Android Interview

Android interview Questions 2025

- Android is a widely used open-source operating system developed by Google, primarily designed for mobile devices such as smartphones, tablets, smartwatches, TVs, and other connected devices. Based on the Linux kernel, Android powers a large variety of devices, providing users with an interactive and versatile mobile experience.

The Linux kernel is the part of the OS that interacts directly with the hardware (CPU, memory, devices).

1. Early Beginnings (2003-2005)

- **2003**: Android Inc. was founded by Andy Rubin and team, initially as an OS for digital cameras.

- **2005**: Google acquired Android Inc., bringing resources and expertise to the project.

2. Development Under Google (2005-2007)

- Android was developed to be an open-source, flexible mobile OS, based on the **Linux kernel** to compete with systems like Symbian and BlackBerry.

3. The First Android Phone and Launch (2008)

- **2008**: The first Android phone, **T-Mobile G1 (HTC Dream)**, launched with Android 1.0, featuring a touchscreen, physical keyboard, and access to **Google services**.

4. Android's Growth and Key Releases (2009-2011)

- **2009**: Android 1.5 (Cupcake) introduced **widgets** and the **Android Market**.

Android interview Questions

- **2011**: Android 3.0 (Honeycomb) for tablets, followed by Android 4.0 (Ice Cream Sandwich) unifying phone and tablet features.

5. Android Becomes Dominant (2012-2014)

- **2012**: Android 4.1 (Jelly Bean) introduced **Project Butter** for smoother UI and **Google Now**.
- **2014**: Android 5.0 (Lollipop) introduced **Material Design**, a unified design language.

6. Consolidation and Innovation (2015-2019)

- **2015**: Android 6.0 (Marshmallow) brought **Doze mode** and **Google Now on Tap**.
- **2017**: Android 8.0 (Oreo) added **Picture-in-Picture** mode and **background app limits**.
- **2018**: Android 9.0 (Pie) introduced **gesture navigation** and **privacy improvements**.
- Android expanded into **Wear OS**, **Android TV**, and **Android Auto**.

7. Android's Modern Era (2020-Present)

- **2020**: Android 10 moved to numerical versioning and introduced **dark mode** and **privacy features**.
- **2021**: Android 12 introduced **Material You** and **privacy controls**.
- **2023-2024**: Android 13 (Tiramisu) and 14 brought improved **multitasking**, **foldable device support**, and enhanced **security**.
- **Android version 16 launched on 10 june 2025 , API level 36**

Android Studio Narwhal | 2025.1.1 (Patch 1) july 10 2025

Android interview Questions

An **Operating System (OS)** is system software that manages hardware resources and provides services for computer programs. It acts as an intermediary between computer hardware and the software applications that run on a device, enabling the efficient operation of the device and allowing users and applications to interact with it.

Key Functions of an Operating System:

1. **Hardware Management**: The OS manages the computer's hardware resources, including the CPU, memory (RAM), storage devices, and input/output devices like the keyboard, mouse, and display.

2. **Task Management**: It schedules tasks and manages processes, allowing multiple applications to run simultaneously through multitasking.

3. **Memory Management**: The OS allocates and deallocates memory for applications and processes, ensuring efficient use of system resources.

4. **File System Management**: It handles the storage, organization, and access to files, folders, and directories on the storage devices (e.g., hard drives, SSDs).

5. **Security and Access Control**: The OS ensures that only authorized users and programs can access certain resources, protecting against unauthorized access and threats.

6. **User Interface (UI)**: It provides a way for users to interact with the system, either through a **graphical user interface (GUI)** or a **command-line interface (CLI)**.

7. **Networking**: The OS manages network connections and communication, allowing devices to connect to the internet and other systems.

Examples of Operating Systems:

- **Desktop/PC OS**:
    - **Windows** (by Microsoft)
    - **macOS** (by Apple)
    - **Linux** (various distributions like Ubuntu, Fedora, etc.)

Android interview Questions

- **Mobile OS**:
  - **Android** (by Google)
  - **iOS** (by Apple)
  - **HarmonyOS** (by Huawei)
- **Embedded OS**:
  - **RTOS (Real-Time Operating System)** for devices like robots, medical equipment, etc. or **Embedded Linux** for appliances, IoT devices, and more.

Why Kotlin we have Java

1. Conciseness

- **Kotlin** is more concise than **Java**. It requires fewer lines of code to achieve the same functionality, which makes the codebase more readable and maintainable.
- Example: Kotlin has built-in features like data classes, which eliminate the need for verbose code for objects, constructors, getters, and setters.

2. Null Safety

- **Kotlin** has built-in **null safety**. It helps avoid **NullPointerExceptions** (NPE), a common source of runtime errors in Java.
- Kotlin distinguishes nullable and non-nullable types, forcing developers to handle null values explicitly and safely.

3. Extension Functions

- Kotlin allows developers to add new functions to existing classes without modifying their source code, using **extension functions**.
- This provides better flexibility and allows cleaner, more modular code.

Android interview Questions

- **Kotlin** is fully **interoperable with Java**, meaning you can call Kotlin code from Java and vice versa.

- Kotlin is designed to work seamlessly with the Android framework, which was initially built in Java.

5. Coroutines for Asynchronous Programming

- Kotlin offers **Coroutines** for asynchronous programming, which are easier to use and more efficient than Java's traditional **AsyncTask** or **Handler**.

- Coroutines make it simple to handle background tasks like network calls or database queries, reducing the complexity of callback-based code.

6. Smart Casts

- Kotlin has **smart casting**, which automatically casts objects to the appropriate type after checking their type. This reduces the need for explicit casting, making the code more concise and less error-prone.

```
fun main(){

    val obj="Rahul"

    val num=12

    val un= ""

    smartDescribe(obj)

    smartDescribe(num)

}



fun smartDescribe(obj: Any) {
```

```
        is String -> println("String of length ${obj.length}")

        is Int -> println("Integer: ${obj + 1}")

        else -> println("Unknown type")

    }

}
```

7. Better Syntax and Readability

- Kotlin's syntax is more modern and expressive. It removes boilerplate code present in Java, like the need for semicolons, getters/setters, and explicit type declarations (in many cases, Kotlin can infer types).
- **Lambda expressions** and **functional programming features** are more easily implemented in Kotlin, making code cleaner and more readable.

8. Default Arguments and Named Parameters

- **Kotlin** allows default values for parameters and named arguments, which reduces the need for multiple overloaded methods as in Java.
- This improves code clarity and simplifies method calls.

✅ Benefits of Default Parameters & Named Arguments in Kotlin

Android interview Questions

In Java, you often create multiple overloaded methods to handle optional parameters:

```
// Java
void greet(String name) {
    greet(name, "Hello");
}
void greet(String name, String greeting) {
    System.out.println(greeting + ", " + name);
}
```

Kotlin replaces this with a single function:

```
fun greet(name: String, greeting: String = "Hello") {
    println("$greeting, $name")
}
```

- **Benefit**: Less code, less duplication, easier to maintain.

You can specify which argument is being passed, improving clarity:

```
greet(name = "Alice", greeting = "Hi")

greet("Bob") // Uses default greeting
```

🔹 **Benefit**: Makes function calls self-documenting — especially useful when functions have multiple parameters of the same type.

---

3. 📉 Reduced Boilerplate

- You don't need to write every combination of parameters.
- Cleaner API surface, especially for libraries and SDKs.

---

4. 🔁 Flexible Parameter Order (with Named Args)

You can change the order of arguments when calling a function:

```
fun draw(x: Int = 0, y: Int = 0, color: String = "black") {
    println("Drawing at ($x, $y) in $color")
}

// Call with any order using named arguments
```

- **Benefit**: Greater flexibility for callers and easier evolution of APIs.

---

5. 💡 Better Defaults for Clean APIs

You can provide **sensible defaults** so developers don't need to learn all parameters:

```
fun createUser(name: String, isAdmin: Boolean = false)
```

- **Benefit**: Encourages best practices and reduces misuse.

⬅ END Summary Table

Feature	Java	Kotlin	Benefit

Android interview Questions

ed arguments	❌ Not available	✅ Supported	More readable calls
neter reordering	❌ Not possible	✅ Named args allow it	Flexible usage
ılt values	❌ Must simulate via overloads	✅ Built-in	Cleaner, more maintainable code
volution	Requires overloads	Just add defaulted params	Backward-compatible improvements

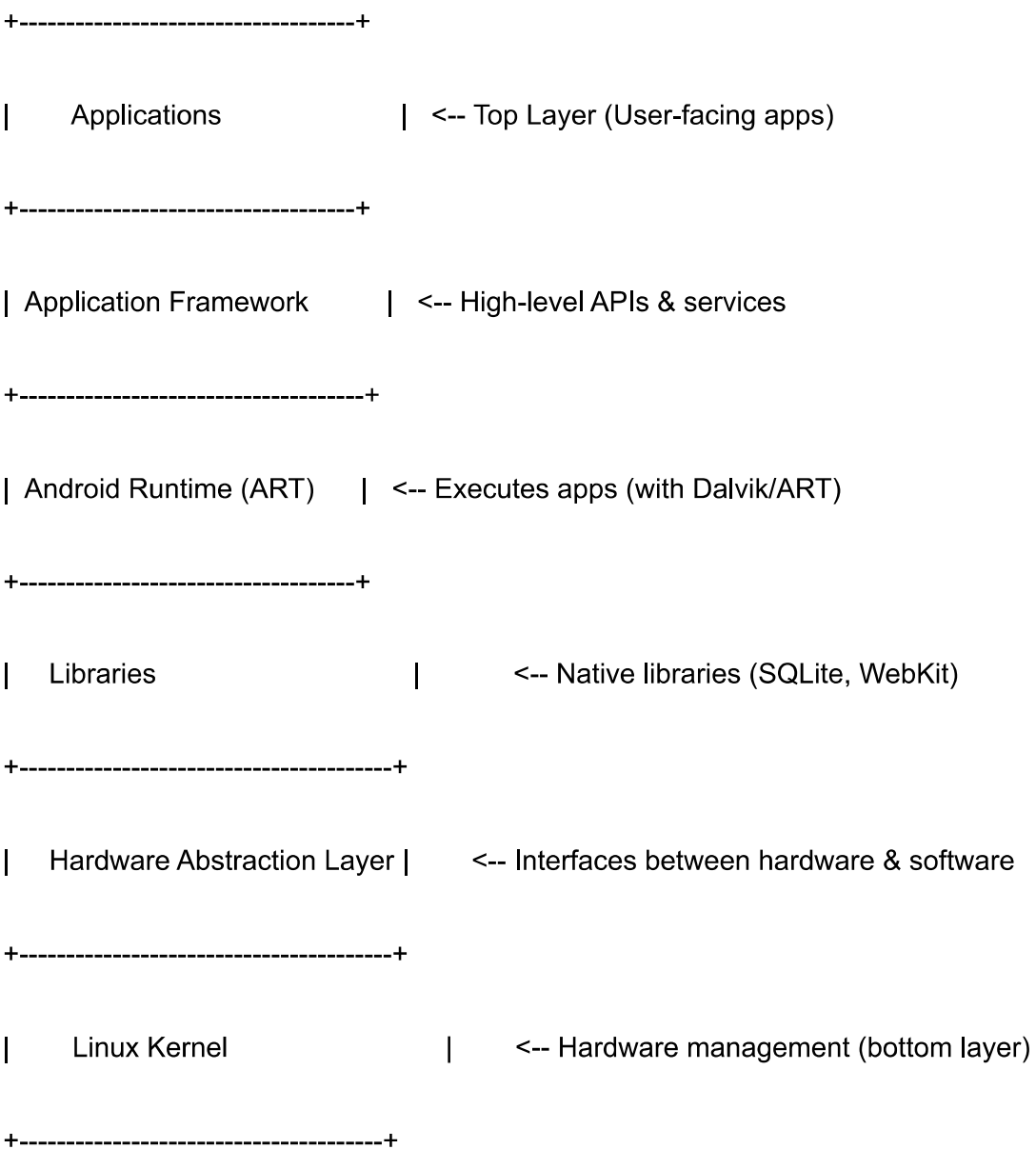
10. Modern Language Features

- Kotlin comes with modern features such as **seals, destructuring declarations, destructuring in loops**, and **lambda expressions**, which Java lacks in its earlier versions.
- It also supports **functional programming** paradigms, which gives developers more flexibility.

11. Popularity and Community Support

- Since **Google** announced Kotlin as the preferred language for Android development in 2017, its popularity has grown significantly.

Android interview Questions



1. Linux Kernel

The **Linux Kernel** is the foundation of the Android architecture and provides low-level system services. It handles core system functions, including:

Android interview Questions

- **Process management**
- **Networking**
- **Security**
- **Driver support** (for hardware like camera, Wi-Fi, Bluetooth, etc.)

The Android operating system is built on top of this kernel, making it responsible for interacting with hardware and managing system resources efficiently.

2. Hardware Abstraction Layer (HAL)

The **Hardware Abstraction Layer (HAL)** is an intermediary between the hardware and the higher-level software (such as the operating system and framework). HAL provides standard interfaces to communicate with various hardware components without the need for OS-level modification.

- It ensures that Android can work on a wide variety of hardware platforms by abstracting the device-specific hardware features.
- Examples of HAL components include camera, Bluetooth, GPS, and audio systems.

3. Android Runtime (ART)

The **Android Runtime (ART)** is responsible for running Android applications. ART is a critical component and provides the following:

- **Dalvik Virtual Machine (DVM)**: Prior to Android 5.0, Android used DVM for executing app code. DVM was optimized for mobile devices with limited resources.
- **ART (Android Runtime)**: Replaced DVM in Android 5.0 (Lollipop). ART uses **Ahead-of-Time (AOT) compilation**, which compiles app code into machine code when the app is installed, resulting in faster performance and reduced memory usage.
- **Libraries**: ART includes essential runtime libraries like memory management, threading, and I/O management.

4. Libraries

features. Key libraries include:

- **WebKit**: A library for .
- **SQLite**: A lightweight relational database used for storing and managing local app data.
- **OpenGL ES**: A graphics library that supports rendering 2D and 3D graphics.
- **Media Framework**: For audio, video, and multimedia playback and recording.
- **Surface Manager**: Manages display and graphics rendering.
- **SSL libraries**: Handle security for encrypted communication.

5. Application Framework

The **Application Framework** provides the necessary tools and APIs to help developers build Android apps. It includes higher-level services and reusable components that Android applications rely on. Key components of the Application Framework include:

- **Activity Manager**: Manages the lifecycle of activities and application components (e.g., start, pause, resume).
- **Window Manager**: Handles the layout and display of UI components.
- **Content Providers**: Manage data and allow sharing between applications.
- **View System**: Provides UI components like buttons, text fields, and layouts.
- **Resource Manager**: Manages access to resources such as images, strings, and layout files.
- **Notification Manager**: Manages and sends notifications to the user.
- **Location Manager**: Provides location-based services like GPS.
- **Telephony Manager**: Handles telephony services like making calls and messaging.

6. Applications

- **Core Applications**: Pre-installed apps like Phone, SMS, Email, Calendar, and Contacts.
- **Third-party Applications**: Apps installed by the user, typically from the **Google Play Store** or other sources.
- Apps interact with the Application Framework and make use of the services and APIs provided by the lower layers.

Launch Modes

1. Standard (Default Launch Mode)

- **Description**: This is the default launch mode for an activity. Each time an activity is launched, a new instance of that activity is created, even if it already exists in the activity stack.
- **Behavior**: Every time the activity is launched, it creates a new instance and pushes it onto the stack.
- **Use Case**: Suitable for most situations where you want each instance of an activity to be independent.

Example: If you launch the same activity multiple times from different places in your app, each time it will create a new instance.

2. SingleTop

- **Description**: If an instance of the activity is already at the top of the activity stack, it will **not** be re-created. Instead, the system will deliver the intent to the existing instance's onNewIntent() method.
- **Behavior**: If the activity is at the top of the stack, a new instance will **not** be created, and the system will call onNewIntent() to handle the new intent.
- **Use Case**: Useful when you want to avoid creating multiple instances of the same activity at the top of the stack, such as when an activity is intended to handle certain actions repeatedly (e.g., notification handling).

Android interview Questions

3. SingleTask

- **Description**: When an activity is launched with the singleTask launch mode, the system checks whether an instance of the activity already exists in the activity stack. If it does, the system will bring that existing instance to the foreground and call its onNewIntent() method, while clearing any activities on top of it in the stack.

- **Behavior**: Only one instance of the activity will exist in the entire task. If an instance exists, it will be reused, and any activities above it in the stack will be removed.

- **Use Case**: Suitable for activities that should be unique within a task, such as a home screen or main menu, where only one instance should be active.

Example: Use this for the main activity that should always be the root of the task, such as a main screen or dashboard.

4. SingleInstance

- **Description**: The singleInstance launch mode is similar to singleTask, but it enforces that **only one instance** of the activity will exist in the system, and it will **always be in its own task.** No other activities will be in the same task as this activity.

- **Behavior**: It is guaranteed to be the only activity in its task. When launched, it will always be placed in a new task, and no other activities will be able to join this task.

- **Use Case**: Suitable for activities like a **video player**, **map** application,login screen in banking system or other apps that should operate independently of other activities.

Example: Use this for an activity that should not share its task with other activities, such as a media player or system-level activity.

Task and Activity Stack:

- When an activity is launched, it is pushed onto the **activity stack** (also called the **back stack**).
- The **task** is a collection of activities that are related to a single application or action.
- The **launch mode** determines how activities behave when they are pushed or popped from the stack.

Android interview Questions

The main components of an Android application are:

1. Activities

- **Definition**: An **Activity** represents a single screen in an app. It is where users interact
- with the app. Each activity corresponds to a specific user interface and provides an entry point to different features of the app.
- **Purpose**: It manages the user interface and handles user input.
- **Example**: A login screen, home screen, or settings screen.
- **Lifecycle**: Activities go through several lifecycle stages, such as onCreate(), onStart(), onResume(), onPause(), onStop(), and onDestroy().
-

🔁 Example Scenario: Activity B Restart (like screen rotation)

```
→ onPause()      // Activity is pausing
→ onStop()       // Activity is no longer visible
→ onDestroy()    // Activity is destroyed
---
→ onCreate()     // Re-created
→ onStart()      // Becoming visible
→ onResume()     // Ready for user interaction
```

✅ Scenario: Activity A → B → C then C-> B->A

Action	A	B	C

Android interview Questions

		`onResume()`	
Start C from B		`onPause()`	`onCreate()`, `onStart()`, `onResume()`
Back from C to B		`onResume()`	`onPause()`, `onStop()`, `onDestroy()`
Back from B to A	`onResume()`	`onPause()`, `onStop()`, `onDestroy()`	

📤 Output on pressing back from C to B:

```
C: onPause

C: onStop

C: onDestroy

B: onResume
```

🔸 **Q: What happens to the lifecycle methods when navigating from Activity A → B → C and then pressing back?**
✅ A:

- Each new activity is **pushed onto the back stack.**
- The previous activity is **paused** but **not destroyed.**
- When you press back:
  - Current activity is destroyed (`onPause()`, `onStop()`, `onDestroy()`).

Android interview Questions

✅ **Step 1: From Activity A → Start Activity B with Data**

```
// In FirstActivity.kt
val intent = Intent(this, SecondActivity::class.java)
intent.putExtra("username", "Rahul Kumar")
startActivity(intent)
```

✅ **Step 2: In Activity B → Receive the Data**

```
// In SecondActivity.kt
class SecondActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // Get the data
        val name = intent.getStringExtra("username") ?: "Guest"

        setContent {
```

Android interview Questions

```
                }
            }
        }
```

✅ Step 3: Compose UI to Show Passed Data

```
@Composable
        fun SecondScreen(name: String) {
            Surface(modifier = Modifier.fillMaxSize(), color = Color.White) {
                Column(
                    verticalArrangement = Arrangement.Center,
                    horizontalAlignment = Alignment.CenterHorizontally,
                    modifier = Modifier.fillMaxSize()
                ) {
                    Text(text = "Welcome, $name!", fontSize = 24.sp)
                }
            }
```

Android interview Questions

📝 Pass Data Activity to Activity $ F-F

Direction	Method	Example
Activity → Activity	Intent.putExtra()	intent.putExtra("key", value)
Fragment → Fragment	Jetpack Compose Nav route args	"screen/{arg}" + arguments
Compose → Compose	NavController with routes	navigate("details/value")

Activities are **tightly integrated with the system**, which makes them heavier.

OR

An **Activity is heavier** than a Fragment because it is a complete, self-contained component with its own lifecycle, UI window, context, and system registration.

