **layouts** are used to define the visual structure of UI elements. Two commonly used layouts are **TableLayout** and **FrameLayout**. Both organize child views differently and serve distinct purposes in UI design.

1. TableLayout
Definition:
TableLayout arranges its children into **rows and columns**, similar to an HTML table. Each child element of a TableLayout is a TableRow object that defines a single row containing multiple views (cells).
Characteristics:

- Organizes child views into tabular form.
- Each TableRow can contain multiple cells (views).
- Columns can be stretched, collapsed, or shrunk.
- Useful for data display, form layouts, or grid-like arrangements.

Advantages:

- Provides precise control over alignment in rows and columns.
- Supports android:layout_column and android:layout_span attributes.
- Useful for structured UIs such as login forms or calculators.

Disadvantages:

- Less flexible for overlapping views.
- Performance may degrade with many nested rows.

Example:
```
<TableLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:stretchColumns="1">

   <TableRow>
      <TextView android:text="Username:" />
      <EditText android:hint="Enter username" />
   </TableRow>

   <TableRow>
      <TextView android:text="Password:" />
      <EditText android:hint="Enter password" android:inputType="textPassword"/>
   </TableRow>

</TableLayout>
```

2. FrameLayout
Definition:
FrameLayout is designed to hold a **single child view** (though it can technically contain

multiple). All children are stacked on top of each other, with the most recently added child displayed at the top.

Characteristics:

- Simple layout for placing one view or overlapping multiple views.
- Commonly used as a container for fragments, media, or overlays.

Advantages:

- Efficient and lightweight.
- Ideal for fragment transactions, image overlays, and custom view stacking.

Disadvantages:

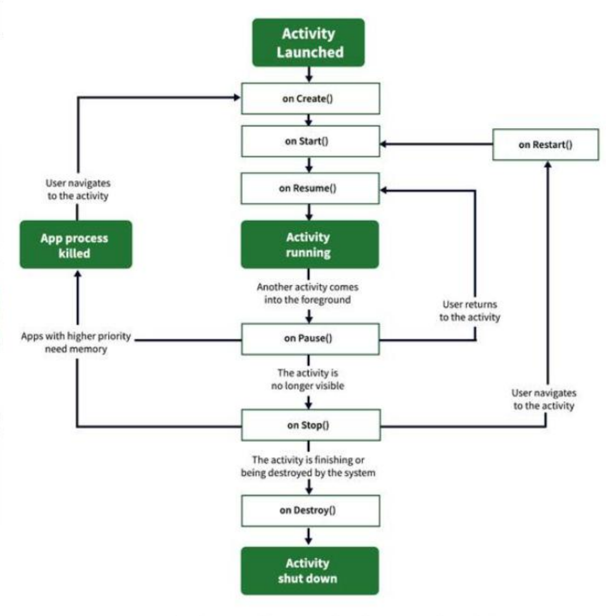
- Limited layout control.
- Not suitable for arranging views in structured rows or columns.

Example:

```
<FrameLayout
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <ImageView android:src="@drawable/background" />
   <TextView android:text="Overlay Text"
          android:layout_gravity="center"
          android:textSize="20sp" />
</FrameLayout>
```

2. **Draw and explain Activity lifecycle of an Android application in detail.**


Activity Launched
on Create()
on Start()
on Restart()
on Resume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
on Pause()
The activity is no longer visible
User navigates to the activity
on Stop()
The activity is finishing or being destroyed by the system
on Destroy()
Activity shut down
Activity Lifecycle in Android

An **Activity** in Android represents a single screen with a user interface. The **Activity Lifecycle** defines the sequence of states and callbacks through which an activity passes, from creation to destruction. Understanding this lifecycle is critical for managing resources, user data, and app stability.

1. Lifecycle Overview
Each activity transitions through a set of predefined methods controlled by the Android OS. The lifecycle ensures that the application behaves predictably during user interactions such as opening, pausing, or closing the app.
2. Core Lifecycle Callbacks

1. **onCreate(Bundle savedInstanceState)**
   - Called when the activity is first created.
   - Used to initialize components, inflate layouts, and bind data.
   - Executed only once in the lifecycle unless the activity is recreated.
   - Example:
   - @Override
   - protected void onCreate(Bundle savedInstanceState) {
   -     super.onCreate(savedInstanceState);
   -     setContentView(R.layout.activity_main);
   - }
2. **onStart()**
   - Invoked when the activity becomes visible to the user.
   - The activity is not yet in the foreground for interaction.
3. **onResume()**
   - Called when the activity starts interacting with the user.
   - The activity is now in the **foreground** and receives user input.
   - Ideal for animations, sensors, or listeners.
4. **onPause()**
   - Triggered when another activity comes partially in front.
   - Used to pause ongoing processes like video playback or location updates.
   - Must be lightweight since the system may kill the activity afterward.
5. **onStop()**
   - Called when the activity is no longer visible.
   - Used for resource release, stopping threads, or saving data.
6. **onRestart()**
   - Invoked after onStop() when the activity is about to restart.
   - Used to reinitialize components paused earlier.
7. **onDestroy()**
   - Called before the activity is destroyed or finished.
   - Used to release all remaining resources and clean up memory.