A **Fragment is lighter** since it exists within an Activity, shares resources, and is easier to create, destroy, and navigate.

📦 Data Types You Can Pass

You can pass all primitive types:

- `putString()`, `putInt()`, `putBoolean()`, `putFloat()`

Android interview Questions

🧠 Interview Insight:

Q: How do you pass data between activities in Jetpack Compose?
✅ A: Use `Intent.putExtra()` in the launching activity, and access it in `onCreate()` of the target activity before setting Compose content.

✅ Bonus: Passing Objects

If you want to pass a custom object (e.g., a data class), make it **Parcelable**:

```
@Parcelize

data class User(val id: Int, val name: String) : Parcelable
```

code//

```
 intent.putExtra("user", user)

val user = intent.getParcelableExtra<User>("user")
```

USE **Jetpack Compose Navigation (`navController`)** to pass data between screens instead of activities?

🔄 Scenario: B is *recreated due to rotation* or system-initiated recreation (while on B)

If you're already on **B**, and you **rotate the device** or Android **kills and recreates B**, the following happens:

Android interview Questions

```
B: onPause()

B: onStop()

B: onDestroy()    ------ destroy then

B: onCreate()

B: onStart()

B: onResume()
```

✅ C and A are **not affected** in this case. Only **B** is restarted.

✅ **Case 1: You're on Activity C, and you restart B using an `Intent`**

```
val intent = Intent(this, B::class.java)

startActivity(intent)
```

By default, this creates **another instance** of B **on top of C**, resulting in stack: **A → B → C → B**

📜 **Lifecycle Calls:**

- **C:**
  - `onPause()` (because it's going to the background)
- **B (new instance):**
  - `onCreate()`
  - `onStart()`

➡️ You now have two B instances in the stack (not usually desired).

2. Services

In Android, a Service is a component that performs long-running operations in the background without a user interface. It's designed for tasks that don't require direct interaction and can continue running even when the user navigates away from the application. There are different types of services, including started services, bound services, foreground services, and background services, each with its own characteristics and use cases.

Types of Android Services:

Started Services:

These are started by `startService()` and run until explicitly stopped or until they finish their task. They are suitable for tasks like downloading files or playing music in the background.

Bound Services:

These are bound to other application components, allowing them to interact with and use the service. A bound service is active only when there are components bound to it. Once all bound components unbind, the service can be destroyed.

Foreground Services:

These services perform tasks that are noticeable to the user, requiring them to display a notification in the status bar. They are suitable for tasks like music playback or location tracking, where the user is aware of the service's activity.

Background Services:

These perform tasks in the background without direct user interaction. However, Android 8.0 (Oreo) and higher have restrictions on background services for battery and performance optimization. Background services may be killed by the system to reclaim

Android interview Questions

When to use a Service:

• When you need to perform long-running operations that don't require a user interface.

• When you need to perform tasks that should continue even when the user switches to another app.

• When you need to interact with other components of your application or even other applications.

Key points about Services:

• Services do not have a user interface.

• They run in the background and can be started and stopped.

• They can be bound to other components for interaction.

• Services can be killed by the system if resources are needed for foreground tasks.

• **Definition**: A **Service** is a component that runs in the background to perform long-running tasks without a user interface. It can continue to run even if the user is interacting with other applications.

• **Purpose**: To perform tasks like downloading files, playing music, or handling network operations, even when the app is not in the foreground.

• **Example**: A service for playing music in the background or for syncing data with a server.

• **Types**:

  ○ **Started Service**: Runs indefinitely until it is explicitly stopped.

- **Foreground Service**s : Runs with notifications ,user knows its running —>Fitness tracker ,GPS tracker
- **Background Services**: Runs in Backgroung → file downloading ,photo uploading ,

✅ 3. Service Lifecycle (Important!)

There are **two types** of services and their **lifecycle differs**:

🔹 a) Started Service

- Starts when `startService()` is called.
- Runs in the **background indefinitely**.
- Stops with `stopSelf()` or `stopService()`.

Lifecycle Methods:

```
onCreate()   // Called once when service is created
onStartCommand()  // Called every time startService() is called
onDestroy()  // Called when service is stopped
```

🔹 b) Bound Service

- Bound to an **Activity or component** using `bindService()`.
- Lives only as long as the components are bound.
- Allows **two-way communication**.

Lifecycle Methods:

```
onCreate()
onBind()     // Returns IBinder for communication
```

Android interview Questions

Q: Will a Service run on a background thread?

A: No! By default, Service code runs on the main thread. You must create your own background thread (e.g., Coroutine, HandlerThread, or ThreadPool).

✅ 6. Types of Services

Type	Description	Use Case
Started Service	Runs until stopped explicitly.	Music, Downloads
Bound Service	Bound to a component. Dies when all clients unbind.	Media playback with UI controls
Foreground Service	Displays a notification. Cannot be killed easily.	GPS tracking, fitness apps
JobIntentService	Handles work on background thread, compatible with older APIs.	Background sync
IntentService (Deprecated)	Like a Started Service, runs on a worker thread.	Upload/Sync tasks

Feature	Explanation
Services run on	Main Thread by default
Problem	Blocking main thread causes UI freezes, ANRs
Solution	Use `Thread`, `Coroutine`, or `ExecutorService`
Best Practice	Never run long tasks directly in `onStartCommand()` or `onBind()`

✅ 3. Best Practice

- **Never run long tasks directly in `onStartCommand()`. Use:**
  - **`Thread`**
  - **`ExecutorService`**
  - **`Coroutines`**
- **For long-lived tasks (like location tracking), use Foreground Service.**

Basic example

```
class MyService : Service() {



    override fun onCreate() {

        super.onCreate()

        // Called only once when the service is first created

    }



    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {

        // This is where the service does its work

        // Important: This runs on the main thread!



        // Example: Start a background task
```

```
        // Simulate background work
        for (i in 1..5) {
            Log.d("MyService", "Running task $i")
            Thread.sleep(1000)
        }
        stopSelf() // Stop service after task is done
    }.start()

    return START_NOT_STICKY // Or START_STICKY, etc.
}

override fun onDestroy() {
    super.onDestroy()
    Log.d("MyService", "Service destroyed")
}

override fun onBind(intent: Intent?): IBinder? {
    return null // This is a StartedService, not a BoundService
}
```

Android interview Questions

START_STICKY,---> means service that need to continue running in the backgroung even if they are killed or stopped.

🚀 Common Ways to Perform Background Tasks:

Method	Best For	Lifecycle Aware	API Support
✅ Coroutines (Kotlin)	Lightweight async tasks in app	Yes	Android Jetpack
✅ WorkManager	Guaranteed background work (even after reboot)	Yes	API 14+
✅ Services	Long-running background tasks	No	API 1+
Handler/Thread	Low-level threading (manual)	No	API 1+
AsyncTask ❌ (deprecated)	Legacy short background tasks	No	Deprecated since API 30

3. Broadcast Receivers

- **Definition**: A **Broadcast Receiver** listens for system-wide or app-specific broadcast messages. It allows apps to respond to various system-wide events like Wi-Fi status changes, battery level warnings, or incoming messages.
- **Purpose**: To listen for and respond to events, without requiring a user interface.
- **Example**: An app might use a broadcast receiver to listen for changes in network connectivity or for a broadcast intent indicating the device is charging.
- **Life Cycle**: Broadcast receivers are short-lived and are activated when they receive an intent. They don't have a UI and are not in the activity stack.

Android interview Questions

A BroadcastReceiver is a component that allows your app to receive and respond to broadcast Intents sent by the Android system or other apps.

📌 **It acts like an event listener that reacts to:**

- **System events (e.g., boot, connectivity change)**
- **Custom broadcasts within or between apps**

✅ 2. What are some common use cases for BroadcastReceiver?

- **React to device boot (`BOOT_COMPLETED`)**
- **Monitor internet connectivity changes**
- **Detect incoming SMS**
- **Track battery status (`BATTERY_LOW`)**
- **Trigger tasks when airplane mode toggles**

✅ 3. How do you define a BroadcastReceiver?

Extend the `BroadcastReceiver` class and override `onReceive()`:

```
class MyReceiver : BroadcastReceiver() {

    override fun onReceive(context: Context, intent: Intent) {

        // Handle event here

    }

}
```

Android interview Questions

✅ 4. How do you register a BroadcastReceiver in the manifest?

```
<receiver android:name=".MyReceiver" android:exported="true">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

Also add `<uses-permission>` if required.

✅ 5. How do you register a BroadcastReceiver dynamically?

```
val receiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context?, intent: Intent?) {
        // Handle
    }
}

val filter = IntentFilter(Intent.ACTION_AIRPLANE_MODE_CHANGED)
```

Android interview Questions

📌 **Unregister in `onStop()` or `onPause()` using `unregisterReceiver()`.**

✅ 6. What is the difference between static and dynamic registration?

Type	Registered In	When Active
Static	`AndroidManifest.xml`	Always (app restarts when needed)
Dynamic	In code (Activity, Service)	While component is alive

✅ 7. Which thread does `onReceive()` run on?

It runs on the main (UI) thread.

Never perform long tasks here. Use:

- **A background `Thread`**
- **`Service` or `JobIntentService`**
- **Kotlin `CoroutineScope(Dispatchers.IO)`**

✅ 8. How can you perform long operations in `onReceive()`?

Avoid blocking the main thread. Instead:

```
    CoroutineScope(Dispatchers.IO).launch {

        // Long-running task

    } }
```

Or start a `Service`/`WorkManager` for background work.

✅ 9. What are ordered broadcasts?

Ordered broadcasts are delivered one receiver at a time in priority order.

- **You can abort the broadcast.**
- **Use `sendOrderedBroadcast()` to send it.**

```
val intent = Intent("com.example.MY_ACTION")

sendOrderedBroadcast(intent, null)
```

✅ 10. What is a sticky broadcast?

A sticky broadcast stays in the system after it is sent, allowing future receivers to get the last broadcasted value.

```
sendStickyBroadcast(intent) // Deprecated
```

⚠️ **Sticky broadcasts are deprecated due to memory and security concerns.**

```
val intent = Intent("com.example.CUSTOM_ACTION")

sendBroadcast(intent)
```

To receive it:

```
val filter = IntentFilter("com.example.CUSTOM_ACTION")

registerReceiver(receiver, filter)
```

✅ 12. Can a BroadcastReceiver start an Activity or Service?

✅ **Yes.**

- **You can launch an `Activity` using `Intent.FLAG_ACTIVITY_NEW_TASK`.**
- **You can also start a `Service`.**

```
val serviceIntent = Intent(context, MyService::class.java)

context.startService(serviceIntent)
```

Android interview Questions

If you don't unregister it, you risk:

- **Memory leaks**
- **IllegalStateException if context is gone**

📌 **Always unregister in `onPause()` or `onStop()`.**

✅ 14. Why are some broadcasts restricted from being declared in the manifest (Android 8+)?

From API 26 (Android 8.0), implicit broadcasts cannot be declared in the manifest.

📌 **Reason: Improve performance and battery life.**

Allowed static ones include:

- `BOOT_COMPLETED`
- `SMS_RECEIVED`
- `BATTERY_CHANGED`

✅ 15. Can two receivers listen to the same broadcast?

✅ **Yes. Multiple receivers can listen to the same broadcast (e.g., `AIRPLANE_MODE_CHANGED`), especially if it's a normal broadcast.**

With ordered broadcasts, receivers are notified one-by-one in priority order.

✅ 16. What happens when you call `abortBroadcast()`?

This prevents the broadcast from being passed to the remaining receivers.

📌 **Works only with ordered broadcasts.**

```
    abortBroadcast()

}
```

✅ 17. What is the difference between `BroadcastReceiver` and `LocalBroadcastManager`?

Feature	BroadcastReceiver	LocalBroadcastManager (deprecated)
Scope	System-wide or app-wide	App-only
Performance	Slower (can cross app boundaries)	Faster
Security	May expose to other apps	More secure (internal only)

✅ **Recommended alternatives: SharedFlow, LiveData, or callback interfaces.**

✅ 18. Can a BroadcastReceiver be triggered when the app is not running?

✅ **Yes, if registered statically (in manifest) for eligible system broadcasts (e.g., `BOOT_COMPLETED`).**

But from Android 8+, the system may delay or restrict broadcast delivery unless your app is in the foreground or has special permissions.

Android interview Questions

✅ 19. Can BroadcastReceiver update the UI?

🚫 **Directly? No.**

Since it runs in the background, `onReceive()` may not have access to current UI context.

✅ **Solutions:**

- **Use `LiveData`/`Flow` to update UI via `ViewModel`**
- **Use `Handler`/`ViewModelScope`**
- **Use `Context` to send a notification**

✅ 20. What is the lifecycle of a BroadcastReceiver?

- **Very short-lived.**
- **Exists only while `onReceive()` is executing.**
- **Once it returns, the receiver is considered dead.**
- **If you need more time (e.g., for downloads), start a `Service`.**

✅ Bonus Tip: When to use BroadcastReceiver vs other components?

Use Case	Best Component
React to system events	BroadcastReceiver

Android interview Questions

UI-bound data	**LiveData, ViewModel**
Internal app messaging	**Flow, Channel, EventBus (not recommended)**

4. Content Providers

- **Definition**: A **Content Provider** is a component that manages access to structured data and allows data to be shared between applications. It provides a standard interface to interact with data (such as contacts, media, or app data) across different apps.
- **Purpose**: To store and retrieve data from shared databases, files, or content across applications.
- **Example**: Accessing contacts, calendar events, or media files (such as images or videos) from the device's storage.
- **URI (Uniform Resource Identifier)**: Data in a content provider is accessed through a URI, and apps can use content resolver APIs to interact with the provider.

Additional Important Components

5. Manifest File

- **Definition**: The AndroidManifest.xml file is a required part of every Android application. It provides essential information about the app, such as its components (activities, services, broadcast receivers, content providers), permissions, hardware requirements, and more.
- **Purpose**: To declare the structure and configuration of the app to the Android system.
- **Example**: Declaring the activities and their corresponding launch modes, as well as permissions like internet access or camera usage.

6. Intents

Android interview Questions

- **Purpose**: To facilitate communication between different components (e.g., start a new activity or service, send a broadcast, etc.).
- **Types**:
  - **Explicit Intents**: Used to specify the exact component to be launched.
  - **Implicit Intents**: Used to request an action without specifying the exact component (e.g., opening a web page or sending an email).

Kotlin:

val intent = Intent (this, NewActivity :: class.java)

startActivity (intent)

✅ 1. Explicit Intent in Jetpack Compose

📌 **Used to launch a specific activity in your app.**

▶️ **Example: Start `SecondActivity` from a Composable**

```
@Composable

fun FirstScreen() {

    val context = LocalContext.current



    Button(onClick = {

        val intent = Intent(context, SecondActivity::class.java)

        context.startActivity(intent)

    }) {
```

```
    }

}
```

✅ This is an **explicit intent** because you're directly specifying the target activity class.

✅ 2. Implicit Intent in Jetpack Compose

📌 **Used to ask Android to handle an action (like opening a URL, sending email, etc.).**

▶️ **Example: Open a URL in the browser**

```
@Composable

fun OpenWebButton() {

    val context = LocalContext.current

    val url = "https://www.google.com"



    Button(onClick = {

        val intent = Intent(Intent.ACTION_VIEW, Uri.parse(url))

        context.startActivity(intent)

    }) {

        Text("Open Google")}}
```

Activity Lifecycle

Android interview Questions

onCreate	**called when activity is first created.**
onStart	**called when activity is becoming visible to the user.**
onResume	**called when activity will start interacting with the user.**
onPause	**called when activity is not visible to the user.or another activity comes in foreground**
onStop	**called when activity is no longer visible to the user.**
onRestart	**called after your activity is stopped, prior to start.**
onDestroy	**called before the activity is destroyed.**

Fragment

In Android, the fragment is the part of the Activity that represents a portion of the User Interface(UI) on the screen. It is the modular section of the Android activity that is very helpful in creating UI designs that are flexible in nature and auto-adjustable based on the device screen size.

Fragments in android are always embedded in Activities i.e., they are added to the layout of activity in which they reside. Multiple fragments can be added to one activity. This task can be carried out in 2 ways:

1. Statically: Explicitly mention the fragment in the XML file of the activity. This type of fragment can not be replaced during the run time.

Lifecycle of Fragment

onAttach()

onCreate()

onCreateView()

onViewCreated()

onStart()

onResume()

onPause()

onStop()

onDestroyView()

onDestroy()

onDetach()

Difference b/w Activity and Fragment

Activity	**Fragment**
Activity is an application component that gives a user interface where the user can interact.	The fragment is only part of an activity, it basically contributes its UI to that activity.

Android interview Questions

	independently.
we need to mention all activity it in the manifest.xml file	Fragment is not required to mention in the manifest file
We can't create multi-screen UI without using fragment in an activity,	After using multiple fragments in a single activity, we can create a multi-screen UI.
Activity can exist without a Fragment	Fragment cannot be used without an Activity.
Creating a project using only Activity then it's difficult to manage	While Using fragments in the project, the project structure will be good and we can handle it easily.
Lifecycle methods are hosted by OS. The activity has its own life cycle.	Lifecycle methods in fragments are hosted by hosting the activity.
Communication between Activity by Intent	Communication between Fragment by interface/callback

add() Method:

- **Purpose**: The add() method adds a new fragment to the activity, without removing or replacing any existing fragments.
- **Effect**: It stacks the new fragment on top of the current fragment(s), which means multiple fragments can exist in the fragment back stack.
- **Back Stack**: The new fragment is placed in the **back stack** (if you call addToBackStack()), so the user can navigate back to the previous fragment.

replace() Method:

fragment.

● Back Stack: The replaced fragment can be added to the back stack if desired, so users can navigate back to the previous fragment.

When to Use:

● **Use `add()`** when you want to stack multiple fragments on top of each other, such as in situations like navigating between different sections in the app, where you want to keep the previous fragments in memory (e.g., a navigation flow).

● **Use `replace()`** when you need to switch out fragments completely, like showing a different screen or content in a container (e.g., replacing a "home" screen with a "details" screen).

Handler And Coroutines

Handler is a class used to manage and process messages or runnable tasks in a specific thread, typically the main thread or UI thread. It allows you to schedule tasks to run on a particular thread, often for communication between threads, especially when interacting with the user interface.

```
val handler = Handler(Looper.getMainLooper())

handler.post {

    // Code to run on the UI thread

    // Update UI here

}
```

Coroutines

1. **Asynchronous Programming:** Coroutines simplify asynchronous programming, allowing you to perform tasks like network requests, database operations, or heavy computations in the background without blocking the main thread (UI thread).

2. **Lightweight**: Unlike traditional threads, coroutines are lightweight and can handle many concurrent

Android interview Questions

requests or data processing don't freeze or block the user interface.

4. **Sequential Code for Async Tasks**: With coroutines, you can write asynchronous code in a sequential manner, avoiding the complexity of nested callbacks (callback hell).

5. **suspend Functions:** Coroutines use suspending functions (functions marked with suspend keyword) that can pause and resume execution without blocking threads. This allows tasks like network calls to run asynchronously.

6. **Coroutine Scopes:** Coroutines are launched within scopes, which manage their lifecycle. In Android, common scopes include GlobalScope for background tasks, lifecycleScope for activities/fragments, and viewModelScope for ViewModels.

7. **launch and async Builders**:

   - **launch:** Starts a coroutine that doesn't return a result but returns a job.
   - **async:** Starts a coroutine that returns a result (wrapped in a Deferred object).

8. **Error Handling**: Coroutines provide structured concurrency, so exceptions in coroutines are easier to manage and propagate, especially with try-catch blocks.

9. **Main Thread Updates**: Coroutines make it easy to update the UI from a background thread using withContext(Dispatchers.Main) or simply using lifecycleScope.

10. **Integration with Retrofit and Room**: Coroutines work seamlessly with libraries like Retrofit (for network requests) and Room (for database operations) in Android, making these tasks simpler and more efficient.

Example in Android (UI updates with background work):

kotlin

```
// Inside an Activity or Fragment

lifecycleScope.launch {

    // Running network call in background

    val data = fetchDataFromNetwork()

    // Update UI after the background work is completed

    updateUI(data)
```

Android interview Questions

```
// Suspending function to simulate a network call

suspend fun fetchDataFromNetwork(): String {

    delay(2000)  // Simulating delay

    return "Fetched data"

}
```

In this example:

- **lifecycleScope.launch** ensures that the coroutine will be canceled automatically when the activity is destroyed.
- **fetchDataFromNetwork()** is a suspending function that simulates a network delay.

1. What is Kotlin?

Kotlin is a modern, statically typed programming language developed by JetBrains. It runs on the Java Virtual Machine (JVM) and is fully interoperable with Java. It's officially supported for Android development.

A **statically typed programming language** is one where **type checking is done at compile time**, meaning **variable types must be known before the program runs.** This contrasts with dynamically typed languages, where type checking occurs at runtime.

2. What are the main features of Kotlin?

- Null safety
- Interoperability with Java
- Concise syntax
- Extension functions
- Smart casts
- Coroutines for asynchronous programming
- Functional programming support
- Type inference

- val = read-only (immutable) reference. Can't be reassigned.Resolved or evaluated at Runtime
- var = mutable reference. Can be reassigned.
- Const val PI=3.14 Resolved or evaluated at Compile time , it directly embedded into bytecode which makes faster and memory efficient

```
val name = "Rahul"  // immutable

var age = 25            // mutable

Const val PI=3.14  //immutable
```

4. What is a nullable type in Kotlin?

A nullable type can hold null. Declared using ?.

```
var name: String? = null
```

5. How do you handle null safety in Kotlin?

Using:

- Safe call ?.
- Elvis operator ?:
- !! (not recommended)
- if (variable != null)
- let block

6. What is an Elvis operator (?:) in Kotlin?

Used to provide a default value if the left expression is null.

```
val name: String? = null

val result = name ?: "Default"
```

7. What is a safe call operator (?.) in Kotlin?

Android interview Questions

```
val length = name?.length
```

8. What is the difference between == and === in Kotlin?

- == checks **structural equality** (like .equals() in Java)
- === checks **referential equality** (same memory reference)

```
fun main(){

    data class User(val name: String)

val u1 = User("Rahul")

val u2 = User("Rahul")

println(u1 == u2)    // ✅ true – data class auto-overrides equals

println(u1 === u2)   // false  in case of object



  val list1 = listOf(1, 2, 3)

val list2 = listOf(1, 2, 3)

println(list1 == list2)   // ✅ true – content equal

println(list1 === list2)      // false  in case of object

//*************************************************************************//

val x = 100

val y = 100
```

Android interview Questions

println(x === y) // true ✅ same reference (small ints are cached)

val s1 = "Kotlin"

val s2 = "Kotlin"

println(s1 == s2) // true ✅ value match

println(s1 === s2) // true (due to string interning) or primitive types

}

9. What are Kotlin data types?

- Basic: Int, Double, Boolean, Char, String
- Reference types: Any, Unit, Nothing, nullable types

10. What are smart casts in Kotlin?

Kotlin automatically casts a variable to the correct type after a type check.

if (obj is String) {

  println(obj.length) } // smart cast to String

11. How do you define a function in Kotlin?

fun add(a: Int, b: Int): Int {

  return a + b

}

12. What are default and named arguments?

- **Default arguments:** Provide default values to parameters.

Android interview Questions

fun greet(name: String = "User") { println("Hello, $name") }

greet() // Hello, User

greet(name = "Tom") // Hello, Tom

13. What is a lambda expression in Kotlin?

An anonymous function used to pass functionality.

val sum = { a: Int, b: Int -> a + b }

println(sum(2, 3)) // 5

---

14. What is the use of the when expression?

Acts like a switch statement.

val x = 2

when (x) {

1 -> println("One")

2 -> println("Two")

else -> println("Other")

}

15. What is a range in Kotlin?

Defines a range of values.

for (i in 1..5) {

```
}
```

16. How do you create an array in Kotlin?

```
 val arr = arrayOf(1, 2, 3)
```

17. What is a string template in Kotlin?

Allows embedding variables inside strings.

```
val name = "Kotlin"

println("Hello, $name")
```

18. Difference between listOf, mutableListOf, and arrayListOf?

- listOf: Immutable list
- mutableListOf: Mutable list
- arrayListOf: Backed by Java's ArrayList

19. How do you handle exceptions in Kotlin?

Using try-catch-finally

```
try {
    val result = 10 / 0
} catch (e: ArithmeticException) {
    println("Divide by zero")
} finally {
    println("Always executed")
}
```

Android interview Questions

Similar to void in Java; represents no return value.

```
fun printName(): Unit {

    println("Kotlin")

}
```

21. What is the difference between Any, Unit, and Nothing?

- Any: Supertype of all types (like Object in Java)

```
fun printValue(value: Any) {
    println(value)
}
printValue(10)          // ✅ Int
printValue("Hello")     // ✅ String
printValue(3.14)        // ✅ Double
```

📌 Use Any when you want to accept any type of object, similar to polymorphism.

- Unit: Function returns nothing meaningful

✅ 2. `Unit` – Represents "No Meaningful Value"

- Returned by functions that **do not return anything**.
- Like `void` in Java, but it's a **real type**.
- You can assign it and pass it as a value (rare but possible).

```
fun sayHello(): Unit {
    println("Hello!")
}
```

Android interview Questions

```
fun sayHello() {  // Unit return type is implicit
        println("Hello!")
    }
```

📌 Use `Unit` for functions that are meant to **perform actions** but **not return anything.**

- Nothing: Represents no value at all (e.g., function throws exception)

```
val value = when {
   x > 0 -> "Positive"
   x < 0 -> "Negative"
   else -> throw Exception("Unexpected")   // Nothing type here
}
```

🔍 What's happening?

This when expression must return a value for all branches (since it's used in val value =....)

Kotlin needs to infer a common type for all branches.

The first two branches return String

The else branch throws an exception, so it never returns — Kotlin treats that expression as type Nothing.

22. What is a type alias in Kotlin?

Gives an alternative name to a type.

```
typealias UserMap = Map<String, User>
```

23. What is a sealed class?

Android interview Questions

This gives the compiler full knowledge of all possible subclasses, which enables:

- **Exhaustive `when` expressions**
- **Better type safety**
- Similar to an `enum`, but more powerful (can hold data, logic)

```
sealed class Result

data class Success(val data: String) : Result()

object Error : Result()
```

`Success` and `Error` are the **only allowed** subclasses of `Result`

---

2. Why are sealed classes useful?

They allow for **exhaustive `when` expressions**, enable **type safety**, and reduce bugs by ensuring all possible subtypes are known at compile time.

"Exhaustive" means **all possible cases** are **explicitly handled** in a `when` expression**—no case is left out.**

Android interview Questions

4. Can a sealed class be abstract or have abstract members?

Yes, sealed classes can be **abstract** and have **abstract functions.**

```
sealed class Shape {

    abstract fun area(): Double

}
```

---

5. What restrictions apply to subclassing a sealed class?

All direct subclasses must be declared **in the same Kotlin file** as the sealed class itself**.** Otherwise, compilation fails**.**

---

6. How does Kotlin ensure exhaustive `when` statements with sealed classes?

The compiler **knows all subclasses** of a sealed class. If a `when` expression is not exhaustive, the compiler will throw a warning or error—unless an `else` is added**.**

7. Write a Kotlin sealed class representing a network state.

```
sealed class NetworkState {

    object Loading : NetworkState()

    data class Success(val data: String) : NetworkState()

    data class Error(val message: String) : NetworkState()
```

Android interview Questions

8. Can a sealed class have a constructor with parameters?

Yes. Like regular classes, sealed classes can have constructors.

```
sealed class Event(val time: Long)
```

9. Can a sealed class be generic? Provide an example.

Yes.

```
sealed class Result<T> {

    data class Success<T>(val data: T): Result<T>()

    data class Error<T>(val error: Throwable): Result<T>()

}
```

10. What happens if you define a subclass in another file?

It **won't compile.** Subclasses of a sealed class must be defined in the **same Kotlin file**, though not necessarily the same class or scope

11. Can sealed classes be nested?

Yes. You can nest them inside other classes or even inside other sealed classes.

```
class Wrapper {

    sealed class State {

        object Idle : State()

        object Loading : State()

    }
```

Android interview Questions

12. Is it mandatory to mark subclasses as `data` or `object`?

No. Subclasses can be `data`, `object`, or regular classes—depending on what fits the use case.

13. What is the difference between `sealed class` and `sealed interface`?

- `sealed class` can hold state (fields), have constructors.
- `sealed interface` allows **multiple inheritance**, which is useful in **composition-based designs**.

```
sealed interface UIState
```

14. How do sealed classes help avoid `else` in `when` expressions?

Since the compiler knows all subclasses, you don't need `else` if all cases are covered.

```
when (state) {
    is State.Loading -> ...
    is State.Success -> ...
    is State.Error -> ...
} // no 'else' needed
```

---

16. Can sealed classes be used to model UI state in Jetpack Compose?

Yes, very commonly:

```
sealed class UIState {
    object Loading : UIState()
```

```
    data class Error(val message: String) : UIState()
}
```

Then in Compose:

```
when (state) {
    is UIState.Loading -> CircularProgressIndicator()
    is UIState.Success -> ShowItems(state.data)
    is UIState.Error -> ShowError(state.message)
}
```

---

17. What's the performance impact of using sealed classes vs enums?

- **Enums** are more lightweight and optimized by the JVM.
- **Sealed classes** offer flexibility and type safety but may have slightly more overhead.
- Use enums for constants, sealed classes for complex hierarchies with data.

---

18. How would you model a Result wrapper using a sealed class?

```
sealed class Result<out T> {
```

```
    data class Failure(val throwable: Throwable) : Result<Nothing>()

    object Loading : Result<Nothing>()

}
```

---

19. Explain how sealed classes support type safety in state management.

By restricting possible states, sealed classes ensure you handle **only valid**, **known** states at compile time—reducing runtime bugs and null checks.

---

20. Can you use sealed classes with coroutines or flows?

Yes. You can emit sealed class instances from a **Flow** to represent different states.

```
fun fetchData(): Flow<Result<String>> = flow {

    emit(Result.Loading)

    try {

        val response = apiCall()

        emit(Result.Success(response))

    } catch (e: Exception) {

        emit(Result.Failure(e))

    }
```

Android interview Questions

1. ✅ `inline`

When you mark a function as `inline`, the compiler **copies** the code of the function and its lambda argument **directly into the call site.** This avoids **function object creation** and **runtime overhead** of lambda calls.

💡 **Why use it?**

- Improves performance
- Reduces memory allocation

💉 **Example:**

```
inline fun doTwice(action: () -> Unit) {
    println("Before")
    action()
    action()
    println("After")
}
```

💉 **Usage:**

```
fun main() {
    doTwice {
        println("Running action")
    }
```

📄 **Output:**

```
Before

Running action

Running action

After
```

---

2. 🚫 `noinline`

Used inside an `inline` function to **exclude a specific lambda** from being inlined.

💡 Why?

Sometimes you need to **pass a lambda around**, store it, or return it from a function—those can't be inlined.

💉 Example:

```
inline fun doSomething(normal: () -> Unit, noinline delayed: () -> Unit) {

    normal()           // gets inlined

    val later = delayed // can't be inlined

}
```

Android interview Questions

Used to prevent **non-local returns** in an inline lambda.

💡 **Why?**

If a lambda is **inlined** and may be called from a **different context (e.g., inside a thread)**, you can't allow it to return from the outer function.

🍼 **Example:**

```
inline fun runAction(crossinline action: () -> Unit) {
    val thread = Thread {
        action() // crossinline required here
    }
    thread.start()
}
```

If you **don't use `crossinline`**, Kotlin will warn you about illegal non-local return.

2. What happens if you don't use `inline` for a higher-order function?

Answer: Kotlin creates a function object (closure) for the lambda, which uses more memory and has performance overhead.

5. Can you return from an `inline` lambda? What about a `noinline` one?

Answer:

- Yes, you can return from an **inlined lambda** (non-local return).
- You **cannot** return from a `noinline` lambda because it's not inlined.

Android interview Questions

```
inline fun test(
    action: () -> Unit,
    noinline log: () -> Unit,
    crossinline callback: () -> Unit
) {
    action()        // can return early
    log()           // cannot return early or be inlined
    Thread {
        callback()  // must use crossinline
    }.start()
}
```

✅ What is a Higher-Order Function in Kotlin?

A Higher-Order Function is a function that either:

1. **Takes one or more functions as parameters, or**
2. **Returns a function as its result.**

👉 **In Kotlin, functions are first-class citizens, meaning you can pass them around just like variables.**

🔹 Syntax Example:

```
fun operate(a: Int, b: Int, operation: (Int, Int) -> Int): Int {
```

```
}
```

💉 **Usage:**

```
fun add(x: Int, y: Int) = x + y



fun main() {
    val result = operate(10, 5, ::add)
    println(result)  // Output: 15
}
```

🔹 **Lambda with Higher-Order Function:**

```
fun greet(message: String, action: (String) -> Unit) {
    action(message)
}
fun main() {
    greet("Hello Kotlin") { msg ->
        println("Greeting: $msg")
    }
}
```

Android interview Questions

Benefit	Explanation
✅ Code reusability	Define logic once and pass different behaviors
✅ Clean and readable code	Reduces boilerplate; better for functional-style programming
✅ Custom control flow	You can build DSLs or control structures like `repeat`, `run`, etc.
✅ Useful for Callbacks	Ideal for handling Android click listeners, async tasks, etc.
✅ Used in Kotlin collections	Functions like `map`, `filter`, `forEach`, `reduce` are higher-order

💡 Real-life Analogy:

Imagine a chef (function) that doesn't cook a specific dish, but takes a recipe (another function) and prepares food.
That chef is a higher-order function — it relies on behavior passed to it.

25. How to extend a class with extension functions?

Add new functions to existing classes without modifying their source.An **extension function** adds **new behavior** to a class **without**:

- modifying its source code

Android interview Questions

```
fun String.capitalizeFirst(): String {

    return this.replaceFirstChar { it.uppercase() }

}

println("kotlin".capitalizeFirst()) // Kotlin
```

2. What are the main features of Kotlin?

- **Null Safety:** Eliminates the null pointer exception (NPE) problem.
- **Extension Functions:** Adds new functionality to existing classes.
- **Type Inference:** Reduces verbosity by inferring types.
- **Coroutines:** For asynchronous programming without callbacks.
- **Smart Casts:** Automatic casting after type checks.
- **Data Classes:** Auto-generates equals, hashCode, toString, etc.
- **Sealed Classes:** Restricted class hierarchies useful in when expressions.
- **Interoperability:** Calls Java code and vice versa..

5. How do you handle null safety in Kotlin?

- **Safe Call (?.)**: Executes code only if the object is not null.
- **Elvis Operator (?:)**: Provides a default value if the object is null.
- **Non-null Assertion (!!)**: Forcefully unwraps a nullable variable (may throw NPE).
- **Safe Cast (as?)**: Casts safely and returns null if cast fails.

7. What is a safe call operator (?.) in Kotlin?

Used to access a property or call a method only when the variable is not null.

```
val name: String? = "Kotlin"

val length = name?.length  // returns
```

Android interview Questions

1. How is OOP implemented in Kotlin?

Kotlin supports all major OOP principles

- **Class & Object**
- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Encapsulation**

Kotlin enhances these with features like data class, sealed class, and extension functions.

2. What is a class and how do you declare it?

A class is a blueprint for creating objects.

class Person(val name: String, var age: Int)

Use class keyword. Kotlin supports primary & secondary constructors.

3. What is the primary constructor?

A **primary constructor** is a **concise and direct way** to declare and initialize properties of a class **in the class header.**

```
class User(val name: String, val age: Int)
```

- `val` or `var` in the constructor means Kotlin will automatically **create class properties.**
- You can also use an `init` block to add logic.

```
class User(val name: String, val age: Int) {

    init {

        println("User created: $name, $age")

    }
```

Android interview Questions

```
val u = User("Rahul", 25)

// Output: User created: Rahul, 25
```

📌 **Key Points:**

- Automatically generates constructor.
- Ideal for classes where all properties are known at creation time.
- Supports default values:

```
class Book(val title: String, val price: Double = 0.0)
```

4. What is a secondary constructor?

A **secondary constructor** is an **additional constructor** defined using the `constructor` keyword, allowing **different ways** to create an object.

```
class User {
    var name: String
    constructor(name: String) {
        this.name = name
    }
}
```

Yes! ✅ In Kotlin, when a class has both a **primary constructor** and **secondary constructors**, every **secondary constructor must explicitly or indirectly call the primary constructor** using `this(...)`.

This is because Kotlin enforces a **single entry point** for object initialization. The **primary constructor must always be invoked** to ensure all base properties are properly initialized.

✅ Example – Secondary Constructor Calling Primary

```
class User(val name: String) {

    var age: Int = 0



    constructor(name: String, age: Int) : this(name) {

        this.age = age

    }

}
```

🔍 What's happening?

- `val name: String` is part of the **primary constructor**
- The secondary constructor `constructor(name, age)` calls the **primary constructor** using `: this(name)`
- Then adds extra initialization logic (`this.age = age`)

❗ What if You Don’t Call the Primary Constructor?

You’ll get a **compiler error**:

```
Secondary constructor must delegate to the primary constructor
```

```
Secondary Constructor
    ↓ (must call)
Primary Constructor
    ↓
Object initialized
```

5. What is an init block?

Code inside init runs **immediately after the primary constructor.**

```
class Person(val name: String) {
   init {
        println("Person created: $name")
   }
}
```

6. What is inheritance in Kotlin?

Inheritance allows a class to acquire properties and methods from another class.

```
open class Animal {
   fun eat() {}
}
class Dog : Animal()
```

Android interview Questions

1.single inheritance 2.multilevel inheritance 3.Hierarchical inheritance

7. What is an open class?

```
open class Vehicle

class Car : Vehicle()
```

8. What is the use of the override keyword?

Used to override methods or properties from a superclass or interface.

```
open class Animal {
    open fun sound() {}
}



class Dog : Animal() {
    override fun sound() = println("Bark")
}
```

9. What is an abstract class in Kotlin?

An `abstract class` in Kotlin:

- **Cannot be instantiated** directly.
- Can contain both:
  - **Abstract members** (no implementation)
  - **Concrete members** (with implementation)

```
abstract class Shape {
    abstract fun draw()
}
```

Must be inherited and abstract members must be implemented.

10. What is an interface in Kotlin?

Interface defines a contract — can contain **abstract methods** and **default implementations**.

```
interface Clickable {
    fun click()
    fun show() = println("Showing clickable")
}
```

---

4. What is encapsulation in Kotlin?

Answer: Encapsulation means **hiding internal data** and only exposing necessary information through **getters/setters** or functions.

```
class BankAccount {
    private var balance: Int = 0
```

```
    fun getBalance(): Int = balance

}
```

---

6. What is polymorphism in Kotlin?

Answer: Polymorphism allows objects to take many forms. Kotlin supports:

- **Method overriding**
- **Interface implementation**
  interfaces can have default implementations.
- kotlin
-

```
open class Shape {

    open fun draw() = println("Drawing shape")

}

class Circle : Shape() {

    override fun draw() = println("Drawing circle")

}
```

Answer: Abstraction hides internal implementation and exposes only essential features. Achieved using **abstract classes** or **interfaces**.

```
abstract class Vehicle {

    abstract fun start()

}
```

---

10. What is an interface in Kotlin? Can it have default methods?

Answer: Interfaces define contracts. Yes,

```
interface Clickable {

    fun click()

    fun show() = println("Showing clickable")

}
```

---

11. What is constructor overloading in Kotlin?

Answer: Kotlin allows multiple constructors using **primary** and **secondary constructors**.

```
class Person(val name: String) {

    constructor(name: String, age: Int) : this(name)
```

Android interview Questions

---

12. Does Kotlin support multiple inheritance?

Answer: No, Kotlin doesn't support multiple class inheritance, but **supports multiple interfaces.**

```
interface A

interface B

class C : A, B
```

17. What is the difference between abstract class and interface?

Feature	Abstract Class	Interface
State allowed	✅ Yes	❌ (until Kotlin 1.4+, now partially allowed)
Multiple inheritance	❌ No	✅ Yes
Constructors	✅ Yes	❌ No

Android interview Questions

Answer: Yes. Kotlin classes and methods are **final by default**—you must use `open` to allow inheritance.

20. What is delegation in Kotlin (using `by` keyword)?

Answer: Kotlin supports **delegation pattern** natively using `by`.