3. **Explain Content Provider**.
A Content Provider in Android is a component that manages access to a central repository of data, allowing applications to share data with other applications securely and efficiently. It acts as an abstraction layer over the underlying data storage mechanism, which can be a database (like SQLite), files, or even a network.
Here's a breakdown of its key aspects:

- **Data Sharing:**

The primary purpose of a Content Provider is to enable secure data sharing between different applications. For example, the Contacts application uses a Content Provider to expose contact information, allowing other apps like messaging or social media platforms to access it.

- **Standardized Interface:**

Content Providers offer a consistent and standardized interface for accessing and modifying data. This interface is defined by the ContentProvider class and is accessed by other applications through a ContentResolver object.

- **Security and Control:**

Content Providers provide mechanisms for defining data security and controlling access to specific data. This ensures that only authorized applications can access or modify the data managed by the provider.

- **Interprocess Communication (IPC):**

Content Providers facilitate secure interprocess communication, allowing applications running in different processes to exchange data.

- **Content URI:**

Each Content Provider is associated with a unique Content URI, which serves as a pointer to the data it manages. Other applications use these URIs to request specific data from the provider.

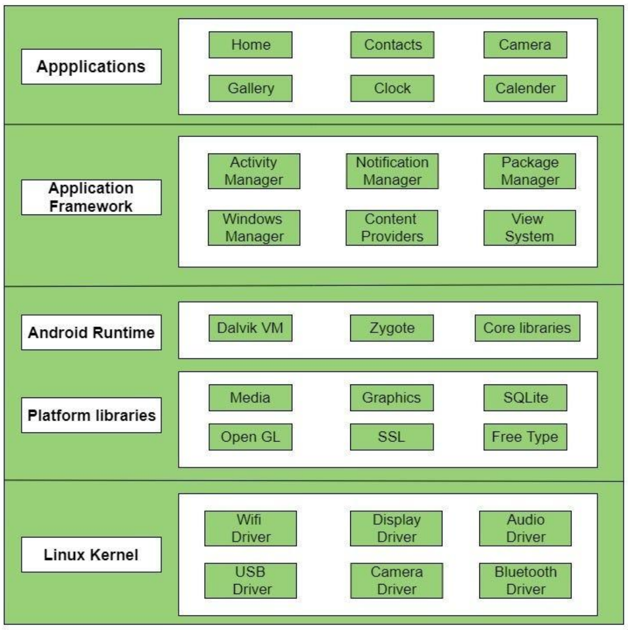
- **Methods for Data Operations:**

Content Providers implement methods like query(), insert(), update(), and delete() to perform standard data operations.

- **Built-in and Custom Providers:**

Android includes several built-in Content Providers (e.g., for contacts, media), and developers can also create custom Content Providers to expose and manage their own application's data.

4. **Write a brief note on Android Architecture with proper diagram.**



Android architecture is a **layered software stack** designed to support mobile devices efficiently. It provides a flexible, modular, and secure framework for building mobile applications. The architecture consists of **four main layers** arranged from the lowest (closest to hardware) to the highest (closest to the user): **Linux Kernel**, **Libraries & Android Runtime**, **Application Framework**, and **Applications**.

1. Linux Kernel (Base Layer)

- The foundation of Android architecture.
- Provides core system services like **memory management**, **process management**, **networking**, and **security**.
- Acts as a hardware abstraction layer between device hardware and software components.
- Includes important drivers such as display, camera, keypad, Wi-Fi, and Bluetooth.

Key Components:

- Display driver
- Camera driver
- Audio driver
- Power management
- Binder (IPC mechanism)

2. Libraries and Android Runtime

a) Native Libraries:

- Written in C/C++, these provide capabilities for core Android features.
- Examples:
    - SQLite (database management)
    - WebKit (browser engine)
    - Media Framework (audio/video playback)
    - OpenGL ES (graphics rendering)

b) Android Runtime (ART):

- Replaced the older Dalvik Virtual Machine.
- Executes applications compiled into bytecode.
- Uses **Ahead-of-Time (AOT)** compilation for faster execution and reduced memory usage.
- Handles memory management and garbage collection.

3. Application Framework

- Provides high-level APIs for application development.
- Manages UI, resources, and application life cycles.
- Enables developers to build feature-rich applications using system-managed services.

Important Managers:

- ActivityManager – Controls activity lifecycle.
- PackageManager – Manages app packages.
- WindowManager – Manages window display.
- ContentProvider – Handles data sharing.
- ResourceManager – Manages app resources (layouts, strings, etc.).

4. Applications (Top Layer)

- The topmost layer where users interact directly.
- Includes **system apps** (Phone, Contacts, Settings) and **user-installed apps**.