```
interface Logger {

    fun log(message: String)

}

class ConsoleLogger : Logger {

    override fun log(message: String) = println(message)

}


class Service(logger: Logger) : Logger by logger
```

Feature	`abstract class`	`interface`
Can be instantiated	❌ No	❌ No
Can have abstract methods	✅ Yes	✅ Yes
Can have method bodies	✅ Yes	✅ Yes (default methods)
Can hold state (fields/vars)	✅ Yes	✅ Yes (only from Kotlin 1.1+, with `val` and default getters)
Constructors allowed?	✅ Yes	❌ No

Android interview Questions

Primary use	Shared **state + behavior** (template)	Define a **contract** (API behavior)

🔐 Encapsulation vs Abstraction

Feature	Encapsulation	Abstraction
What it means	Hiding internal **data** using access control	Hiding internal **implementation** details
Focus	**How** data is protected	**What** to expose and hide from the user
Purpose	To **restrict access** and protect data integrity	To **simplify complexity** by exposing only essentials
Achieved using	Access modifiers: `private`, `protected`, `public`	Abstract classes, interfaces
Example	Private fields with public getters/setters	Interface with method declarations only
Real-world analogy	Medical capsule (you don't see what's inside)	Car steering wheel (you don't know how it works internally)

🧠 Simple Kotlin Example

✅ Encapsulation

```
class BankAccount {
    private var balance: Int = 0  // internal data is hidden

    fun deposit(amount: Int) {
        if (amount > 0) balance += amount
    }

    fun getBalance(): Int = balance
}
```

- **Balance** is hidden from direct access.
- Controlled via methods.

✅ Abstraction

```
abstract class Vehicle {
```

Android interview Questions

```
class Car : Vehicle() {
    override fun startEngine() = println("Engine started")
}
```

- User interacts with `Vehicle`, not caring how `startEngine()` is implemented.
- **Encapsulation** = Hiding **data** (security + control)
- **Abstraction** = Hiding **implementation** (simplicity + usability)

12. What is object declaration in Kotlin?

Used to create a **singleton** object directly.

```
object NetworkManager {
    fun connect() = println("Connecting...")
}
```

---

13. What is a companion object?

A **companion object** is Kotlin's way of defining **static-like members** inside a class.

- It's an **object that belongs to the class**, not to instances.
- Members inside a companion object can be accessed **without creating an object of the class**.

```
class Utils {
    companion object {
        fun printHello() = println("Hello")
```

```
}

Utils.printHello()
```

What is a Singleton Class in Android (Kotlin)?

A **Singleton** is a design pattern that ensures a **class has only one instance** throughout the app and provides a **global point of access** to it.

🧠 Why Use Singleton?

- To **share data or functionality** across the entire app.
- To **avoid multiple instances** that can waste memory or cause bugs.
- Commonly used for **logging**, **network clients**, **database helpers**, etc.

🔸 Singleton in Kotlin (Simple Example)

```
object MySingleton {
    var count = 0

    fun increment() {
        count++
    }
}
```

💉 Usage:

```
MySingleton.increment()
println(MySingleton.count)  // Output: 1
```

`object` in Kotlin automatically makes the class a thread-safe singleton — no need to write boilerplate code like in Java.

Android interview Questions

🔹 **Real-world Use Case:**

✅ **Example: Singleton Retrofit Client**

```
object ApiClient {
    val retrofit: Retrofit = Retrofit.Builder()
        .baseUrl("https://api.example.com/")
        .addConverterFactory(GsonConverterFactory.create())
        .build()
}
```

💉 **Usage:**

```
val api = ApiClient.retrofit.create(MyApiService::class.java)
```

✅ Features of Kotlin Singleton (`object`):

- Thread-safe
- Lazy-initialized
- Global access
- Only one instance created in the app

🏁 Summary

Feature	Description
Ensures single instance	Only one object created throughout app
Global access	Can access from anywhere
Memory efficient	Avoids creating multiple instances
Kotlin keyword	`object` is used for Singleton in Kotlin

---

14. What is the Singleton pattern in Kotlin?

Android interview Questions

access point to it.

```
object Logger {
    fun log(message: String) {
        println("Log: $message")
    }
}
```

🔍 Usage:

```
Logger.log("Hello World")  // Output: Log: Hello World
```

- No need to create an instance (`Logger()`).
- Only one instance is created — automatically thread-safe and lazy-initialized.

---

15. What is a data class?

Used to hold data. Automatically generates:

- toString()
- equals(), hashCode()
- copy(), componentN()

```
data class User(val name: String, val age: Int)
```

We can't inherits data class why — By default, a `data class` in Kotlin is `final`, meaning **not open for inheritance**:

Android interview Questions

No. Data classes are **implicitly final** (no subclassing). This is to preserve immutability and avoid ambiguity in generated methods.

---

17. What is a nested class and inner class?

- **Nested class**: Does not hold reference to outer class.
- **Inner class**: Holds reference to outer class.

```
class Outer {

    private val x = 10

    class Nested {

        fun foo() = 2

    }

    inner class Inner {

        fun foo() = x  // can access Outer.x

    }

}
```

---

18. What is visibility modifier in Kotlin?

Modifier	Meaning	Where it's visible
`public`	**Default. Accessible everywhere**	**Any file, class, module**

Android interview Questions

`protected`	**Visible to subclasses (only inside class hierarchy)**	**Same class + subclasses only**
`private`	**Visible only inside the same class/file**	**Very restricted**

🔷 2. Application Lifecycle in Android

✅ What is the `Application` class?

The `Application` class is a **base class for maintaining global application state**. It runs **before any activity, service, or receiver** is created and **remains alive** as long as the app is running.

It's used for:

- **Initializing libraries** (e.g., Hilt, Firebase, Timber)
- **Global configurations**
- **Tracking app-wide state**

🧠 Key Lifecycle Methods:

Method	When it's called
`onCreate()`	Called **once** when the app process is created
`onTerminate()`	Only called in **emulators** (never in real devices)

✅ Kotlin Example:

```
class MyApp : Application() {
    override fun onCreate() {
```

Android interview Questions

```
        // ✅ Initialize global dependencies/libraries
        // Example:
        // FirebaseApp.initializeApp(this)
        // Timber.plant(Timber.DebugTree())
        // start Koin / Dagger / Hilt DI framework here
    }
}
```

📌 How to Register the Application Class

To make this class run, **register it** in your `AndroidManifest.xml`:

xml
CopyEdit

```
<application
    android:name=".MyApp"
    android:label="MyApp"
    android:icon="@mipmap/ic_launcher">

    <!-- activities, services, etc -->
</application>
```

🧪 Use Cases of `Application` class:

- Setting up:
  - Dependency Injection (Hilt, Koin, Dagger)
  - Analytics tools (Firebase, Google Analytics)
  - Logging tools (Timber)
- Storing global state (not recommended for large data)
- Initializing database (e.g., Room or Realm)

⚠️ `onTerminate()` — Why it doesn't work in production?

- It's **only called in emulators or debug mode** when you stop the app.

✅ SOLID Design Principles Explained (with Kotlin examples)

1. S – Single Responsibility Principle (SRP)

> **A class should have only one reason to change.**

💡 **Meaning:**
A class should do only **one job**—no mixing of responsibilities like handling UI + logic + data access in one class.

```
class InvoicePrinter {
    fun print(invoice: Invoice) = println(invoice.details)
}

class InvoiceSaver {
    fun saveToDb(invoice: Invoice) { /* save logic */ }
}
```

☑ Good: Separate classes handle different responsibilities.

---

2. O – Open/Closed Principle (OCP)

> **Software entities should be open for extension, but closed for modification.**

💡 **Meaning:**
You should be able to add new behavior **without changing existing code.**

```
interface Shape {
    fun area(): Double
}

class Circle(val radius: Double): Shape {
    override fun area() = Math.PI * radius * radius
}

class Square(val side: Double): Shape {
    override fun area() = side * side
}

fun printArea(shape: Shape) = println(shape.area())
```

Android interview Questions

3. L – Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

💡 **Meaning:**
You should be able to use a subclass **anywhere** you use its superclass—**without breaking the program.**

```
open class Bird {
    open fun fly() = println("Flying...")
}

class Sparrow: Bird() // ✅ OK
class Ostrich: Bird() {
    override fun fly() = throw UnsupportedOperationException("Ostriches can't
fly!")
}
```

🚫 Violates LSP: `Ostrich` is a `Bird` but can't fly.

✅ **Fix using proper abstraction:**

kotlin
CopyEdit

```
interface Bird
interface FlyingBird : Bird {
    fun fly()
}

class Sparrow: FlyingBird {
    override fun fly() = println("Flying")
}

class Ostrich: Bird // Doesn't fly, and no violation now
```

---

4. I – Interface Segregation Principle (ISP)

Clients should not be forced to depend on methods they do not use.

💡 **Meaning:**
Keep interfaces **small and focused.**

🚫 **Bad:**

```
interface Machine {
```

Android interview Questions

```
}

class SimplePrinter : Machine {
    override fun print() = println("Printing...")
    override fun scan() = throw NotImplementedError()
    override fun fax() = throw NotImplementedError()
}
```

✅ **Good:**

kotlin
CopyEdit

```
interface Printer {
    fun print()
}

interface Scanner {
    fun scan()
}
```

Now classes implement only what they need.

---

5. D – Dependency Inversion Principle (DIP)

> **Depend on abstractions, not on concrete implementations.**

💡 **Meaning:**
High-level modules should not depend on low-level ones—**both should depend on interfaces.**

🔧 **Kotlin Example:**

```
interface Keyboard {
    fun input(): String
}

class USBKeyboard : Keyboard {
    override fun input() = "User input"
}

class Computer(private val keyboard: Keyboard) {
    fun readInput() = println(keyboard.input())
}
```

☑ `Computer` doesn't care **which** keyboard is used—it depends on the abstraction (`Keyboard`), not a specific class.

Android interview Questions

Principle	Goal
SRP	One job per class
OCP	Extend without modifying
LSP	Subclasses should behave like base classes
ISP	Prefer small, focused interfaces
DIP	Depend on abstractions, not concretes

1. What is a value class in Kotlin?

Answer:
A **value class** wraps a single value and avoids object allocation at runtime when possible.
Declared using the `@JvmInline` annotation:

```
@JvmInline

value class UserId(val id: String)
```

---

2. What is the purpose of using value classes?

Answer:

- Reduce **memory allocation**
- Improve **performance**
- Add **type safety** (e.g., distinguish `UserId` from `String`)

---

3. What are the requirements of a value class in Kotlin?

Answer:

- Must have exactly **one property** in the primary constructor.

- The property must be a `val` (immutable).

---

4. How does a value class differ from a regular class?

Answer:

Feature	Value Class	Regular Class
Overhead	Low / Inlined	Full object allocation
Constructor	One property only	Multiple allowed
Use case	Type-safe wrappers	General-purpose logic

---

5. Can value classes have functions?

Answer:
Yes. You can define member functions and override methods like `toString()` or `equals()`.

```
@JvmInline
value class Email(val value: String) {
    fun isValid() = value.contains("@")
}
```

---

6. Can you use a value class as a generic type parameter?

Answer:
Yes, but be cautious. The compiler may **box** the value class when used generically, which can reduce performance benefits.

---

7. Can value classes inherit or be inherited from?

Answer:
No. **Value classes cannot inherit** other classes or be extended—they are **final by design**.

Answer:

- Cannot have init blocks.
- Cannot have mutable properties (`var`).
- Cannot inherit from other classes or be open/abstract.
- May be **boxed** in generics, reflection, or nullable types.

---

9. What's the difference between inline and value classes in Kotlin?

Answer:

- Before Kotlin 1.5: used `inline class`.
- From Kotlin 1.5+: use `@JvmInline value class`.

Same concept, just updated syntax.

19. What is a value class (inline class)?

A **value class** is a **special Kotlin class** that wraps a **single property**, typically a primitive or value type, and is **inlined at compile-time** to avoid object allocation.

```
@JvmInline

value class UserId(val id: String)
```

- `@JvmInline` is required for JVM interop.
- Must have **exactly one property** in the **primary constructor**.
- The compiler may **replace** the class instance with the **underlying type** during runtime (to improve performance).

Android interview Questions

To **wrap primitives or simple values** with **type safety**, **without runtime overhead.**

20. What are Kotlin enums and their use cases?

Enums represent a **fixed set of constants.**

```
enum class Direction {
    NORTH, SOUTH, EAST, WEST
}
```

Can contain properties and functions too.

21. What is the difference between lateinit and lazy?

Feature	lateinit	lazy
For var/val	Only var	Only val
Null safety	You must initialize before use	Automatically initialized on first use
Use case	Mutable properties (e.g., View)	Expensive, read-only properties

23. What are Kotlin scopes: this, super, inner?

- this: Refers to current class.
- super: Calls parent class's method/property.
- inner: Used inside inner classes to access outer class.

```
inner class Inner {
    fun getOuter() = this@Outer.name
```

Android interview Questions

24. What is delegation in Kotlin?

One class delegates functionality to another class.

```
interface Printer {
    fun print()
}
class ActualPrinter : Printer {
    override fun print() = println("Printing...")
}
class DelegatedPrinter(p: Printer) : Printer by p
```

What's Happening Here?

- `ActualPrinter` does the real work (`print()` method).
- `DelegatedPrinter` **implements the** `Printer` **interface**, but **delegates all function calls** to the instance `p`.
- So `DelegatedPrinter.print()` will internally call `ActualPrinter.print()`

25. What is the purpose of @JvmStatic, @JvmOverloads, etc.?

Annotation	Purpose
@JvmStatic	Makes function static when called from Java
@JvmOverloads	Generates Java-compatible overloads for default arguments

Android interview Questions

✅ `static` in Java – Complete Explanation

In Java, the `static` keyword is used for **members (variables or methods) that belong to the class**, rather than to instances of the class. This means you can access them **without creating an object.**

✅ Use `@JvmStatic` when:

- **You want Java interoperability.**
- **You want a Kotlin function to look and act like a static method in Java.**

51. What are scope functions in Kotlin?

Functions like let, run, apply, also, and with that execute a block of code **in the context of an object.**

---

52. Difference between let, run, with, apply, and also

Function	Object Reference	Return Value	Use Case
let	it	Lambda result	Null checks, chaining
run	this	Lambda result	Object configuration + return result
with	this	Lambda result	When you already have the object
apply	this	The object	Initialize or configure an object

Android interview Questions

---

53. When to use apply vs also?

- apply: When you're **configuring the object.**
- also: When you're doing **side operations** like logging or validation.

---

54. What does let return?

It returns the result of the lambda block.

```
val result = "Kotlin".let {
    it.uppercase()
} // result = "KOTLIN"
```

---

55. How is run used for null checking?

Use run with ?. to safely execute code if the object is not null.

```
val name: String? = "JetBrains"
name?.run {
    println(length)
}
```

57. Best use case for with in Kotlin?

When you already have an object and want to operate on it without extension.

```
val sb = StringBuilder()
```

```
    append("Hello, ")

    append("World!")

}
```

---

59. Is there a performance difference between scope functions?

No major performance difference. All are **inline functions**, so they get compiled efficiently. The choice is based on readability and context.

---

60. What are common mistakes with scope functions?

- Overusing them leading to **nested/unreadable code**
- Confusing return types (this vs result)
- Misusing also or let when apply is more appropriate
- Null safety assumptions when chaining

📝 Kotlin Coroutines Interview Questions

---

3. What is suspend keyword in Kotlin?

A suspend function is a function that **can be paused and resumed** without blocking the thread.

suspend fun fetchData() {

- delay(1000)}

4. What is CoroutineScope?

```
CoroutineScope(Dispatchers.IO).launch {

    // coroutine code

}
```

---

5. What are launch and async in coroutines?

Kotlin Coroutines, the function `launch {}` does not return a result — instead, it returns a `Job` object.

Function	Returns	Use case
launch	Job (no result)	Fire-and-forget operations
async	Deferred<T> (result)	When you need a result

```
launch { ... }          // doesn't return a result// But returns a job object

val job = CoroutineScope(Dispatchers.IO).launch {

    // Some background task

    println("Running in coroutine")

}



val result = async { 10 }.await() // returns 10
```

---

6. What is delay() in coroutines?

delay(1000) // suspends coroutine for 1 second

7. What is the difference between delay() and Thread.sleep()?

Feature	delay()	Thread.sleep()
Blocking	Non-blocking	Blocking
Coroutine-safe	Yes	No
Suspension	Yes	No (pauses thread instead)

8. What is Dispatchers in Kotlin?

Dispatchers define **where and how** coroutines run.

launch(Dispatchers.IO) { ... } // background

launch(Dispatchers.Main) { ... } // UI thread

- Main: UI thread (Android)
- IO: Disk/network I/O
- Default: CPU-intensive
- Unconfined: Inherits caller's thread

it **starts execution in the thread where the coroutine was launched —** typically, the **caller's thread —** and **continues in whatever thread the suspension resumes it on.**

Dispatcher	Description	Use Case Example
`Main`	UI/Main thread	Updating views in Android
`IO`	Optimized for blocking I/O	Retrofit calls, Room DB, file I/O
`Default`	Optimized for CPU work	Sorting, parsing, encryption

Android interview Questions

9. What is GlobalScope?

A coroutine scope that is **not bound** to any job or lifecycle.

GlobalScope.launch { ... }

Not recommended in Android — use viewModelScope, lifecycleScope, or custom CoroutineScope.

---

10. What is Job in coroutines?

1. **Tracks Coroutine Status**
A `Job` helps track whether a coroutine is active, completed, or cancelled.

2. **Cancellation**
You can cancel a coroutine by calling `job.cancel()`.

3. **Parent-Child Relationship**
Jobs form a hierarchy, so cancelling a parent job cancels all its children automatically.

A Job is a **coroutine handle** that represents its **lifecycle —** such as **running**, **completed**,

Used to:

- Cancel coroutines
- Check completion
- Chain coroutines

🔧 Core Methods of Job

Method / Property	Description
`start()`	Starts the job if it's in the `New` state and was created lazily.

Android interview Questions

`cancel(CancellationException)`	Cancels with a specific reason.
`join()`	Suspends until the job is complete.
`invokeOnCompletion()`	Registers a callback to be called once the job completes or is canceled.
`getCancellationException()`	Returns the exception that caused the job's cancellation.

📍 Job State Properties

Property	Description
`isActive`	`true` if the job is still active (not completed or cancelled).
`isCompleted`	`true` if the job has finished (successfully or not).
`isCancelled`	`true` if the job was cancelled.

🧬 Example:

```
val job = GlobalScope.launch {
    delay(1000)
    println("Coroutine finished")
}

println(job.isActive)    // true
job.cancel()             // Cancels the job
println(job.isCancelled) // true
```

Deferred is a coroutine that **returns a result** (like a future or promise).

val result: Deferred<Int> = async { 10 }

println(result.await()) // prints 10

12. How do you cancel a coroutine?

You can cancel using job.cancel() or use withTimeout / withTimeoutOrNull.

val job = launch {

delay(5000)

}

job.cancel()

✅ 1. `job.cancel()` – Manual cancellation

You use `job.cancel()` when **you want full control** over when and how a coroutine should be cancelled.

Example:

kotlin

CopyEdit

```
val job = CoroutineScope(Dispatchers.Default).launch {
    repeat(1000) {
        delay(500)
        println("Running $it")
```

```
}


// Cancel after some time

delay(2000)

job.cancel()
```

Pros:

- Full manual control
- Can cancel from outside the coroutine
- Useful for long-lived or externally controlled tasks

Cons:

- You must remember to cancel
- You must handle cleanup (e.g., using `try/finally` or `invokeOnCompletion`)

⏱️ 2. `withTimeout` / `withTimeoutOrNull` – Automatic cancellation

`withTimeout(timeMillis)` throws `TimeoutCancellationException`

`withTimeoutOrNull(timeMillis)` returns `null` instead of throwing

Example – `withTimeout`:

kotlin

Android interview Questions

```
try {
    withTimeout(2000) {
        repeat(1000) {
            delay(500)
            println("Running $it")
        }
    }
} catch (e: TimeoutCancellationException) {
    println("Cancelled due to timeout")
}
```

Example – `withTimeoutOrNull`:

kotlin

CopyEdit

```
val result = withTimeoutOrNull(2000) {
    repeat(1000) {
        delay(500)
        println("Running $it")
    }
```

```
}

println(result) // null if timed out
```

Pros:

- Built-in timeout logic
- Cleaner for one-shot or short-lived operations
- `withTimeoutOrNull` avoids exception handling

Cons:

- Not suitable for long-lived tasks
- Limited control once started

13. What is withContext() in Kotlin?

Switches the coroutine to a different Dispatcher.

```
withContext(Dispatchers.IO) {
    // long-running IO task
}
```

It's a **suspend function**, used to offload work to another thread.

---

14. What is structured concurrency?

Android interview Questions

Structured Concurrency is a design principle in Kotlin coroutines where **child coroutines are bound to the lifecycle of a parent**, ensuring:

✅ No coroutine is left running in the background unintentionally.
✅ Easy cancellation and error handling.
✅ Safer and more maintainable asynchronous code.

```
CoroutineScope(Dispatchers.Main).launch {
    val job = launch {
        // child coroutine
    }
    // all children are cancelled when parent scope is cancelled
}
```

---

15. What is coroutineContext?

It holds information like Job, Dispatcher, CoroutineName, etc., used by the coroutine.