- Built using the APIs provided by the Application Framework.

5. Describe Shared Preference with example.

SharedPreferences in Android is a simple data storage mechanism designed to store small, structured pieces of information in the form of **key-value pairs**. It provides a persistent way to save user data and application states across app launches, even after the app is closed or the device restarts. This mechanism is ideal for maintaining settings, preferences, or lightweight user data without needing complex databases.

1. Concept and Purpose

SharedPreferences is part of Android's internal storage system and is primarily used for saving **primitive data types** such as:

- boolean
- int
- float
- long
- String
- Set<String>

It acts as a private XML file maintained by the Android system, allowing developers to write and retrieve values easily. The data is stored within:
/data/data/<package_name>/shared_prefs/<file_name>.xml
Since the storage is private, only the same application can access it by default.

2. Typical Uses

- Storing user preferences such as **theme settings (dark/light mode)**.
- Maintaining session data like **login state** or **username**.
- Remembering **last opened page or app configurations**.
- Enabling **"Remember Me"** options in login screens.

3. Implementation Process

a) Saving Data

```
SharedPreferences prefs = getSharedPreferences("UserPrefs", MODE_PRIVATE);
SharedPreferences.Editor editor = prefs.edit();
editor.putString("username", "JohnDoe");
editor.putBoolean("isLoggedIn", true);
editor.apply();
```

- The apply() method saves data asynchronously, improving performance.
- Data is written into an XML file under the app's internal directory.

b) Retrieving Data

```
SharedPreferences prefs = getSharedPreferences("UserPrefs", MODE_PRIVATE);
String name = prefs.getString("username", "Guest");
boolean loggedIn = prefs.getBoolean("isLoggedIn", false);
```

- The second argument in each method represents the **default value** if no value exists for the key.

c) Deleting or Clearing Data

```
    SharedPreferences.Editor editor = prefs.edit();
    editor.remove("username");  // Remove a specific key
    editor.clear();             // Remove all stored data
    editor.apply();
```

4. Internal Behavior
Internally, SharedPreferences uses XML-based storage and loads data into memory when accessed. It maintains data consistency through thread-safe operations, allowing multiple parts of the app to retrieve or modify preferences simultaneously. SharedPreferences thus ensures efficient and lightweight persistence for essential app data, especially for user settings and configurations.

6. Write the steps for CRUD operation to use Firebase Database in Android application.

Firebase Realtime Database is a cloud-hosted NoSQL database that allows data to be stored and synchronized in real-time across all connected clients. CRUD stands for **Create, Read, Update, and Delete** — the four fundamental operations for managing data. The following are the steps to perform CRUD operations in an Android app using Firebase:

1. Setup Firebase in Android Project

- **Step 1:** Create a new project in Firebase Console.
- **Step 2:** Add your Android app by registering the package name.
- **Step 3:** Download the google-services.json file and place it inside the app/ folder.
- **Step 4:** Add Firebase dependencies in build.gradle:
- implementation 'com.google.firebase:firebase-database:20.3.0'
- implementation 'com.google.firebase:firebase-auth:23.0.0'
- **Step 5:** Sync your project and initialize Firebase in your app.

2. Create (Insert Data)
Use the DatabaseReference class to add data:
```
DatabaseReference dbRef = FirebaseDatabase.getInstance().getReference("Users");
String userId = dbRef.push().getKey();
dbRef.child(userId).setValue(new User("John", "john@gmail.com"));
```
This code stores a new user object in the “Users” node.
3. Read (Retrieve Data)
Read data using ValueEventListener:
```
dbRef.addValueEventListener(new ValueEventListener() {
   public void onDataChange(DataSnapshot snapshot) {
      for (DataSnapshot data : snapshot.getChildren()) {
         User user = data.getValue(User.class);
      }
   }
   public void onCancelled(DatabaseError error) {}
});
```

4. Update Data

To update a specific field:

```
dbRef.child(userId).child("name").setValue("Johnny");
```

5. Delete Data

To delete a record:

```
dbRef.child(userId).removeValue();
```

7. **Write a code to insert product information (pid, pname, pcategory, pprice) in SQLite database.**

Create Database Helper Class

```
public class DBHelper extends SQLiteOpenHelper {
   private static final String DB_NAME = "ProductDB";
   private static final int DB_VERSION = 1;
   private static final String TABLE_NAME = "Products";

   public DBHelper(Context context) {
      super(context, DB_NAME, null, DB_VERSION);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      String query = "CREATE TABLE " + TABLE_NAME +
            "(pid INTEGER PRIMARY KEY AUTOINCREMENT,  " +
            "pname TEXT, " +
            "pcategory TEXT, " +
            "pprice REAL)";
      db.execSQL(query);
   }

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

   public boolean insertProduct(String pname, String pcategory, double pprice) {
      SQLiteDatabase db = this.getWritableDatabase();
      ContentValues cv = new ContentValues();
      cv.put("pname", pname);
      cv.put("pcategory", pcategory);
      cv.put("pprice", pprice);
      long result = db.insert(TABLE_NAME, null, cv);
      db.close();
      return result != -1;
   }
}
```

Insert Product Data from Activity

```
DBHelper dbHelper = new DBHelper(this);
boolean inserted = dbHelper.insertProduct("Laptop", "Electronics", 85000.0);
if (inserted) {
   Log.d("DB_STATUS", "Product inserted successfully");
} else {
   Log.d("DB_STATUS", "Insertion failed");
}
```

Explanation

- SQLiteOpenHelper manages database creation and version control.
- ContentValues maps column names to values for insertion.
- insert() returns the row ID if successful, or -1 if failed.
- The database is closed after operation to prevent memory leaks.

8. **What is Broadcast Receiver in Android? Explain in brief**.

A **Broadcast Receiver** in Android is a key component that allows applications to respond to system-level or application-level broadcast messages. These broadcasts can come from the Android operating system itself or from other applications. The receiver works as a listener that waits for specific intents (messages) and executes defined actions when those events occur. It helps applications stay updated about system changes without remaining active in the foreground, thus improving performance and power efficiency.
There are two main types of broadcasts in Android. **System broadcasts** are automatically sent by the Android system to notify about events such as battery low (ACTION_BATTERY_LOW), network connectivity changes (CONNECTIVITY_CHANGE), or device boot completion (BOOT_COMPLETED). **Custom broadcasts**, on the other hand, are user-defined and are sent from one app component to another using methods like sendBroadcast() or sendOrderedBroadcast(). These are useful for communication within or across applications.
To use a Broadcast Receiver, a developer must first create a class that extends the BroadcastReceiver class and override the onReceive() method. For example:

```
public class MyReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
      Toast.makeText(context, "Airplane Mode Changed", Toast.LENGTH_SHORT).show();
   }
}
```

The receiver must then be registered so that the Android system knows when it should be triggered. Registration can be done in two ways: **statically**, by declaring the receiver in the AndroidManifest.xml file, or **dynamically**, at runtime using registerReceiver() within an activity or service. Static registration is used for events that should be received even when the app is not running, while dynamic registration is used for receivers that operate only when an app component is active.

9. **Explain RecyclerView and CardView in Android with example.**

In Android development, **RecyclerView** and **CardView** are modern UI components used for displaying large sets of data efficiently and attractively. They were introduced to replace the older ListView and GridView by providing better performance, flexibility, and visual appeal.

RecyclerView is a more advanced and flexible version of ListView. It is designed to efficiently reuse item views as the user scrolls through a list. Instead of creating a new view for each data item, RecyclerView "recycles" existing views using a pattern called the **ViewHolder pattern**. This minimizes memory consumption and improves scrolling performance. It works in three main parts:

1. **Adapter** – Binds data to the view items.
2. **ViewHolder** – Holds references to the item views for quick access.
3. **LayoutManager** – Defines how items are arranged (linear, grid, or staggered).

CardView, on the other hand, is a container view that displays content in a card-like layout with rounded corners and elevation (shadow) for a modern Material Design look. It is often used inside RecyclerView to represent each item neatly.

Example Implementation:

```
<!-- item_product.xml -->
<androidx.cardview.widget.CardView
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:layout_margin="8dp"
   android:elevation="4dp"
   android:padding="8dp">

   <TextView
      android:id="@+id/tvProductName"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textSize="18sp"
      android:text="Product Name" />
</androidx.cardview.widget.CardView>
```

In the main layout, include the RecyclerView:

```
<androidx.recyclerview.widget.RecyclerView
   android:id="@+id/recyclerView"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />
```

10. **Explain Location Service in Android with example.**

Location Service in Android is a system feature that allows applications to obtain the geographical location of a device using different providers such as GPS, Wi-Fi, and mobile networks. It enables location-based functionalities like navigation, weather updates, delivery tracking, and nearby place detection. Android provides APIs through the

Fused Location Provider, which combines multiple signals (GPS, network, and sensors) to deliver accurate and power-efficient location data.
The **FusedLocationProviderClient** class, part of Google's Location Services API, is the most efficient way to access location. It automatically chooses the best provider based on accuracy, power, and availability, simplifying the developer's work.
To use location services, permissions must be declared in the **AndroidManifest.xml**:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"
/>
```

Then, in the activity, create a client instance and request location updates:

```
public class MainActivity extends AppCompatActivity {
    FusedLocationProviderClient fusedLocationClient;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);

        if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) ==
PackageManager.PERMISSION_GRANTED) {
            fusedLocationClient.getLastLocation()
                .addOnSuccessListener(this, location -> {
                    if (location != null) {
                        double latitude = location.getLatitude();
                        double longitude = location.getLongitude();
                        Log.d("LOCATION", "Lat: " + latitude + ", Lng: " + longitude);
                    }
                });
        }
    }
}
```

This example retrieves the device's last known location using the Fused Location Provider. Developers can also use **LocationRequest** to receive continuous updates at specific time intervals.

11. Write a code to get current location & display it in TextView.

Code to Get Current Location and Display in TextView (≈300 words)
Android provides the **Fused Location Provider API** for efficiently obtaining the device's location using GPS, Wi-Fi, and mobile networks. This example demonstrates retrieving the current location and displaying it in a TextView.

1. Add Permissions in AndroidManifest.xml:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

---

2. Layout (activity_main.xml)

```
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:padding="16dp">

   <TextView
      android:id="@+id/tvLocation"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Current Location"
      android:textSize="18sp"/>
</LinearLayout>
```

3. MainActivity.java

```
public class MainActivity extends AppCompatActivity {
   private FusedLocationProviderClient fusedLocationClient;
   private TextView tvLocation;
   private final int LOCATION_REQUEST_CODE = 100;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      tvLocation = findViewById(R.id.tvLocation);
      fusedLocationClient = LocationServices.getFusedLocationProviderClient(this);

      if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION) !=
PackageManager.PERMISSION_GRANTED) {
         ActivityCompat.requestPermissions(this, new
String[]{Manifest.permission.ACCESS_FINE_LOCATION},
LOCATION_REQUEST_CODE);
      } else {
         getCurrentLocation();
      }
   }

   private void getCurrentLocation() {
      fusedLocationClient.getLastLocation()
```

```
            .addOnSuccessListener(this, location -> {
                if (location != null) {
                    double latitude = location.getLatitude();
                    double longitude = location.getLongitude();
                    tvLocation.setText("Latitude: " + latitude + "\nLongitude: " + longitude);
                } else {
                    tvLocation.setText("Unable to get location.");
                }
            });
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
@NonNull int[] grantResults) {
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
        if (requestCode == LOCATION_REQUEST_CODE  && grantResults.length > 0 &&
grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            getCurrentLocation();
        }
    }
}
```

Explanation:

- FusedLocationProviderClient handles location retrieval efficiently.
- Permissions are required at runtime for devices running Android 6.0 and above.
- getLastLocation() returns the most recent known location.
- The TextView updates with latitude and longitude values.
- If location is unavailable, a fallback message is displayed.

12. Describe Shared Preference with example.

Shared Preferences in Android is a built-in framework that allows an application to store small amounts of data in the form of **key-value pairs**. It is commonly used to save user-specific settings, preferences, or state information that should persist between app launches. The data stored in Shared Preferences remains available even after the app is closed or the device is restarted, making it suitable for lightweight and persistent storage.

Shared Preferences operates on the principle of **key-value mapping**, where each key uniquely identifies a stored value. It supports only primitive data types such as boolean, float, int, long, String, and Set<String>. The data is stored in an XML file located inside the app's internal storage directory:

/data/data/<package_name>/shared_prefs/<file_name>.xml

Use Cases

- Saving user login status or credentials.
- Remembering user-selected settings like theme or notification preferences.
- Storing app configuration details or last visited page.

Example Implementation

1. Writing Data to SharedPreferences

```
SharedPreferences sharedPref = getSharedPreferences("UserPrefs", MODE_PRIVATE);
SharedPreferences.Editor editor = sharedPref.edit();

editor.putString("username", "JohnDoe");
editor.putBoolean("isLoggedIn", true);
editor.putInt("userScore", 90);
editor.apply();   // Saves data asynchronously
```

The apply() method commits changes in the background, while commit() saves synchronously and returns a success status.

2. Reading Data from SharedPreferences

```
SharedPreferences sharedPref = getSharedPreferences("UserPrefs", MODE_PRIVATE);

String name = sharedPref.getString("username", "Guest");
boolean loggedIn = sharedPref.getBoolean("isLoggedIn", false);
int score = sharedPref.getInt("userScore", 0);
```

Each get method retrieves the stored value, returning the default value if the key does not exist.

3. Removing Data