```
coroutineContext[Job]
```

---

16. What are viewModelScope and lifecycleScope?

Provided by AndroidX for automatic lifecycle-aware coroutine management.

- viewModelScope: Cancels coroutines when ViewModel is cleared.
- lifecycleScope: Cancels coroutines based on lifecycle state (onDestroy, etc.)

Android interview Questions

Using:

- try-catch inside coroutine
- CoroutineExceptionHandler for top-level
- supervisorScope to isolate child failures

Because coroutines can be **asynchronous** and **concurrent**, exceptions don't always behave like they do in synchronous code. For example:

- Exceptions in child coroutines may cancel the parent.
- `launch {}` and `async {}` behave differently.
- `try-catch` doesn't always catch coroutine exceptions at the top level.

✅ 1. `try-catch` Inside a Coroutine Block

```
fun main() = runBlocking {

    launch {

        try {

            throw RuntimeException("Boom!")

        } catch (e: Exception) {

            println("Caught in try-catch: $e")

        }

    }
```

🧠 This works well when **you're inside the coroutine.**

✅ 2. CoroutineExceptionHandler (For `launch {}`)

Used when:

- You want to catch **uncaught exceptions** in **top-level** or `launch` coroutines.

```
val handler = CoroutineExceptionHandler { _, throwable ->
    println("Caught by handler: $throwable")
}
fun main() = runBlocking {
    val job = launch(handler) {
        throw RuntimeException("Handled by CoroutineExceptionHandler")
    }
    job.join()
}
```

🧠 Works only for `launch {}` — not for `async {}` (because `async` stores exception in `Deferred`).

❗ 3. `async {}` and `Deferred`

With `async`, exceptions are not thrown until you call `await()`.

```
val deferred = async {
    throw Exception("Failure in async")
```

```
try {
    deferred.await()  // Exception is thrown here
} catch (e: Exception) {
    println("Caught async exception: $e")
}
```

🧠 `CoroutineExceptionHandler` **won't catch** this. You must wrap `await()` in `try-catch`.

✅ 4. `supervisorScope` — Isolate Failures Between Child Coroutines

Used when:

- You want to **run multiple coroutines independently**, so that **one failing coroutine doesn't cancel others.**

```
fun main() = runBlocking {
    supervisorScope {
        val job1 = launch {
            delay(100)
            throw RuntimeException("Job 1 failed")
        }
```

Android interview Questions

```
            delay(500)
            println("Job 2 completed")
        }

        job1.invokeOnCompletion { println("Job1 completed: ${it?.message}") }
        job2.join()
    }
}
```

🧠 `supervisorScope` helps keep your app resilient — ideal for multiple UI calls where one failure shouldn't stop others

---

18. What is supervisorScope?

Unlike coroutineScope, it **isolates child failures.**

```
supervisorScope {
   launch { throw Exception("Fail") }
   launch { println("Still running") }
}
```

Useful when one child should not affect others.

Android interview Questions

Scope	Failure Propagation
coroutineScope	Cancels all children on one failure
supervisorScope	Other children continue on failure

20. What are best practices with coroutines?

✅ Use lifecycle-aware scopes (viewModelScope, lifecycleScope)
✅ Always cancel unused coroutines
✅ Use withContext for switching threads
✅ Use supervisorScope for fault tolerance
✅ Avoid GlobalScope in Android

1. What are coroutines in Kotlin?

Coroutines are **lightweight, suspendable tasks** that allow writing **asynchronous, non-blocking** code in a sequential way.

They are built on top of Kotlin's suspend functions and use **structured concurrency** to manage execution safely.

Think of them as lightweight threads with better lifecycle and memory handling.

---

2. Difference between Thread and Coroutine

Aspect	Thread	Coroutine
Weight	Heavy (1MB stack per thread)	Lightweight (few KBs)
Creation cost	Expensive	Cheap
Switching	Context switch (slow)	No context switch (fast)

Android interview Questions

Lifecycle	Managed by OS	Managed by CoroutineScope

---

3. What is a suspend function?

A suspend function can **pause** its execution without blocking the thread and **resume** later.

```
suspend fun fetchData(): String {
    delay(1000)
    return "Result"
}
```

Can only be called from another suspend function or coroutine.

---

4. What is CoroutineScope?

It defines the **lifecycle** of coroutines and manages their execution. Every coroutine runs within a scope.

```
val scope = CoroutineScope(Dispatchers.IO)
scope.launch { ... }
```

Android-specific scopes:

- viewModelScope
- lifecycleScope

6. How to launch a coroutine?

Use launch inside a CoroutineScope.

```
    // background work
}
```

In Android:

```
viewModelScope.launch {
    // safe within ViewModel
```

---

11. What is GlobalScope?

A global CoroutineScope that **lives for the entire app**. Not tied to any lifecycle.

```
GlobalScope.launch {
    // runs forever unless cancelled
}
```

❗ Avoid in Android apps — use viewModelScope, lifecycleScope, or CoroutineScope.

What is LiveData?

- `LiveData` is a **lifecycle-aware**, observable data holder.
- It only sends updates to active lifecycle owners (e.g., an Activity or Fragment).

Android interview Questions

Types of LiveData in Android

Type	Description	Used For
LiveData<T>	Read-only LiveData. Cannot be modified directly.	Exposing data to UI components from ViewModel safely.
MutableLiveData<T>	Subclass of LiveData. You can modify its value using `setValue()` or `postValue()`.	Updating data from ViewModel or Repository.
MediatorLiveData<T>	Observes multiple LiveData sources and reacts to their changes.	Merging or combining multiple LiveData sources.
Transformations.map()	Creates a new LiveData by transforming the data from another LiveData.	Converting one data type to another (e.g., Int → String).
Transformations.switchMap()	Switches to a new LiveData source based on changes in original LiveData.	Useful for dependent or nested LiveData (e.g., loading data based on user input).

1. LiveData<T> (Read-Only)

Android interview Questions

- Used to expose data.
- Cannot be changed directly.

🔹 2. MutableLiveData<T>

kotlin

CopyEdit

```
val _count = MutableLiveData<Int>()

_count.value = 10
```

- Can be modified with `setValue()` (main thread) or `postValue()` (background thread).
- Typically private in ViewModel.

🔹 3. MediatorLiveData<T>

kotlin

CopyEdit

```
val mediator = MediatorLiveData<String>()

mediator.addSource(liveData1) { value -> /* react */ }
```

Android interview Questions

- Observes multiple LiveData and updates accordingly.
- Used for combining multiple sources.

🔹 4. Transformations.map()

kotlin

CopyEdit

```
val nameLength: LiveData<Int> = Transformations.map(nameLiveData) { name ->
    name.length
}
```

- Maps one LiveData to another with a new value.

🔹 5. Transformations.switchMap()

```
val userId = MutableLiveData<String>()
val userData: LiveData<User> = Transformations.switchMap(userId) { id ->
    repository.getUserById(id)
}
```

Android interview Questions

🔄 Kotlin Flow & Reactive Streams

---

13. What is a Flow in Kotlin?

A **cold asynchronous stream** that emits multiple values sequentially over time.

> `Flow` is a **cold**, **asynchronous**, and **reactive stream** that emits **multiple sequential values** over time.

🔧 Think of it like:

- A **suspendable sequence**
- A **lightweight alternative** to RxJava's `Observable`
- Supports **backpressure**, cancellation, and **operators**

```
fun numbers(): Flow<Int> = flow {
    emit(1)
    emit(2)
    emit(3)
}


fun main() = runBlocking {
    numbers().collect { value ->
```

Android interview Questions

```
    }

}
```

Only starts collecting when collect() is called.

Feature	Flow	LiveData
Lifecycle-aware	❌ Not by default (✅ with `lifecycleScope`)	✅ Yes, auto cancels when lifecycle ends
Thread control	✅ Full control with `Dispatchers`	❌ Limited (usually runs on main thread)
Emits multiple values	✅ Emits streams of values	✅ Can emit, but less flexible
Cold / Hot	❄️ Cold: starts only when collected	🔥 Hot: always active once observed
Cancellation	Manual (e.g., `collect {}` is suspendable)	Automatic via lifecycle
Error Handling	✅ With `catch`, `onCompletion`	❌ No built-in try/catch
Use in ViewModel	Great for background data / repository	Great for UI-bound state
State management	Can use `StateFlow` / `SharedFlow` for state	Built-in observable state
Backpressure	✅ Yes	❌ No

Use `LiveData` when you need **lifecycle-aware**, simple UI-bound data.
Use `Flow` when you need **powerful, coroutine-based streams**, especially in **repository**, **data** or **background** layers.

✅ 1. `StateFlow`

A **state holder** that emits the current and new values. It's **hot**, lifecycle-friendly, and retains the **last known value.**

```
val _state = MutableStateFlow(0)

val state: StateFlow<Int> = _state
```

🔄 Emit New Value

```
_state.value = 1
```

🔁 Collect It

```
state.collect {

    println("State: $it")

}
```

✅ Use when:

- Holding **UI state**
- You want **initial/default value**
- Only latest value matters (like a `LiveData` replacement)

SharedFlow

A **shared event broadcaster** that **does not hold state.** Useful for one-time events like navigation, toasts, etc.

🔧 Declaration

```
val _event = MutableSharedFlow<String>()
```

Android interview Questions

📤 Emit Event

```
viewModelScope.launch {

    _event.emit("Show Toast")

}
```

🔁 Collect Event

```
lifecycleScope.launch {

    viewModel.event.collect {

        Toast.makeText(context, it, Toast.LENGTH_SHORT).show()

    }

}
```

✅ Use when:

- Emitting **one-time events** (e.g., errors, navigation)
- Supporting **multiple collectors**
- You don’t need to store the value

Android interview Questions

Converts a **cold flow** (like `flow {}` or `map {}`) into a **hot `StateFlow`**, keeping the last emitted value.

🔧 Syntax

```
val coldFlow = flowOf(1, 2, 3)



val hotState: StateFlow<Int> = coldFlow

    .stateIn(

        scope = viewModelScope,

        started = SharingStarted.WhileSubscribed(),

        initialValue = 0

    )
```

✅ Use when:

- You want to **convert a cold flow** (like repository data)
- You want to **retain latest state** in ViewModel/UI

✅ Difference between Internal Storage and External Storage in Android

Feature	Internal Storage	External Storage

	internal memory	
🔒 **Access Permission**	No permission needed	Requires `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE` (before Android 10)
👀 **Visibility to Other Apps**	**Private** to the app	**Public** (unless in app-specific directory)
🧼 **Auto-deleted on uninstall**	✅ Yes	❌ No (unless in app-specific external directory)
🧠 **Security**	More secure (sandboxed)	Less secure (visible to file managers or other apps)
📦 **Best Use Case**	Store private files: user settings, cache, DBs	Store large files: images, videos, documents
🔐 Scoped Storage (Android 10+)	Still fully accessible	Access is limited unless using MediaStore / SAF

⚙️ SHARED PREFERENCES – Interview Questions

14. What are SharedPreferences in Android?

A key-value storage mechanism for saving simple app data like settings or flags.

```
val prefs = getSharedPreferences("settings", Context.MODE_PRIVATE)

prefs.edit().putString("name", "John").apply()
```

16. How do you read data from SharedPreferences?

```
val name = prefs.getString("name", "default")
```

17. What is the difference between `apply()` and `commit()`?

`apply()` is async and faster. `commit()` is synchronous and returns success status.

18. Where are SharedPreferences stored?

In an XML file under:
`/data/data/<package>/shared_prefs/`

19. Are SharedPreferences secure?

Not by default. You should use EncryptedSharedPreferences for sensitive data.

20. Can you store complex objects in SharedPreferences?

No. Only primitives, `String`, and `Set<String>`. For objects, use JSON or serialization.

📦 CONTENT PROVIDER – Interview Questions

21. What is a Content Provider?

It manages access to structured data and allows sharing data across apps.

When you need to share app data (DB, files) with other apps securely.

23. What methods must a ContentProvider override?

- `onCreate()`
- `query()`
- `insert()`
- `update()`
- `delete()`
- `getType()`

24. What is a Content URI?

A URI that identifies a specific resource in the provider.
Format: `content://<authority>/<table>/<id>`

---

25. What is a ContentResolver?

A client API that apps use to interact with content providers.

`ContentResolver` is a class in Android that **provides access to data from content providers**. It acts as a **middleman** between your app and the content provider, allowing you to perform operations like:

- Querying data
- Inserting data
- Updating data

Android interview Questions

🔧 Why is ContentResolver Used?

Because apps do **not access a content provider's data directly**, the `ContentResolver` helps your app **communicate** with it using **URIs (Uniform Resource Identifiers).**

26. How do you register a Content Provider?

In `AndroidManifest.xml`:

```
<provider
    android:name=".MyProvider"
    android:authorities="com.example.provider"
    android:exported="true" />
```

27. What is a UriMatcher used for?

To route incoming URIs to the correct database operation (like insert/query/delete).

28. How do you perform a query using a ContentResolver?

```
val cursor = contentResolver.query(uri, null, null, null, null)
```

29. How do you notify data change in a ContentProvider?

```
context.contentResolver.notifyChange(uri, null)
```

It allows secure data sharing between apps by exposing a standardized interface via URIs.

✅ Bonus Questions

31. How do you choose between internal vs external storage?

Use internal for private, secure data. Use external for large or shared files (e.g., media).

32. How do you make SharedPreferences secure?

Use **EncryptedSharedPreferences** provided by Jetpack Security.

33. Can a ContentProvider access SharedPreferences?

Technically yes, but it's uncommon. They're separate storage systems.

✅ What is `SharedPreferences` in Kotlin + Jetpack Compose?

`SharedPreferences` is an Android API used to **store small key-value pairs persistently** — such as user settings, flags, or simple state like login status.

In Jetpack Compose, you access `SharedPreferences` the same way using **Kotlin**, but from inside a `@Composable` using `LocalContext`.

✅ When to Use

- Saving a **Boolean flag** (e.g., isLoggedIn)
- Remembering **user-selected theme**
- Storing **username, language, etc.**

- **Step 1: Save Data**

```
val context = LocalContext.current

val sharedPref = context.getSharedPreferences("MyPrefs",
Context.MODE_PRIVATE)



Button(onClick = {

    sharedPref.edit()

        .putString("username", "Rahul Kumar")

        .putBoolean("isLoggedIn", true)

        .apply()

}) {

    Text("Save to SharedPreferences")

}
```

- **Step 2: Read Data**

```
val context = LocalContext.current

val sharedPref = context.getSharedPreferences("MyPrefs",
Context.MODE_PRIVATE)

val username = sharedPref.getString("username", "Guest")

val isLoggedIn = sharedPref.getBoolean("isLoggedIn", false)
```

Android interview Questions

🧠 How it Works Internally

- Stores data in a private **XML file** in internal storage.
- Automatically persists across app restarts.
- Ideal for small, lightweight data.

🚫 Limitations

- Not recommended for large or complex data.
- Not lifecycle-aware or observable.
- Blocking I/O if `commit()` is used.

✅ Data Types Supported

Type	Method
:ring	`putString(key, value)`
t	`putInt(key, value)`

oat	`putFloat(key, value)`
ong	`putLong(key, value)`
et<String>	`putStringSet(key, value)`

⚠️ Modern Alternative: Jetpack DataStore (Recommended)

- Type-safe
- Coroutine-based (asynchronous)
- Replaces SharedPreferences in new apps

📦 What is Gradle?

Gradle is a powerful build automation tool used in Android for:

- Compiling code
- Managing dependencies
- Building APKs/AABs

- Automating tasks (e.g., ProGuard, code generation)

It uses a **Groovy** or **Kotlin DSL (Domain Specific Language)** in `build.gradle` files.

🔧 General DI Concepts

1. What is Dependency Injection (DI)? Why is it useful?

Answer:
DI is a design pattern where a class receives its dependencies from an external source instead of creating them itself. It promotes decoupling, testability, and maintainability.

2. What are the types of DI?

Answer:

- **Constructor Injection**
- **Field Injection**
- **Method Injection**

3. What are the benefits of using DI in Android?

Answer:

- Decouples code
- Promotes testability
- Easier lifecycle management
- Cleaner architecture

Answer:

- **Service Locator**: Class looks up its dependencies (pull).
- **DI**: Dependencies are passed into the class (push).

💉 Hilt-Specific Questions

---

5. What is Hilt in Android?

Answer:
Hilt is a DI library built on top of Dagger, officially recommended by Google. It simplifies DI in Android by providing standard components and annotations.

6. What are key annotations used in Hilt?

Answer:

- `@HiltAndroidApp`
- `@AndroidEntryPoint`
- `@Inject`
- `@Module` and `@InstallIn`
- `@Provides`
- Scopes like `@Singleton`, `@ActivityScoped`

Android interview Questions

Answer:
It triggers Hilt's code generation, including a base class for your application that Hilt can use to manage the DI container.

---

8. What does `@AndroidEntryPoint` do?

Answer:
It marks Android classes (Activity, Fragment, etc.) as DI entry points where Hilt will inject dependencies.

---

9. What is the difference between `@Inject` and `@Provides`?

Answer:

- `@Inject`: Used on constructors (automatic binding)
- `@Provides`: Used in a module to provide a dependency manually

---

10. What is a Hilt module?

Answer:
A class annotated with `@Module` and `@InstallIn(...)` that defines how to provide dependencies (via `@Provides` or `@Binds`).

---

11. What are scopes in Hilt?

Answer:
They define the lifetime of dependencies:

- `@Singleton`: Application-wide

- `@ViewModelScoped`: One per ViewModel
- `@FragmentScoped`: One per fragment

---

12. How is ViewModel injected using Hilt?

```
@HiltViewModel

class MyViewModel @Inject constructor(private val repo: MyRepo) :
ViewModel()
```

In Fragment/Activity:

```
val viewModel: MyViewModel by viewModels()
```

---

13. Can you inject interfaces using Hilt?

Answer:
Yes, using `@Binds` inside a `@Module`:

```
@Binds

abstract fun bindRepo(impl: MyRepoImpl): MyRepo
```

Android interview Questions

14. What is Koin?

Answer:
Koin is a lightweight Kotlin-based DI framework that uses a DSL for defining dependencies. No annotation processing is needed.

---

15. How do you declare and inject a dependency in Koin?