```
SharedPreferences.Editor editor = sharedPref.edit();
editor.remove("userScore");
editor.apply();
```

Shared Preferences provides a **simple, fast, and persistent** way to manage small datasets within an app. It is best suited for maintaining user configurations and states, ensuring that an application remembers its last-used preferences seamlessly.

13. Explain Intent and Fragment in detail.

An **Intent** and a **Fragment** are two essential components of Android that play distinct roles in application development. Intents enable communication between components, while Fragments allow flexible and modular user interface design within an Activity.

Intent

An **Intent** is a messaging object used to request an action from another app component. It acts as a bridge for **inter-component communication**, allowing data to be passed between Activities, Services, or Broadcast Receivers.

Types of Intents

1. **Explicit Intent:**
   - Used to start a specific component (Activity or Service) within the same application.
   - Example: navigating from one activity to another.
2. Intent intent = new Intent(MainActivity.this, SecondActivity.class);

3. intent.putExtra("username", "John");
4. startActivity(intent);
5. **Implicit Intent:**
   - Used when the target component is not specified.
   - The Android system determines the appropriate app to handle the request.
   - Example:
6. Intent intent = new Intent(Intent.ACTION_VIEW);
7. intent.setData(Uri.parse("https://www.google.com"));
8. startActivity(intent);

Common Uses of Intents

- Launching new activities or services.
- Sending and receiving data between components.
- Triggering system events (e.g., camera, sharing, calls).
- Broadcasting messages to multiple receivers.

Fragment

A **Fragment** is a reusable portion of the UI that can be embedded within an Activity. It represents a part of an activity's interface and has its own lifecycle, which is closely tied to the hosting Activity's lifecycle. Fragments improve modularity and make UI design more flexible, especially on devices with large screens like tablets.

Fragment Lifecycle Methods

1. onAttach() – Called when the fragment is attached to an activity.
2. onCreateView() – Inflates the layout for the fragment.
3. onStart() – Makes the fragment visible.
4. onResume() – Fragment becomes active.
5. onPause(), onStop(), onDestroyView() – Handle cleanup tasks.

Example

```
public class MyFragment extends Fragment {
   @Override
   public View onCreateView(LayoutInflater inflater, ViewGroup container,
                            Bundle savedInstanceState) {
      return inflater.inflate(R.layout.fragment_layout, container, false);
   }
}
```

Adding Fragment in Activity:

```
getSupportFragmentManager()
   .beginTransaction()
   .replace(R.id.container, new MyFragment())
   .commit();
```

14. Explain Dalvik Debug Tool and ADB in Android.

Android provides powerful debugging and communication tools to help developers test, monitor, and manage applications efficiently. Two of the most essential tools are the **Dalvik Debug Monitor Service (DDMS)** and the **Android Debug Bridge (ADB)**. Both play distinct roles in application debugging, testing, and device communication.

1. Dalvik Debug Monitor Service (DDMS)

The **Dalvik Debug Monitor Service**, commonly referred to as **DDMS**, is a debugging tool integrated with Android Studio. It was originally designed to monitor and control the **Dalvik Virtual Machine (DVM)** processes but continues to support the Android Runtime (ART) as well.

Primary Functions of DDMS

- **Thread and Heap Monitoring:** Displays running threads and memory allocation in real-time.
- **Logcat Viewer:** Shows system and application logs for debugging purposes.
- **Screen Capture:** Allows developers to take screenshots of the device screen directly from Android Studio.
- **Network Statistics:** Monitors data sent and received by applications.
- **Location and Telephony Simulation:** Simulates GPS coordinates or incoming calls/SMS for testing.
- **Process Control:** Enables force-stopping, debugging, or profiling of applications.

Working

DDMS connects to devices through ADB and communicates with the **Dalvik/ART runtime**. When a developer runs an app in debug mode, DDMS assigns a unique port to each process, allowing detailed tracking and analysis.

Example Use:

To analyze memory usage, developers can open **Android Profiler → Memory**, which internally uses DDMS functionality to display heap allocation and garbage collection events.

2. Android Debug Bridge (ADB)

ADB is a **command-line tool** that allows communication between a development machine and an Android device or emulator. It provides a versatile interface for debugging, file management, and device control.

Key Features

- **Install/Uninstall Apps:**
- adb install app.apk
- adb uninstall com.example.app
- **Device Shell Access:**
- adb shell

Used for executing commands directly on the device.

- **Logcat Access:**
- adb logcat

Displays system and app logs in real time.

- **File Transfer:**
- adb push <local> <remote>
- adb pull <remote> <local>
- **Reboot and Recovery Control:**
- adb reboot
- adb reboot recovery

- **DDMS** is a graphical debugging and monitoring tool that provides detailed insights into app performance, memory, and processes.
- **ADB** is a command-line tool that facilitates direct interaction with devices for installing, debugging, and managing applications.
  Together, they form the backbone of Android's debugging and development environment.

15. What is parsing? Discuss how you can perform parsing using JSON in an Android application.

Parsing in Android
Parsing is the process of **reading and converting structured data** (like XML or JSON) into a usable format for an Android application. It allows the app to understand data coming from external sources — such as a web API, database, or file — and use it for displaying information or performing operations.
In Android development, **JSON (JavaScript Object Notation)** is the most widely used format for data exchange because it is **lightweight, easy to read, and human-friendly**. JSON represents data as key-value pairs and arrays, making it simple to interpret programmatically.

JSON Parsing in Android
JSON Parsing is the process of **extracting information from a JSON string** and converting it into objects or variables that can be used within the app. For example, if a server sends data about users, the app can parse that JSON response to display usernames, ages, etc.
There are **two main methods** for JSON parsing in Android:

1. **Manual Parsing using JSONObject and JSONArray:**
   Android provides built-in classes in the org.json package to handle JSON data.
   - JSONObject is used for JSON objects (key-value pairs).
   - JSONArray is used for JSON arrays (lists of objects).
2. **Automatic Parsing using Libraries:**
   Libraries like **GSON**, **Moshi**, or **Jackson** automatically convert JSON into Java objects, reducing manual effort and chances of error.

Example: JSON Parsing using JSONObject and JSONArray

```
String jsonData = "{ \"student\": {\"name\":\"Amit\", \"age\":20} }";

try {
   JSONObject jsonObject = new JSONObject(jsonData);
   JSONObject student = jsonObject.getJSONObject("student");
   String name = student.getString("name");
   int age = student.getInt("age");
} catch (JSONException e) {
   e.printStackTrace();
}
```

Parsing helps Android apps read and interpret data in structured formats like JSON.

- **JSON Parsing** is essential when working with APIs or web services.
- Developers can use built-in classes (JSONObject, JSONArray) for simple cases or libraries like **GSON** for more complex data structures.
- It makes data exchange between client and server efficient and human-readable.

16. Briefly explain AsyncTask Loader.

AsyncTaskLoader is a specialized loader in Android used to perform **asynchronous background operations** and deliver results to the user interface efficiently. It is part of the **Loader framework**, introduced to manage background data loading tasks while maintaining proper lifecycle handling with activities and fragments. AsyncTaskLoader simplifies running long or intensive operations (like fetching data from the internet or reading from a database) without freezing the main UI thread.

Purpose

Android applications must keep the main thread (UI thread) responsive. Performing heavy operations such as network calls or database queries on the main thread can cause **Application Not Responding (ANR)** errors.
AsyncTaskLoader handles such operations by executing them in the background, automatically managing configuration changes like screen rotations without losing data.

Key Features

- Runs background tasks efficiently using a separate thread.
- Automatically reconnects to the last loaded data after configuration changes.
- Caches previously loaded results for faster reloads.
- Works seamlessly with the **LoaderManager** to manage data lifecycle.

Lifecycle of AsyncTaskLoader

1. **onStartLoading():**
   Called when the loader starts; triggers data loading.
2. **loadInBackground():**
   Executes the heavy operation (e.g., fetching data). Runs on a background thread.
3. **deliverResult():**
   Sends the computed result back to the main thread for UI updates.
4. **onStopLoading():**
   Stops the current loading operation if it’s no longer needed.
5. **onReset():**
   Cleans up or releases resources when the loader is reset.

17. List and explain components of Android SDK.

Components of Android SDK

The **Android Software Development Kit (SDK)** is a collection of tools, libraries, and APIs required to develop, test, and debug Android applications. It provides everything necessary for developers to build functional and optimized Android apps. Each component plays a specific role in the development process, from coding to deployment.

1. Android SDK Tools
These are essential command-line and graphical tools used for app development and debugging.

- Include utilities like **adb (Android Debug Bridge)**, **emulator**, and **proguard**.
- Help in managing devices, simulating Android environments, and signing applications.
- Example:
  - adb install – installs an APK on a connected device.
  - adb logcat – displays system logs for debugging.

2. Android Platform Tools
These tools are updated with every new Android release and provide compatibility with different Android versions.

- Include **adb**, **fastboot**, and **dmtracedump**.
- Ensure the app can run across multiple device versions and configurations.

3. Android Build Tools
These are used to **compile, package, and convert code** into an Android application (APK).

- Include aapt, aidl, and dx tools.
- Work with Gradle to handle dependencies, generate bytecode, and produce executable APK files.
- aapt (Android Asset Packaging Tool) compiles resources into the application package.

4. Android Emulator
A virtual device that simulates real Android hardware and software on a computer.

- Used for testing applications without needing a physical device.
- Supports different device configurations (screen sizes, OS versions, sensors).

5. Android Libraries
Provide reusable code components for various functionalities.

- Include **android.jar**, containing core classes for UI, graphics, and data handling.
- Third-party libraries can also be added for additional features like image loading or network handling.

6. Android Debug Bridge (ADB)
A command-line interface that allows communication between the development environment and the Android device/emulator.

- Used for installing, debugging, and monitoring applications during runtime.

7. Android Developer Tools (ADT) / Android Studio
The integrated development environment (IDE) for building Android apps.

- Offers code editing, debugging, layout design, and performance analysis tools.
- Works with Gradle for automated builds.

18. How many ways data stored in Android?
Ways to Store Data in Android
Android provides multiple mechanisms to store, manage, and retrieve data efficiently based on the size, structure, and persistence needs of the application. Data storage methods are chosen according to whether data should be private to the app, shared with other apps, or persisted permanently even after the app is closed. The main storage options in Android are as follows:

1. Shared Preferences
Used for storing **small amounts of key-value pair data** such as user settings, login states, or app configurations.

- Stores data as XML files in the app's private directory.
- Suitable for lightweight data such as boolean, int, float, long, and String.
  Example: saving theme preferences or remembering last login status.