```
val appModule = module {

    single { MyRepository() }

}


val repo: MyRepository by inject()
```

---

16. What are `single`, `factory`, and `scoped` in Koin?

Answer:

- `single`: Creates a singleton
- `factory`: Creates a new instance each time

17. How does Koin compare to Hilt?

Feature	Hilt	Koin
Based On	Dagger (Java)	Kotlin DSL
Codegen	Yes (KAPT)	No
Compile-time safety	Yes	No
Learning Curve	Higher	Lower

✋ Manual DI Questions

18. What is manual dependency injection?

Answer:
Providing dependencies manually through constructors or setters without any DI framework.

19. When should you use manual DI?

Answer:
In small projects, or when you want full control without adding a third-party framework.

Android interview Questions

Answer:

- Boilerplate increases with app complexity
- Hard to manage object graph
- No lifecycle management or scopes

Refied type: unables you to access the type of info at runtime that should have been erased by JVM during code compilation ,which only work with inline .

Ex:

```
Inline fun <refied T> getType():String{
Return T::class.java.name
{
fun main(){
print(getType<String>)
}
```

`Log.d()` vs `Log.e()`

```
val userId = 123
Log.d("MainActivity", "User ID is $userId")
```

```
try {
        val result = 10 / 0
    } catch (e: ArithmeticException) {
        Log.e("MainActivity", "Division by zero", e)
    }
```

✅ 1. `lazy` – Lazy Initialization (Immutable)

```
val someValue: String by lazy {
    println("Computed!")
    "Hello Lazy"}
```

✅ Key Points:

- Used with `val` (i.e., **read-only**)
- Initialization happens **once** – when the value is **first accessed**
- Thread-safe by default (`LazyThreadSafetyMode.SYNCHRONIZED`)
- Good for **heavy computations or expensive objects**

```
val config: Config by lazy {

    loadConfigFromDisk() // loaded only on first access

}
```

✅ 2. `lateinit` – Late Initialization (Mutable)

```
lateinit var userName: String
```

✅ Key Points:

- Used only with **`var`**
- **Must be a non-null type** (`String`, not `String?`)
- Must be initialized **before use** (or you'll get an exception)
- Mostly used for **dependency injection** or when the value isn't available at object creation

✅ Example:

```
lateinit var viewModel: MyViewModel

fun onCreate() {

    viewModel = ViewModelProvider(this)[MyViewModel::class.java]

}
```

❓ *What will happen if you access a `lateinit` variable before it's initialized?*
✅ *It will throw `UninitializedPropertyAccessException` at runtime.*

Android interview Questions

➤ Definition:

An **Exception** is a condition that occurs during **program execution**, which can be **caught and handled.**

✅ Examples:

- `NullPointerException`
- `IllegalArgumentException`
- `IOException`
- `IndexOutOfBoundsException`

✅ Usage:

```
try {
    val result = 10 / 0
} catch (e: ArithmeticException) {
    println("Caught exception: ${e.message}")
}
```

✅ Key Features:

- **Recoverable —** app can usually continue
- Can be handled using `try-catch` blocks
- Can be custom-defined

Checked - IOexception,fileNotfoundExc,----------checked at compiletime

❌ 2. Error

➤ Definition:

An **Error** indicates a **serious issue** that **should not be caught** or handled — typically something wrong at the **JVM/system level.**

❗ Examples:

- `OutOfMemoryError`
- `StackOverflowError`
- `InternalError`
- `NoClassDefFoundError`

✅ Example:

```
fun recursive() {

    recursive() // causes StackOverflowError

}
```

⚠️ Key Features:

- **Not meant to be caught**
- Usually indicates a fatal problem
- Should not be handled in normal application logic

Android interview Questions

Feature	MVVM	MVI
Direction	Bi-directional	Unidirectional
State	Mutable	Immutable
Complexity	Easier for small apps	Better for complex flows
Debuggability	Medium	High (pure functions, state logs)
Testability	Good	Excellent
Tooling	Android Jetpack friendly	Needs custom intent/state design

✅ **What is Jetpack in Android?**

Jetpack is a set of Android libraries, tools, and architectural guidance provided by Google to help

Android interview Questions

Jetpack simplifies complex tasks like background work, navigation, data persistence, and UI using well-structured, backward-compatible components.

🚀 Key Benefits of Jetpack

- **Reduce boilerplate code**
- **Follow best practices (like MVVM)**
- **Backward compatibility**
- **Modular and testable**
- **Integrated with Kotlin and Jetpack Compose**

Android interview Questions

Category	Description	Key Components
1. Architecture	Manage app lifecycle, data, UI separation	`ViewModel`, `LiveData`, `StateFlow`, `Navigation`, `Room`, `DataStore`, `WorkManager`
2. UI	Build modern UI	`Jetpack Compose`, `Fragment`, `RecyclerView`, `ConstraintLayout`
3. Behavior	Handle app tasks & background work	`Permissions`, `Notifications`, `WorkManager`, `Sharing`
4. Foundation	Core system components	`AppCompat`, `Multidex`, `Android KTX`, `Test` libraries

🔧 **Jetpack Architecture Example (MVVM):**

- **ViewModel – Holds UI-related data**
- **LiveData / StateFlow – Observes data changes**
- **Room – Stores data locally**
- **Repository – Manages data sources (network, DB)**
- **Navigation – Manages screen transitions**

Android interview Questions

```
val users = liveData(Dispatchers.IO) {

    emit(userRepository.getAllUsers())

}
```

🎯 Popular Jetpack Libraries

Component	Purpose
Room	**SQLite DB access**
LiveData	**Lifecycle-aware observable data**
ViewModel	**Retains UI data on config changes**
Navigation	**Manages screen transitions**
WorkManager	**Background tasks that need guaranteed execution**
DataStore	**Replaces SharedPreferences**
Paging	**Load paginated data**
Hilt	**Dependency Injection**
Jetpack Compose	**Modern UI toolkit**

Android interview Questions

Jetpack Compose (New UI Toolkit)

Part of Jetpack – allows you to build UIs with Kotlin code declaratively.

```
@Composable

fun Greeting(name: String) {

    Text("Hello, $name!")

}
```

1. What is Jetpack Compose?

Jetpack Compose is Android's **modern toolkit** to build native UI **declaratively** using Kotlin.

```
@Composable
fun Greeting() {
    Text("Hello, Compose!")
}
```

2. What is a Composable function?

In **Jetpack Compose**, a **Composable function** is a **Kotlin function annotated with** `@Composable` that **describes part of the UI** in a declarative way.

```
@Composable
fun MyButton() {
    Button(onClick = { /* handle click */ }) {
        Text("Click Me")
    }
}
```

---

3. What do you mean by remember ?

`remember` is a **Compose memory utility** that stores a value **in memory** during recomposition so that it **doesn't get reset** every time the UI is recomposed.

Android interview Questions

🧠 Why do we need `remember`?

Without `remember`, any variable you declare **inside a Composable** would be **re-initialized** on every recomposition.

```
var count by remember { mutableStateOf(0) }
```

---

4. What is `mutableStateOf`?

`mutableStateOf()` creates an **observable state holder.** When its value changes, **Jetpack Compose automatically recomposes** the affected Composables.

```
val name = remember { mutableStateOf("Rahul") }
```

---

5. What is recomposition?

Recomposition is the process where Jetpack Compose **redraws parts of the UI** when **state changes.**

But it doesn't redraw the entire screen — only the parts of the UI that depend on the changed state.

```
@Composable
fun Counter() {
    var count by remember { mutableStateOf(0) }
    Button(onClick = { count++ }) {
        Text("Clicked $count times")
    }
}
```

---

7. What is Modifier?

A `Modifier` is a **declarative UI configuration object** used to:

- **Decorate** (like background, padding)
- **Layout** (like size, alignment)

- **Add graphics effects** (like shadow, border)

```
Text("Hello", modifier = Modifier.padding(16.dp))
```

---

9. What is `rememberSaveable`?

Or

`onSaveInstanceState()`

Preserves state across configuration changes.

`rememberSaveable` is like `remember`, **but smarter —** it **saves the value** in a **Bundle** and **restores it automatically** after configuration changes (like orientation changes or process recreation).

```
val username = rememberSaveable { mutableStateOf("") }
```

10. What is LazyColumn?

A list that lazily composes items.

```
LazyColumn {
    items(10) { index ->
        Text("Item $index")
    }
}
```

---

11. How to use ViewModel in Compose?

```
val viewModel: MyViewModel = viewModel()
val name by viewModel.name.collectAsState()
```

---

12. What is state hoisting?

State is lifted to the parent to improve reusability.

State hoisting means **moving (lifting)** the **state variable** from a **child composable to its parent**, and passing it down as parameters (`state` and `onEvent`) to the child.

```
@Composable
fun CounterApp() {
```

```
@Composable
fun CounterScreen() {
    var count by remember { mutableStateOf(0) }
    Counter(count = count, onIncrement = { count++ })
}

@Composable
fun Counter(count: Int, onIncrement: () -> Unit) {
    Button(onClick = onIncrement) {
        Text("Count: $count")
    }
}
```

You have 3 Composables:

- `CounterApp()` → calls `CounterScreen()`
- `CounterScreen()` → holds the state
- `Counter(count, onIncrement)` → stateless UI

---

13. What is Scaffold?

Scaffold provides basic screen layout with top bar, FAB, etc.

```
Scaffold(
    topBar = { TopAppBar(title = { Text("Title") }) },
    floatingActionButton = {
        FloatingActionButton(onClick = {}) {
            Icon(Icons.Default.Add, contentDescription = null)
        }
    }
) {
    Text("Scaffold Content")
}
```

---

14. What is LaunchedEffect?

`LaunchedEffect` is a **side-effect handler** used in Composables to **launch a coroutine tied to the composition.**

Android interview Questions

```
LaunchedEffect(key1) {
    // coroutine block
}
```

- `key1` determines **when** the coroutine should re-run.
- The block **automatically runs** when the key changes or Composable is recomposed for the first time.

```
@Composable
fun WelcomeScreen() {
    LaunchedEffect(Unit) {
        delay(2000)
        println("Welcome!")
    }
}
```

✅ **What happens:**

- Runs **only once** when `WelcomeScreen()` is composed.
- Delays for 2 seconds, then prints `"Welcome!"`

---

15. What is SideEffect?

SideEffect is a **Composable function** that lets you run **synchronous code after every successful recomposition – but only when the Composable is part of the composition**.

It's often used to:

- **Communicate with objects outside Compose** (like logging, updating analytics, or external state)

✅ **Simple Example:**

```
@Composable
fun MyComponent(name: String) {
    SideEffect {
        println("Recomposed for: $name")
```

Android interview Questions

🔁 Every time MyComponent() recomposes (e.g., if name changes), the message prints again.

How is `SideEffect` different from `LaunchedEffect`?

Feature	`SideEffect`	`LaunchedEffect`
pe	Synchronous (non-suspending)	Asynchronous (suspending coroutine)
ıns when?	After every recomposition	On first composition / key change
ɔod for?	Syncing non-Compose state, logging	Data loading, delay, animation, flows
ın use suspend?	❌ No	✅ Yes

16. What is DisposableEffect?

For setup/cleanup like listeners.

`DisposableEffect` is a **side-effect handler** used to **set up and clean up** resources **based on a key** or lifecycle event.

It's ideal for:

- Setting up **listeners, broadcast receivers, observers**
- Managing **resources or callbacks** tied to a Composable's lifecycle

```
DisposableEffect(Unit) {

    println("Start effect")
    onDispose {
        println("Cleanup")
    }
}
```

17. How to show dialog?

Android interview Questions

```
        confirmButton = {
            Button(onClick = { showDialog = false }) {
                Text("OK")
            }
        },
        title = { Text("Dialog") },
        text = { Text("Message here") }
    )
}
```

18. How to show Snackbar?

```
val scaffoldState = rememberScaffoldState()

LaunchedEffect(Unit) {
    scaffoldState.snackbarHostState.showSnackbar("Snackbar Message")
}
```

19. How to use Navigation?

```
val navController = rememberNavController()

NavHost(navController, startDestination = "home") {
    composable("home") { HomeScreen(navController) }
    composable("detail") { DetailScreen() }}
```

20. How to pass data in Navigation?

```
composable("detail/{id}") { backStackEntry ->
    val id = backStackEntry.arguments?.getString("id")
    Text("ID: $id")
}
```

---

21. What is hiltViewModel()?

Used with Hilt to inject ViewModel.

```
val vm: MyViewModel = hiltViewModel()
```

---

22. What is BoxWithConstraints?

Gives layout bounds inside Box.

```
BoxWithConstraints {
    if (maxWidth < 400.dp) {
        Text("Small screen")
    } else {
```

Android interview Questions

---

23. How to observe LiveData?

LiveData is a lifecycle-aware observable data holder used in MVVM architecture to expose data from the `ViewModel` to the UI.

In Jetpack Compose, we use **`observeAsState()`** to convert LiveData into a Compose **State** so that it can **automatically trigger recomposition** when the data changes.

✅ ViewModel:

```
class MyViewModel : ViewModel() {

    private val _name = MutableLiveData("Rahul")

    val name: LiveData<String> = _name

    fun updateName(newName: String) {

        _name.value = newName

    }

}
```

✅ Composable Screen:

```
@Composable

fun MyScreen(viewModel: MyViewModel = viewModel()) {
```

Android interview Questions

```
    Column(modifier = Modifier.padding(16.dp)) {
        Text(text = "Hello, $name")
        Button(onClick = { viewModel.updateName("Amit") }) {
            Text("Change Name")
        }
    }
}
```

🔁 What Happens:

1. `observeAsState()` converts `LiveData<String>` to `State<String?>`
2. When the name changes in `ViewModel`, Compose **recomposes** the UI
3. `Text` updates to show the latest name

24. Difference: fillMaxSize vs wrapContentSize

- `fillMaxSize`: expands to parent size
- `wrapContentSize`: wraps to child size

26. How to use themes?

Use `MaterialTheme` for color, typography.

```
Text("Styled", color = MaterialTheme.colorScheme.primary)
```

`rememberCoroutineScope()` provides a **`CoroutineScope`** that is **tied to the lifecycle of the Composable.**

This allows you to **launch coroutines** (e.g., for async tasks, animations, delays) **from Composables** safely.

```
val scope = rememberCoroutineScope()

scope.launch {

    // Coroutine code here

    delay(1000)

    println("Launched!")

}
```

📌 Why Use It?

You **cannot launch a coroutine directly** in a Composable unless it's inside a `LaunchedEffect`.

`rememberCoroutineScope()` gives you a **safe coroutine scope** for:

- Button click actions
- Snackbar calls
- Animations
- UI events that trigger suspending work

29. How to make scrollable Column?

Android interview Questions

---

31. How to use BottomNavigation?

```
BottomNavigation {
    BottomNavigationItem(
        selected = true,
        onClick = { },
        icon = { Icon(Icons.Default.Home, null) },
        label = { Text("Home") }
    )
}
```

---

32. How to make clickable Card?

```
Card(modifier = Modifier.clickable { }) {
    Text("Card Content")
}
```

33. TextField vs OutlinedTextField?

- `TextField`: filled box
- `OutlinedTextField`: bordered

var name by remember { mutableStateOf("") }

TextField(

value = name,

onValueChange = { name = it },

label = { Text("Name") },

modifier = Modifier.padding(16.dp)

)

➡️ Displays a filled box with a floating label.

Android interview Questions

📏 Example: OutlinedTextField

```
var name by remember { mutableStateOf("") }

OutlinedTextField(
    value = name,
    onValueChange = { name = it },
    label = { Text("Name") },
    modifier = Modifier.padding(16.dp)
)
```

➡️ Displays a bordered box with a floating label.

---

34. What is ConstraintLayout?

Compose's version of classic `ConstraintLayout`.

📘 **Code:**

```
ConstraintLayout(modifier = Modifier.fillMaxSize()) {
    val (text) = createRefs()
    Text("Hello", Modifier.constrainAs(text) {
        top.linkTo(parent.top)
        start.linkTo(parent.start)
    })
}
```

---

38. How to implement loading state?

```
if (loading) {
    CircularProgressIndicator()
```

Android interview Questions

41. What is `rememberUpdatedState()` in Compose?

`rememberUpdatedState` provides the **latest value** of a lambda or variable **within side-effects** like `LaunchedEffect`.

```
@Composable
fun Timer(onTimeout: () -> Unit) {
    val currentOnTimeout by rememberUpdatedState(onTimeout)

    LaunchedEffect(Unit) {
        delay(5000)
        currentOnTimeout() // always uses the latest callback
    }
}
```

---

42. How does recomposition work with lambdas?

Passing lambdas directly can trigger recomposition. To avoid this:

- Use `remember`
- Or extract the lambda outside

📘 **Example:**

```
val onClick = remember { { println("Clicked") } }
Button(onClick = onClick) { Text("Click") }
```

---

43. What are `keys` in LazyColumn used for?

To **preserve item state** across recompositions and reordering.

```
LazyColumn {
    items(items = myList, key = { it.id }) { item ->
        Text(item.name)
    }
}
```

Use the same `NavBackStackEntry` or pass ViewModel as parameter.

```
@Composable
fun ScreenA(viewModel: MyViewModel = viewModel()) {
    Text(viewModel.data)
}
```

---

45. How to create reusable custom components?

Wrap UI and logic inside a composable and use parameters.

📘 **Example**

```
@Composable
fun CustomButton(text: String, onClick: () -> Unit) {
    Button(onClick = onClick) {
        Text(text)
    }
}
```

---

46. What is derivedStateOf()?

It **optimizes recomposition** for computed states.

📘 **Example:**

```
val list = remember { mutableStateListOf(1, 2, 3) }
val count = derivedStateOf { list.count { it > 1 } }
```

47. What is CompositionLocal in Compose?

It allows data sharing (like context, theme) **without prop drilling.**

```
val LocalUser = compositionLocalOf { "Guest" }

@Composable
fun App() {
    CompositionLocalProvider(LocalUser provides "Rahul") {
        Greeting()
    }
}

@Composable
fun Greeting() {
    Text("Hello, ${LocalUser.current}")
```

Android interview Questions

49. How do you manage screen orientation changes?

Use `rememberSaveable` or `ViewModel`.

```
val text = rememberSaveable { mutableStateOf("") }
```

---

51. How to avoid recomposition for specific parts?

Use `key()` or split composables.

```
key(user.id) {
    Text("User: ${user.name}")
}
```

---

52. How to debounce or throttle input in Compose?

Use coroutines inside `LaunchedEffect`.

Debouncing is a technique that ensures:

- The function (e.g. search API) runs **only after the user stops typing** for a certain delay.
- Prevents **excessive function calls** on every keystroke.

📌 It's **perfect for search bars, form validation, and autocomplete.**

```
LaunchedEffect(query) {
    delay(300)
    search(query)
}
```

---

53. How do you manage back press in Compose?

Use `BackHandler`.

`BackHandler` is a Composable function that lets you **override the system back button** behavior **inside**

This is useful for:

- Preventing navigation from certain screens
- Confirming exit with dialogs
- Closing modals, bottom sheets, or drawers manually

```
BackHandler(enabled = true) {
    // Custom back press logic
}
```

---

54. How to detect screen size or orientation in Compose?

Use `BoxWithConstraints` or `Configuration`.

```
val config = LocalConfiguration.current
val isLandscape = config.orientation == Configuration.ORIENTATION_LANDSCAPE
```

---

56. Can you use XML in Compose or vice versa?

Yes, use `ComposeView` in XML or `AndroidView` in Compose.

```
<androidx.compose.ui.platform.ComposeView
    android:id="@+id/composeView"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
```

📘 **XML inside Compose:**

```
AndroidView(factory = { TextView(it).apply { text = "XML View" } })
```

57. How do you show a toast in Compose?

Use `LocalContext`.

```
val context = LocalContext.current
Button(onClick = {
    Toast.makeText(context, "Clicked", Toast.LENGTH_SHORT).show()
}) {
```

58. How do you manage dialogs and visibility?

Track visibility using `remember` state.

```
var showDialog by remember { mutableStateOf(false) }

if (showDialog) {
    AlertDialog(
        onDismissRequest = { showDialog = false },
        confirmButton = {
            Button(onClick = { showDialog = false }) {
                Text("OK")
            }
        },
        text = { Text("Are you sure?") }
    )
}
```

60. What are common mistakes to avoid in Compose?

- Holding state outside `remember`
- Triggering recomposition unintentionally
- Overusing `remember` for static data
- Creating ViewModels inside `@Composable`
- Updating UI directly inside `LaunchedEffect` without condition

61. How do you launch a coroutine in Compose?

```
Use LaunchedEffect or rememberCoroutineScope().



@Composable

fun AutoRunTask(query: String) {

    LaunchedEffect(query) {

        delay(300) // debounce-style delay
```

```
    }


}
```

```
val scope = rememberCoroutineScope()
Button(onClick = {
    scope.launch {
        delay(1000)
        println("Task Done")
    }
}) {
    Text("Run")
}
```

---

62. What is `LaunchedEffect(Unit)` used for?

Used to run a coroutine when the composition enters.

```
LaunchedEffect(Unit) {

    delay(2000)
    println("Launched once on composition")
}
```

Android interview Questions

🔹 Feature	✅ `rememberCoroutineScope`	✅ `LaunchedEffect`
Purpose	**Manually launch coroutine (e.g., onClick)**	**Auto-launch coroutine tied to state or lifecycle**
Scope lifecycle	**Tied to the Composable lifecycle**	**Tied to the Composable + key lifecycle**
Trigger behavior	**Doesn't re-run automatically**	**Re-runs automatically when key changes**
Recomposition behavior	**Survives recomposition**	**Re-launches if key changes**
Use case	**Button clicks, form submits, snackbar events**	**Initial loads, debounce, reacting to state change**
Coroutine cancellation	**Cancels when Composable leaves composition**	**Cancels and restarts if key changes**
Usage pattern	`scope.launch { ... }`	`LaunchedEffect(key) { ... }`
Best for	`User actions`	**One-time or reactive side effects**
Can use suspend?	✅ **Yes**	✅ **Yes**

Android interview Questions

To collect state from a coroutine and expose it to the UI.

`produceState` is a **Composable coroutine builder** that:

- Runs a coroutine in the background
- **Produces a State<T>** that can be **read in Compose UI**
- Perfect for **collecting data from suspend functions or Flows** and exposing it to Composables

```
@Composable
fun Timer(): State<Int> {
    return produceState(initialValue = 0) {
        while (true) {
            delay(1000)
            value++
        }
    }
}
```

---

65. How to collect Flow in Compose?

Use `collectAsState()` or `collectAsStateWithLifecycle()` (recommended with lifecycle-aware).

```
val count by viewModel.countFlow.collectAsState()
Text("Count: $count")
```

---

66. What is `snapshotFlow`?

Converts Compose state into a Flow.

```
val scrollState = rememberScrollState()

LaunchedEffect(Unit) {
    snapshotFlow { scrollState.value }
        .collect { println("Scrolled: $it") }
}
```

- `State<T>`: Read-only
- `MutableState<T>`: Read-write, triggers recomposition

```
var name by remember { mutableStateOf("Rahul") }
```

---

69. How to optimize performance in Compose UI?

✅ Tips:

- Use `remember` to cache values
- Use `key()` in lists
- Avoid unnecessary recomposition
- Split composables into small pieces
- Use `derivedStateOf` for computed state

---

73. How do you debounce text input in Compose?

Use coroutine + `LaunchedEffect`.

```
var query by remember { mutableStateOf("") }

LaunchedEffect(query) {
    delay(500)
    search(query)
}
```

---

76. What is `Modifier.graphicsLayer`?

It gives access to low-level drawing features (scale, alpha, rotation).

```
Image(
```

Android interview Questions

```
        scaleX = 0.5f
        alpha = 0.8f
    }
)
```

---

77. How to preserve scroll position in LazyColumn?

Use `rememberLazyListState()`.

t

```
val listState = rememberLazyListState()

LazyColumn(state = listState) {
    items(100) { Text("Item $it") }
}
```

---

78. What is `snapshotStateListOf()`?

Creates a state-aware list that triggers recomposition.

```
val items = remember { snapshotStateListOf("One", "Two") }
```

---

79. What is `NavigationDrawer` in Compose?

It's a Material3 component for side navigation.

```
ModalNavigationDrawer(
    drawerContent = {
        Text("Drawer Item")
    },
    content = {
        Text("Main content")
    }
)
```

---

80. How do you handle back navigation with Compose Navigation?

Use `navController.popBackStack()`.

```
IconButton(onClick = { navController.popBackStack() }) {
    Icon(Icons.Default.ArrowBack, contentDescription = null)
```

Android interview Questions

Answer: App Not Responding when UI thread is blocked (5 sec).

Avoid: Move heavy tasks to background—Use `AsyncTask`, Coroutine, or WorkManager .

Q: How do you add a fragment programmatically?

```
supportFragmentManager.beginTransaction()
    .replace(R.id.container, MyFragment())
    .commit()
```

✅ 16. What is the difference between `startActivity()` and `startActivityForResult()`?

Answer:

- `startActivity()`: No result expected.
- `startActivityForResult()`: Returns data from another activity.

21. What is an APK file?

Answer:
Android Package — the final compiled app ready for installation.

❌ Incorrect Kotlin Function Declaration

```
fun prime(val n: Int) { ... }
```

This is **not allowed in Kotlin**.
You **cannot** use `val` (or `var`) in a **function parameter**.

Difference between @Composable ,@Stable,@Immutable

✅ `@Composable`

🔹 Purpose:

Marks a **function** that emits **UI** in Jetpack Compose.

Android interview Questions

- Used to **build UI declaratively**.
- Called during **recomposition** when state changes.
- Cannot be called from non-@Composable functions.

📄 Example:

kotlin
CopyEdit

```
@Composable
fun Greeting(name: String) {
    Text("Hello, $name!")
}
```

✅ `@Stable`

🔹 Purpose:

Marks a **class or function** as *"stable"*, meaning:

- Its public properties **do not change unexpectedly**.
- If any property changes, recomposition should happen.

🧠 Use When:

- You have a custom class (e.g., `data class`) with **mutable properties**.
- You want Compose to **optimize recomposition** and **avoid unnecessary UI updates**.

📄 Example:

kotlin
CopyEdit

```
@Stable
class UiState(var count: Int)
```

⚠️ If any property inside the class changes but the object itself doesn't, Compose will still recompose only if it's marked `@Stable`.

Android interview Questions

🔹 Purpose:

Marks a class as **completely immutable** — no internal state can change after it's created.

🧠 Use When:

- You have a **data class or model** where fields are **all `val`** and **never change**.
- Compose can **skip recomposition entirely** when the object stays the same.

🗒️ Example:

kotlin
CopyEdit

```
@Immutable
data class User(val id: Int, val name: String)
```

Compose assumes this class will never change. So, it **skips recomposition** unless the object reference changes.

Q: What are destructuring declarations in Kotlin?
✅ A: A feature that allows unpacking objects into multiple variables using `componentN()` functions — commonly used with data classes, maps, pairs, etc.

Destructuring is made possible using **`componentN()`** functions.
Kotlin automatically generates these for **data classes**.

🔸 Example with Data Class:

```
data class Person(val name: String, val age: Int)

val person = Person("Rahul", 25)

// Destructuring
val (name, age) = person

println(name) // Output: Rahul
println(age)  // Output: 25
```

Behind the scenes, Kotlin calls:

Android interview Questions

🔸 Example with Map Entry:

kotlin
CopyEdit

```
val map = mapOf("A" to 1, "B" to 2)

for ((key, value) in map) {
    println("$key = $value")
}
```

✅ What is `Context` in Android?

In Android, `Context` is an **abstract class** that provides **access to application-specific resources and classes**, and is essential for operations like:

- Accessing system services
- Launching activities
- Accessing databases or SharedPreferences
- Accessing resources (strings, drawables, etc.)

🔹 Think of Context as:

> "**The handle to the global information about the application environment.**"

🔷 1. Application Context

- **Scope:** Entire application lifecycle
- **Accessed by:** `applicationContext`
- **Use it for:**
  - Singleton classes

Android interview Questions

- Background tasks
- Accessing resources globally

- ⚠️ **Limitation:** Cannot be used to show dialogs or launch UI components (Activities)

kotlin
CopyEdit

```kotlin
val appContext = applicationContext
Toast.makeText(appContext, "Hello App", Toast.LENGTH_SHORT).show()
```

🔷 2. Activity Context

- **Scope:** Activity lifecycle
- **Accessed by:** `this` inside an `Activity`
- **Use it for:**
  - Inflating views
  - Showing dialogs
  - Launching new Activities
  - Accessing UI components
- ⚠️ **Warning:** Don't leak this context into static or long-lived classes (e.g., ViewModels)

kotlin
CopyEdit

```kotlin
val intent = Intent(this, SecondActivity::class.java)
startActivity(intent)
```

🔷 3. Service Context

- **Scope:** Service lifecycle

Android interview Questions

- **Use it for:**
  - Background operations
  - Notifications
  - Accessing resources without UI

kotlin
CopyEdit

```
class MyService : Service() {
    override fun onStartCommand(...) {
        val name = getString(R.string.app_name) // using service context
    }
}
```

🔷 4. BroadcastReceiver Context

- **Scope:** Only during `onReceive()`
- **Use it for:**
  - Quick background tasks
  - Scheduling work
  - Accessing system info (e.g., network state)

kotlin
CopyEdit

```
override fun onReceive(context: Context, intent: Intent) {
    Toast.makeText(context, "Broadcast received", Toast.LENGTH_SHORT).show()
}
```

🔷 5. View Context

- **Context from a UI component**

Android interview Questions

- **Use it for:**
  - Showing toasts
  - Inflating other views

kotlin
CopyEdit

```
button.setOnClickListener {
    Toast.makeText(it.context, "Clicked", Toast.LENGTH_SHORT).show()
}
```

✅ What is WorkManager in Android?

WorkManager is a Jetpack library used to schedule and run **deferrable**, **asynchronous**, and **guaranteed** background tasks that **must be completed even if the app is killed or the device restarts.**

📌 When to Use WorkManager?

Use WorkManager for tasks that are:

- **Guaranteed to execute**
- **Deferrable** (not immediate)
- Need **constraints** (e.g., only run on Wi-Fi or charging)

🔍 Use Cases:

- Uploading logs or data to a server
- Syncing data in background
- Periodic backups
- Deferred image processing

Android interview Questions

Feature	Description
✅ Persistent	Survives app kills and reboots
🔁 Retry support	Automatically retries failed tasks
🔗 Chaining	Supports task chaining and dependencies
📅 Periodic	Supports recurring tasks

2. Create a Worker Class

kotlin
CopyEdit

```
class UploadWorker(appContext: Context, params: WorkerParameters) :
    Worker(appContext, params) {

    override fun doWork(): Result {
        // Perform your background task
        uploadDataToServer()

        // Return success or failure
        return Result.success()
    }
}
```

---

3. Enqueue Work (One-Time)

kotlin
CopyEdit

```
val workRequest = OneTimeWorkRequestBuilder<UploadWorker>().build()
WorkManager.getInstance(context).enqueue(workRequest)
```

---

4. Enqueue Periodic Work

kotlin
CopyEdit

```
val periodicWork = PeriodicWorkRequestBuilder<UploadWorker>(
    15, TimeUnit.MINUTES
).build()

WorkManager.getInstance(context).enqueue(periodicWork)
```

⛓️ Work Chaining Example

kotlin

Android interview Questions

```
WorkManager.getInstance(context)
    .beginWith(work1)
    .then(work2)
    .enqueue()
```

📊 Result Types

Result Type	Meaning
`Result.success()`	Task completed successfully
`Result.failure()`	Task failed, won't be retried
`Result.retry()`	Task failed, will retry automatically

✅ What is Retrofit in Android?

📘 Definition:

Retrofit is a **type-safe HTTP client** for Android and Java developed by **Square**. It simplifies the process of making **network/API calls** by allowing you to define REST API requests using **Java/Kotlin interfaces.**

🚀 Why Use Retrofit?

- Easy to use and maintain
- Supports GET, POST, PUT, DELETE, etc.
- Parses JSON/XML into objects using Gson, Moshi, etc.
- Works well with Coroutines, LiveData, RxJava

🔧 How to Make Network Calls with Retrofit (Step-by-Step)

Android interview Questions

CopyEdit

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
```

✅ Step 2: Create a Data Model

kotlin
CopyEdit

```
data class User(
    val id: Int,
    val name: String,
    val email: String
)
```

✅ Step 3: Define the API Interface

kotlin
CopyEdit

```
interface ApiService {
    @GET("users")
    suspend fun getUsers(): List<User>

    @POST("users")
    suspend fun createUser(@Body user: User): Response<User>
}
```

✅ Step 4: Create the Retrofit Instance

kotlin
CopyEdit

```
object RetrofitClient {
    private const val BASE_URL = "https://example.com/api/"

    val apiService: ApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())  // JSON ->
Object
            .build()
            .create(ApiService::class.java)
    }
}
```

Android interview Questions

CopyEdit

```
viewModelScope.launch {
    try {
        val users = RetrofitClient.apiService.getUsers()
        // Use the data (e.g., update LiveData)
    } catch (e: Exception) {
        Log.e("Error", "Failed to fetch users: ${e.message}")
    }
}
```

✅ Bonus: Handling Response

Use `Response<T>` for full control over status codes and errors:

kotlin
CopyEdit

```
val response = RetrofitClient.apiService.createUser(newUser)
if (response.isSuccessful) {
    val createdUser = response.body()
} else {
    Log.e("API", "Error: ${response.errorBody()?.string()}")
}
```

🔁 Retrofit with Coroutines vs LiveData vs RxJava

Feature	Description
Coroutines	Simple and modern (recommended for Kotlin)
LiveData	Used when observing data in View layer
RxJava	Powerful but complex (optional in new projects)

Handling API errors in **Retrofit** is crucial for building robust Android apps. Here's a complete guide on how to handle errors like **timeouts**, **HTTP failures**, **no internet**, and **server-side errors (4xx, 5xx).**

✅ 1. Basic Error Handling with Try-Catch (Coroutine + Retrofit)

Use `try-catch` to catch exceptions like:

- No internet (`IOException`)

Android interview Questions

- Server error (`HttpException`)

kotlin
CopyEdit

```kotlin
suspend fun fetchData(): ResultType? {
    return try {
        val response = apiService.getData()
        if (response.isSuccessful) {
            response.body()
        } else {
            // Handle HTTP error (4xx, 5xx)
            Log.e("API_ERROR", response.errorBody()?.string() ?: "Unknown error")
            null
        }
    } catch (e: IOException) {
        // Network or conversion error
        Log.e("NETWORK_ERROR", "No internet or timeout: ${e.message}")
        null
    } catch (e: HttpException) {
        // Non-2xx HTTP response
        Log.e("HTTP_ERROR", "HTTP exception: ${e.code()}")
        null
    } catch (e: Exception) {
        Log.e("UNKNOWN_ERROR", "Unknown error: ${e.localizedMessage}")
        null
    }
}
```

✅ 2. Sealed Class for API Result (Recommended)

Use a sealed class to wrap success, error, and loading states:

kotlin
CopyEdit

```kotlin
sealed class ApiResult<out T> {
    data class Success<T>(val data: T): ApiResult<T>()
    data class Error(val message: String): ApiResult<Nothing>()
    object Loading : ApiResult<Nothing>()
}
```

Then use it like this:

kotlin
CopyEdit

```kotlin
suspend fun safeApiCall(): ApiResult<MyData> {
    return try {
        val response = api.getData()
        if (response.isSuccessful && response.body() != null) {
            ApiResult.Success(response.body()!!)
        } else {
```

Android interview Questions

```
            ApiResult.Error("No Internet Connection")
        } catch (e: HttpException) {
            ApiResult.Error("HTTP Error: ${e.code()}")
        } catch (e: Exception) {
            ApiResult.Error("Unknown Error: ${e.message}")
        }
    }
```

✅ 3. ViewModel Layer Usage

kotlin
CopyEdit

```
val _state = MutableStateFlow<ApiResult<MyData>>(ApiResult.Loading)
val state: StateFlow<ApiResult<MyData>> = _state

fun fetchData() {
    viewModelScope.launch {
        _state.value = ApiResult.Loading
        _state.value = repository.safeApiCall()
    }
}
```

Then observe it in Compose:

kotlin
CopyEdit

```
when (val result = viewModel.state.collectAsState().value) {
    is ApiResult.Success -> Text("Data: ${result.data}")
    is ApiResult.Error -> Text("Error: ${result.message}")
    is ApiResult.Loading -> CircularProgressIndicator()
}
```

✅ What is the Repository Pattern in Android?

The **Repository Pattern** is a **design pattern** used to **abstract the data layer** of your app. It acts as a **single source of truth** for data—whether it's from a **remote API**, **local database**, or **cache**—and **hides the implementation details** from the rest of the app (like ViewModel or UI).

🔧 Purpose of Repository Pattern

- Centralizes data access logic
- Decouples ViewModel from data sources

Android interview Questions

- Simplifies switching between data sources (e.g., API vs Room)

🧱 Basic Architecture Using Repository

scss
CopyEdit

```
UI (Jetpack Compose / Activity / Fragment)
        ↓
   ViewModel (MVVM)
        ↓
   Repository
     ↙      ↘
Network   Local DB (Room)
```

✅ What is a Configuration Change in Android?

A **Configuration Change** is when the system changes certain device characteristics **while your app is running**, such as:

Change Type	Examples
Orientation	Portrait ↔ Landscape
Language	English ↔ Hindi
Keyboard availability	Physical keyboard plugged in
Screen size/density	Connecting to external display

❗ Problem with Configuration Changes

When a configuration change occurs:

Activity is destroyed and recreated by default
Which means:

- UI state may be **lost**

Android interview Questions

- Fragments and variables are **recreated**

✅ How to Handle Configuration Changes

🔹 1. Use ViewModel (Recommended)

`ViewModel` survives configuration changes because it's tied to the lifecycle of the **Activity or Fragment**, not the UI.

kotlin
CopyEdit
```
class MyViewModel : ViewModel() {
    val count = MutableLiveData(0)
}
```

In Composable:

kotlin
CopyEdit
```
val viewModel: MyViewModel = viewModel()
val count by viewModel.count.observeAsState(0)
Text("Count: $count")
```

✅ **Preserves state across rotation** without recreating the data.

🔹 2. onSaveInstanceState() & Bundle

Use this when you need to save small bits of UI state manually (like scroll position, text input).

kotlin
CopyEdit
```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt("counter", counter)
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    counter = savedInstanceState?.getInt("counter") ?: 0
}
```

✅ Good for small data (< 1MB), like flags or input values.

Android interview Questions

Stores state across configuration changes using `Bundle`.

kotlin
CopyEdit
```
val counter = rememberSaveable { mutableStateOf(0) }
```

✅ Best for preserving Compose state across rotations.

🔹 **4. Handle Configuration Yourself (Advanced)**

In `AndroidManifest.xml`:

xml
CopyEdit
```
<activity
    android:name=".MainActivity"
    android:configChanges="orientation|screenSize">
</activity>
```

Then override this in Activity:

kotlin
CopyEdit
```
override fun onConfigurationChanged(newConfig: Configuration) {
    super.onConfigurationChanged(newConfig)
    // Handle changes manually
}
```

⚠️ **Not recommended** unless you fully manage everything yourself.

✅ Summary Table

Method	Use Case	Survives Rotation?
ViewModel	Store data/state	✅ Yes
onSaveInstanceState	Small UI state (manual)	✅ Yes
rememberSaveable (Compose)	Store Composable state	✅ Yes
configChanges in manifest	Full control (not recommended)	✅ Yes
Global storage (Room/DB)	Persistent data	✅ Yes

Android interview Questions

✅ 1. Check if a number is prime

```
fun isPrime(n: Int): Boolean {
    if (n < 2) return false
    for (i in 2..Math.sqrt(n.toDouble()).toInt()) {
        if (n % i == 0) return false
    }
    return true
}
```

Time Complexity: O(√n)

✅ 2. Find GCD of two numbers (Euclidean Algorithm)

kotlin

CopyEdit

```
fun gcd(a: Int, b: Int): Int {
    return if (b == 0) a else gcd(b, a % b)
}
```

Time Complexity: O(log(min(a, b)))

✅ 3. Find LCM of two numbers

kotlin

CopyEdit

```
fun lcm(a: Int, b: Int): Int {
    return a * b / gcd(a, b)
}
```

Time Complexity: O(log(min(a, b)))

✅ 4. Check if a number is palindrome

kotlin

CopyEdit

```
fun isPalindrome(n: Int): Boolean {
    var num = n
    var rev = 0
    while (num > 0) {
        rev = rev * 10 + num % 10
        num /= 10
    }
    return rev == n
}
```

Time Complexity: O($\log_{10} n$)

✅ 5. Reverse a number

kotlin

CopyEdit

```
fun reverseNumber(n: Int): Int {
    var num = n
    var rev = 0
    while (num != 0) {
        rev = rev * 10 + num % 10
        num /= 10
    }
    return rev
}
```

Time Complexity: O($\log_{10}$n)

✅ 6. Check if a number is Armstrong

kotlin

CopyEdit

```
fun isArmstrong(n: Int): Boolean {
    val digits = n.toString().length
    var sum = 0
    var num = n
    while (num > 0) {
        val d = num % 10
        sum += Math.pow(d.toDouble(), digits.toDouble()).toInt()
        num /= 10
    }
    return sum == n
}
```

Time Complexity: O($\log_{10} n$)

✅ 7. Count digits in a number

kotlin

CopyEdit

```
fun countDigits(n: Int): Int {
    var count = 0
    var num = n
    while (num != 0) {
        count++
        num /= 10
    }
    return count
}
```

Time Complexity: O($\log_{10} n$)

✅ 8. Sum of digits of a number

kotlin

CopyEdit

```
fun sumOfDigits(n: Int): Int {
    var sum = 0
    var num = n
    while (num > 0) {
        sum += num % 10
        num /= 10
    }
    return sum
}
```

Time Complexity: O($\log_{10} n$)

✅ 9. Check if a number is even or odd

kotlin

CopyEdit

```
fun isEven(n: Int): Boolean {
    return n % 2 == 0
}
```

Time Complexity: O(1)

✅ 10. Find factorial of a number

kotlin

CopyEdit

```
fun factorial(n: Int): Long {
    var result = 1L
    for (i in 2..n) {
        result *= i
    }
    return result
}
```

Time Complexity: O(n)

Android interview Questions

✅ 11. Print all prime numbers up to N (Sieve of Eratosthenes)

kotlin

CopyEdit

```
fun sieve(n: Int) {
    val prime = BooleanArray(n + 1) { true }
    prime[0] = false
    prime[1] = false
    for (i in 2..n) {
        if (prime[i]) {
            var j = i * 2
            while (j <= n) {
                prime[j] = false
                j += i
            }
        }
    }
    for (i in 2..n) if (prime[i]) println(i)
}
```

Time Complexity: O(n log log n)

✅ 12. Check if a year is leap year

```
fun isLeapYear(year: Int): Boolean {
    return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)
}
```

Time Complexity: O(1)

✅ 13. Calculate power (a^b)

kotlin

CopyEdit

```
fun power(a: Int, b: Int): Long {
    var result = 1L
    repeat(b) { result *= a }
    return result
}
```

Time Complexity: O(b)