```
SharedPreferences prefs = getSharedPreferences("MyPrefs", MODE_PRIVATE);

SharedPreferences.Editor editor = prefs.edit();

editor.putString("username", "admin");

editor.apply();
```

2. Internal Storage
Used for storing **private application data** on the device's internal memory.

- Files saved here are **accessible only by the application**.
- Data remains even when the app is closed but gets deleted when the app is uninstalled.
  Example: saving text files or cached data.

```
FileOutputStream fos = openFileOutput("data.txt", MODE_PRIVATE);

fos.write("Hello Android".getBytes());

fos.close();
```

3. External Storage
Used for storing **large files** such as images, videos, or documents on SD cards or shared device memory.

- Requires storage permissions (READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE).
- Data may be accessible to other apps and the user.
  Example: saving downloaded images or media files.

4. SQLite Database
Provides **structured storage** for relational data using SQL queries.

- Best suited for complex data requiring relationships or filtering.
- Offers persistence and allows CRUD (Create, Read, Update, Delete) operations.
  Example: managing user profiles, product lists, or offline content.

5. Content Providers
Used for **sharing data between applications** in a secure and controlled manner.

- Abstracts access to data sources such as databases, files, or other storage mechanisms.
- Example: Contacts or Media Store providers.

6. Network Storage / Cloud Storage
Used for storing and retrieving data remotely over the internet.

- Services like **Firebase Realtime Database**, **Cloud Firestore**, or REST APIs store data on servers.
- Ensures data synchronization across multiple devices.

19. Explain Checkbox
A **Checkbox** in Android is a user interface (UI) element that allows users to **select one or more options** from a given set of choices. It represents a **two-state button** — checked (true) or unchecked (false) — and is typically used when multiple selections are required in forms, settings, or questionnaires.
Checkboxes are part of the Android widget package and are implemented using the CheckBox class, which extends the CompoundButton class.
Key Features

- Allows **multiple independent selections**.
- Each checkbox maintains its own state (checked/unchecked).
- Can be programmatically controlled using methods like setChecked() and isChecked().
- Triggers an **OnCheckedChangeListener** to detect when the user changes its state.

Syntax (XML Definition)
```
<CheckBox
   android:id="@+id/checkbox_android"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Android Programming"
   android:checked="false" />
```
This defines a checkbox labeled “Android Programming” in the layout.

Accessing and Handling in Java

```
CheckBox checkBox = findViewById(R.id.checkbox_android);

checkBox.setOnCheckedChangeListener((buttonView, isChecked) -> {
   if (isChecked)
      Toast.makeText(this, "Checked", Toast.LENGTH_SHORT).show();
   else
      Toast.makeText(this, "Unchecked", Toast.LENGTH_SHORT).show();
});
```

Here, the OnCheckedChangeListener detects when the checkbox state changes and performs actions accordingly.

Common Use Cases

- Selecting multiple items in a list (e.g., choosing hobbies or interests).
- Enabling or disabling app settings (e.g., notifications, dark mode).
- Confirming terms and conditions or options in registration forms.

Difference from RadioButton

- **Checkbox**: Multiple options can be selected simultaneously.
- **RadioButton**: Only one option can be selected within a group.

20. List out UI Components in Android Application. Explain three in brief

User Interface (UI) components in Android are the **building blocks of application layouts**. They are the visual elements that allow users to **interact with an app** through input, selection, and display mechanisms. Android provides these UI elements in the form of **Views** and **ViewGroups**, which are organized using XML layouts or programmatically in Java/Kotlin code.

Common UI Components in Android

1. TextView
2. EditText
3. Button
4. ImageView
5. CheckBox
6. RadioButton
7. ToggleButton
8. Spinner
9. ListView
10. RecyclerView
11. ProgressBar
12. Switch
13. SeekBar
14. WebView
15. RatingBar

1. TextView

Purpose:

Used to display **text** to the user, such as labels, messages, or descriptions.

It is one of the most basic UI components and supports features like text color, font style, alignment, and size.

Key Points:

- Non-editable by default (used for static text).
- Can display formatted text using HTML or string resources.
- Often used to show titles, hints, or output results in an application.

Example Use: Showing headings like *"Welcome to Android App"* or status messages.

2. EditText

Purpose:

Used for **user input** of text data, such as names, emails, or passwords.
It extends the TextView class but allows users to modify the text content.

Key Points:

- Can accept different input types (text, number, email, password).
- Supports input validation and hint messages.
- Frequently used in forms and login screens.

Example Use: Accepting username and password in a login form.

3. Button

Purpose:

A **clickable UI element** used to perform an action or trigger an event when pressed.
It provides an interactive way to submit data, navigate, or initiate operations.

Key Points:

- Supports text or image labels.
- Responds to user clicks through onClick events.
- Commonly used for actions like "Submit", "Next", or "Cancel".

Example Use: A "Submit" button that validates form input and sends data to the next activity.