✅ 14. Calculate power using fast exponentiation

kotlin

CopyEdit

```
fun fastPower(a: Int, b: Int): Long {
    var base = a.toLong()
    var exp = b
    var result = 1L
    while (exp > 0) {
        if (exp % 2 == 1) result *= base
        base *= base
        exp /= 2
    }
    return result
}
```

Time Complexity: O(log b)

✅ 15. Check if a number is a power of 2

```
fun isPowerOfTwo(n: Int): Boolean {
    return n > 0 && (n and (n - 1)) == 0
}
```

Time Complexity: O(1)

✅ 16. Find nth Fibonacci number (iterative)

kotlin

CopyEdit

```
fun fibonacci(n: Int): Int {
    if (n <= 1) return n
    var a = 0
    var b = 1
    repeat(n - 1) {
        val temp = a + b
        a = b
        b = temp
    }
    return b
}
```

Time Complexity: O(n)

✅ 17. Check if a number is a perfect square

kotlin

CopyEdit

```
fun isPerfectSquare(n: Int): Boolean {
    val sqrt = Math.sqrt(n.toDouble()).toInt()
    return sqrt * sqrt == n
}
```

Time Complexity: O(1)

✅ 18. Count number of trailing zeros in factorial

kotlin

CopyEdit

```
fun trailingZeros(n: Int): Int {
    var count = 0
    var i = 5
    while (n / i >= 1) {
        count += n / i
        i *= 5
    }
    return count
}
```

Time Complexity: $O(\log_5 n)$

✅ 19. LCM of array of numbers

kotlin

CopyEdit

```
fun lcmArray(arr: IntArray): Int {
    var res = arr[0]
    for (i in 1 until arr.size) {
        res = lcm(res, arr[i])
    }
    return res
}
```

Time Complexity: O(n * log m), where m is the max element

Android interview Questions

✅ 20. Check if number is strong number (sum of factorials of digits = number)

kotlin

CopyEdit

```
fun isStrong(n: Int): Boolean {
    var num = n
    var sum = 0
    while (num > 0) {
        val digit = num % 10
        sum += factorial(digit).toInt()
        num /= 10
    }
    return sum == n
}
```

Time Complexity: O($\log_{10} n$)

1. Find the Maximum Element in an Array

kotlin
CopyEdit

```
fun findMax(arr: IntArray): Int {
    var max = arr[0]
    for (i in arr.indices) {
        if (arr[i] > max)
       max = arr[i]
    }
    return max
```

Android interview Questions

Time Complexity: O(n)

✅ 2. Find the Minimum Element in an Array

kotlin
CopyEdit

```kotlin
fun findMin(arr: IntArray): Int {
    var min = arr[0]
    for (i in arr.indices) {
        if (arr[i] < min) min = arr[i]
    }
    return min
}
```

🕒 Time Complexity: O(n)

✅ 3. Check if Array is Sorted

kotlin
CopyEdit

```kotlin
fun isSorted(arr: IntArray): Boolean {
    for (i in 0 until arr.size - 1) {
        if (arr[i] > arr[i + 1]) return false
    }
    return true
}
```

🕒 Time Complexity: O(n)

✅ 4. Find Sum of All Elements

kotlin
CopyEdit

```kotlin
fun arraySum(arr: IntArray): Int {
    var sum = 0
    for (num in arr) sum += num
    return sum
}
```

🕒 Time Complexity: O(n)

Android interview Questions

CopyEdit

```
fun linearSearch(arr: IntArray, key: Int): Int {
    for (i in arr.indices) {
        if (arr[i] == key) return i
    }
    return -1
}
```

Time Complexity: O(n)

---

6. Binary Search (sorted array)

kotlin
CopyEdit

```
fun binarySearch(arr: IntArray, key: Int): Int {
    var low = 0
    var high = arr.size - 1
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] == key) return mid
        else if (arr[mid] < key) low = mid + 1
        else high = mid - 1
    }
    return -1
}
```

Time Complexity: O(log n)

---

7. Reverse an Array

kotlin
CopyEdit

```
fun reverseArray(arr: IntArray): IntArray {
    var start = 0
    var end = arr.size - 1
    while (start < end) {
        val temp = arr[start]
        arr[start] = arr[end]
        arr[end] = temp
        start++
        end--
    }
    return arr
}
```

Android interview Questions

✅ 8. Count Occurrences of an Element

kotlin
CopyEdit

```kotlin
fun countOccurrences(arr: IntArray, key: Int): Int {
    var count = 0
    for (num in arr) {
        if (num == key) count++
    }
    return count
}
```

🕒 Time Complexity: `O(n)`

✅ 9. Merge Two Sorted Arrays

kotlin
CopyEdit

```kotlin
fun mergeSortedArrays(a: IntArray, b: IntArray): IntArray {
    val merged = IntArray(a.size + b.size)
    var i = 0; var j = 0; var k = 0
    while (i < a.size && j < b.size) {
        if (a[i] < b[j]) merged[k++] = a[i++]
        else merged[k++] = b[j++]
    }
    while (i < a.size) merged[k++] = a[i++]
    while (j < b.size) merged[k++] = b[j++]
    return merged
}
```

🕒 Time Complexity: `O(n + m)`

✅ 10. Left Rotate Array by 1

kotlin
CopyEdit

```kotlin
fun leftRotateByOne(arr: IntArray): IntArray {
    val first = arr[0]
    for (i in 0 until arr.size - 1) {
        arr[i] = arr[i + 1]
    }
    arr[arr.size - 1] = first
    return arr
}
```

Android interview Questions

✅ 11. Find Duplicates in an Array

kotlin
CopyEdit

```kotlin
fun findDuplicates(arr: IntArray): List<Int> {
    val result = mutableListOf<Int>()
    for (i in arr.indices) {
        for (j in i + 1 until arr.size) {
            if (arr[i] == arr[j] && !result.contains(arr[i])) {
                result.add(arr[i])
            }
        }
    }
    return result
}
```

🕒 Time Complexity: O(n²)

✅ 12. Move Zeroes to End

kotlin
CopyEdit

```kotlin
fun moveZeroes(arr: IntArray): IntArray {
    var index = 0
    for (i in arr.indices) {
        if (arr[i] != 0) arr[index++] = arr[i]
    }
    while (index < arr.size) {
        arr[index++] = 0
    }
    return arr
}
```

🕒 Time Complexity: O(n)

✅ 13. Find Second Largest

kotlin
CopyEdit

```kotlin
fun secondLargest(arr: IntArray): Int {
    var max = Int.MIN_VALUE
    var second = Int.MIN_VALUE
    for (i in arr) {
        if (i > max) {
            second = max
```

Android interview Questions

```
        }
    }
    return second
}
```

Time Complexity: O(n)

✅ 14. Insert Element at Specific Index

kotlin
CopyEdit

```
fun insertAtIndex(arr: IntArray, value: Int, index: Int): IntArray {
    val newArr = IntArray(arr.size + 1)
    for (i in 0 until index) newArr[i] = arr[i]
    newArr[index] = value
    for (i in index until arr.size) newArr[i + 1] = arr[i]
    return newArr
}
```

Time Complexity: O(n)

✅ 15. Delete Element from Array

kotlin
CopyEdit

```
fun deleteElement(arr: IntArray, value: Int): IntArray {
    val result = mutableListOf<Int>()
    for (num in arr) {
        if (num != value) result.add(num)
    }
    return result.toIntArray()
}
```

Time Complexity: O(n)

✅ 16. Find Missing Number (1 to n)

kotlin
CopyEdit

```
fun missingNumber(arr: IntArray, n: Int): Int {
    val total = n * (n + 1) / 2
    var sum = 0
    for (num in arr) sum += num
    return total - sum
```

Android interview Questions

Time Complexity: O(n)

✅ 17. Print All Subarrays

kotlin
CopyEdit

```
fun printSubarrays(arr: IntArray) {
    for (i in arr.indices) {
        for (j in i until arr.size) {
            for (k in i..j) print("${arr[k]} ")
            println()
        }
    }
}
```

🕒 Time Complexity: O(n³)

✅ 18. Find Common Elements in Two Arrays

kotlin
CopyEdit

```
fun commonElements(a: IntArray, b: IntArray): List<Int> {
    val result = mutableListOf<Int>()
    for (i in a) {
        for (j in b) {
            if (i == j && !result.contains(i)) {
                result.add(i)
            }
        }
    }
    return result
}
```

🕒 Time Complexity: O(n * m)

✅ 19. Count Even and Odd Numbers

kotlin
CopyEdit

```
fun countEvenOdd(arr: IntArray): Pair<Int, Int> {
    var even = 0
    var odd = 0
    for (num in arr) {
        if (num % 2 == 0) even++
        else odd++
```

Android interview Questions

🕒 Time Complexity: O(n)

✅ 20. Calculate Frequency of Each Element

kotlin
CopyEdit

```
fun elementFrequency(arr: IntArray): Map<Int, Int> {
    val freq = mutableMapOf<Int, Int>()
    for (num in arr) {
        freq[num] = freq.getOrDefault(num, 0) + 1
    }
    return freq
}
```

✅ 1. Find the Largest Element

kotlin
CopyEdit

```
fun findMax(arr: IntArray): Int = arr.maxOrNull() ?: Int.MIN_VALUE
```

🕒 **Time Complexity:** O(n)

✅ 2. Find Second Largest Element

kotlin
CopyEdit

```
fun secondLargest(arr: IntArray): Int? {
    var max = Int.MIN_VALUE
    var second = Int.MIN_VALUE
    for (num in arr) {
        if (num > max) {
            second = max
            max = num
        } else if (num != max && num > second) {
            second = num
        }
    }
    return if (second == Int.MIN_VALUE) null else second
}
```

🕒 **Time Complexity:** O(n)

Android interview Questions

CopyEdit

```
fun isSorted(arr: IntArray): Boolean {
    for (i in 0 until arr.size - 1) {
        if (arr[i] > arr[i + 1]) return false
    }
    return true
}
```

Time Complexity: O(n)

---

4. Reverse Array

kotlin
CopyEdit

```
fun reverseArray(arr: IntArray) {
    var start = 0
    var end = arr.lastIndex
    while (start < end) {
        arr[start] = arr[end].also { arr[end] = arr[start] }
        start++
        end--
    }
}
```

Time Complexity: O(n)

---

5. Sum of Array Elements

kotlin
CopyEdit

```
fun sumArray(arr: IntArray): Int = arr.sum()
```

Time Complexity: O(n)

---

6. Frequency of Elements

kotlin
CopyEdit

```
fun frequencyMap(arr: IntArray): Map<Int, Int> =
    arr.groupingBy { it }.eachCount()
```

Time Complexity: O(n)

Android interview Questions

kotlin
CopyEdit
```
fun leftRotate(arr: IntArray, k: Int): IntArray {
    val d = k % arr.size
    return arr.drop(d) + arr.take(d)
}
```

🕒 **Time Complexity:** O(n)
🧠 **Note:** Uses additional space

✅ 8. Move Zeros to End

kotlin
CopyEdit
```
fun moveZerosToEnd(arr: IntArray): IntArray {
    val result = arr.filter { it != 0 }.toMutableList()
    repeat(arr.size - result.size) { result.add(0) }
    return result.toIntArray()
}
```

🕒 **Time Complexity:** O(n)

✅ 9. Kadane's Algorithm (Max Subarray Sum)

kotlin
CopyEdit
```
fun maxSubarraySum(arr: IntArray): Int {
    var maxSum = arr[0]
    var currentSum = arr[0]
    for (i in 1 until arr.size) {
        currentSum = maxOf(arr[i], currentSum + arr[i])
        maxSum = maxOf(maxSum, currentSum)
    }
    return maxSum
}
```

🕒 **Time Complexity:** O(n)

✅ 10. Missing Number (1 to N)

kotlin
CopyEdit
```
fun missingNumber(arr: IntArray): Int {
```

Android interview Questions

```
}
```

🕒 **Time Complexity:** `O(n)`

✅ 11. Find Duplicates

kotlin
CopyEdit

```
fun findDuplicates(arr: IntArray): Set<Int> {
    val seen = mutableSetOf<Int>()
    val duplicates = mutableSetOf<Int>()
    for (num in arr) {
        if (!seen.add(num)) duplicates.add(num)
    }
    return duplicates
}
```

🕒 **Time Complexity:** `O(n)`

✅ 12. Merge Two Sorted Arrays

kotlin
CopyEdit

```
fun mergeSortedArrays(a: IntArray, b: IntArray): IntArray {
    var i = 0; var j = 0
    val result = mutableListOf<Int>()
    while (i < a.size && j < b.size) {
        if (a[i] < b[j]) result.add(a[i++]) else result.add(b[j++])
    }
    result.addAll(a.drop(i))
    result.addAll(b.drop(j))
    return result.toIntArray()
}
```

🕒 **Time Complexity:** `O(n + m)`
Where `n` and `m` are sizes of arrays `a` and `b`

✅ 13. Intersection of Two Arrays

kotlin
CopyEdit

```
fun intersectArrays(a: IntArray, b: IntArray): Set<Int> {
    return a.toSet().intersect(b.toSet())
}
```

Android interview Questions

✅ 14. Sort 0s, 1s, and 2s (Dutch Flag)

kotlin
CopyEdit

```kotlin
fun sort012(arr: IntArray) {
    var low = 0
    var mid = 0
    var high = arr.lastIndex
    while (mid <= high) {
        when (arr[mid]) {
            0 -> arr[low] = arr[mid].also { arr[mid] = arr[low]; low++; mid++ }
            1 -> mid++
            2 -> arr[mid] = arr[high].also { arr[high] = arr[mid]; high-- }
        }
    }
}
```

🕒 **Time Complexity:** O(n)

✅ 15. Longest Subarray with Sum (Positive Integers)

kotlin
CopyEdit

```kotlin
fun longestSubarrayWithSum(arr: IntArray, k: Int): Int {
    var start = 0
    var sum = 0
    var maxLen = 0
    for (end in arr.indices) {
        sum += arr[end]
        while (sum > k) sum -= arr[start++]
        if (sum == k) maxLen = maxOf(maxLen, end - start + 1)
    }
    return maxLen
}
```

🕒 **Time Complexity:** O(n)

✅ 16. Equilibrium Index

kotlin
CopyEdit

```kotlin
fun equilibriumIndex(arr: IntArray): Int {
    val total = arr.sum()
    var leftSum = 0
    for (i in arr.indices) {
```

Android interview Questions

```
    return -1
}
```

🕒 **Time Complexity:** O(n)

✅ 17. Majority Element (> n/2 times)

kotlin
CopyEdit

```
fun majorityElement(arr: IntArray): Int? {
    var count = 0
    var candidate: Int? = null
    for (num in arr) {
        if (count == 0) candidate = num
        count += if (num == candidate) 1 else -1
    }
    return if (arr.count { it == candidate } > arr.size / 2) candidate else null
}
```

🕒 **Time Complexity:** O(n)

✅ 18. Subarray with Given Sum (Negative Allowed)

kotlin
CopyEdit

```
fun subarraySum(arr: IntArray, k: Int): Boolean {
    val map = mutableMapOf(0 to 1)
    var sum = 0
    for (num in arr) {
        sum += num
        if (map.containsKey(sum - k)) return true
        map[sum] = map.getOrDefault(sum, 0) + 1
    }
    return false
}
```

🕒 **Time Complexity:** O(n)

✅ 19. Maximum Product Subarray

kotlin
CopyEdit

```
fun maxProductSubarray(arr: IntArray): Int {
    var maxProd = arr[0]
```

Android interview Questions

```
        val temp = maxTemp
        maxTemp = maxOf(arr[i], maxTemp * arr[i], minTemp * arr[i])
        minTemp = minOf(arr[i], temp * arr[i], minTemp * arr[i])
        maxProd = maxOf(maxProd, maxTemp)
    }
    return maxProd
}
```

🕒 **Time Complexity:** O(n)

✅ 20. Search in Rotated Sorted Array

kotlin
CopyEdit

```
fun searchRotated(arr: IntArray, target: Int): Int {
    var low = 0
    var high = arr.lastIndex
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] == target) return mid
        if (arr[low] <= arr[mid]) {
            if (target >= arr[low] && target < arr[mid]) high = mid - 1
            else low = mid + 1
        } else {
            if (target > arr[mid] && target <= arr[high]) low = mid + 1
            else high = mid - 1
        }
    }
    return -1
}
```

🕒 **Time Complexity:** O(log n)

Android interview Questions

kotlin
CopyEdit

```kotlin
fun reverseString(s: String): String {
    val result = CharArray(s.length)
    for (i in s.indices) {
        result[i] = s[s.length - 1 - i]
    }
    return String(result)
}
```

Time: O(n)

✅ 2. Check Palindrome

kotlin
CopyEdit

```kotlin
fun isPalindrome(s: String): Boolean {
    var i = 0
    var j = s.length - 1
    while (i < j) {
        if (s[i] != s[j]) return false
        i++
        j--
    }
    return true
}
```

Time: O(n)

✅ 3. Count Vowels and Consonants

kotlin
CopyEdit

```kotlin
fun countVowelsAndConsonants(s: String): Pair<Int, Int> {
    val vowels = charArrayOf('a','e','i','o','u','A','E','I','O','U')
    var v = 0
    var c = 0
    for (ch in s) {
        if (ch in 'A'..'Z' || ch in 'a'..'z') {
            if (vowels.contains(ch)) v++ else c++
        }
    }
    return Pair(v, c)
}
```

Time: O(n)

Android interview Questions

kotlin
CopyEdit

```kotlin
fun areAnagrams(a: String, b: String): Boolean {
    if (a.length != b.length) return false
    val freq = IntArray(26)
    for (i in a.indices) {
        freq[a[i] - 'a']++
        freq[b[i] - 'a']--
    }
    for (f in freq) if (f != 0) return false
    return true
}
```

🕒 Time: O(n)

✅ 5. Remove Duplicates

kotlin
CopyEdit

```kotlin
fun removeDuplicates(s: String): String {
    val seen = BooleanArray(256)
    val result = StringBuilder()
    for (ch in s) {
        if (!seen[ch.code]) {
            result.append(ch)
            seen[ch.code] = true
        }
    }
    return result.toString()
}
```

🕒 Time: O(n)

✅ 6. First Non-Repeating Character

kotlin
CopyEdit

```kotlin
fun firstNonRepeatingChar(s: String): Char? {
    val freq = IntArray(256)
    for (ch in s) freq[ch.code]++
    for (ch in s) if (freq[ch.code] == 1) return ch
    return null
}
```

🕒 Time: O(n)

Android interview Questions

kotlin
CopyEdit

```kotlin
fun areRotations(a: String, b: String): Boolean {
    if (a.length != b.length) return false
    for (i in a.indices) {
        var match = true
        for (j in b.indices) {
            if (a[(i + j) % a.length] != b[j]) {
                match = false
                break
            }
        }
        if (match) return true
    }
    return false
}
```

🕒 Time: O(n²)

✅ 8. Count Words

kotlin
CopyEdit

```kotlin
fun countWords(s: String): Int {
    var count = 0
    var inWord = false
    for (ch in s) {
        if (ch != ' ' && !inWord) {
            inWord = true
            count++
        } else if (ch == ' ') {
            inWord = false
        }
    }
    return count
}
```

🕒 Time: O(n)

✅ 9. Reverse Words in Sentence

kotlin
CopyEdit

```kotlin
fun reverseWords(s: String): String {
    val words = mutableListOf<String>()
    var word = ""
    for (ch in s) {
```

Android interview Questions

```
                word =
            }
        } else {
            word += ch
        }
    }
    if (word.isNotEmpty()) words.add(word)

    val result = StringBuilder()
    for (i in words.lastIndex downTo 0) {
        result.append(words[i])
        if (i != 0) result.append(" ")
    }
    return result.toString()
}
```

🕒 Time: O(n)

✅ 10. Longest Palindromic Substring

kotlin
CopyEdit

```
fun longestPalindrome(s: String): String {
    var maxLength = 1
    var start = 0
    for (i in s.indices) {
        var low = i - 1
        var high = i + 1
        while (high < s.length && s[high] == s[i]) high++
        while (low >= 0 && s[low] == s[i]) low--
        while (low >= 0 && high < s.length && s[low] == s[high]) {
            low--
            high++
        }
        val len = high - low - 1
        if (len > maxLength) {
            maxLength = len
            start = low + 1
        }
    }
    return s.substring(start, start + maxLength)
}
```

🕒 Time: O(n²)

✅ 11. Longest Common Prefix

kotlin

Android interview Questions

```
    val minLen = arr.minOf { it.length }
    var result = ""
    for (i in 0 until minLen) {
        val ch = arr[0][i]
        for (j in 1 until arr.size) {
            if (arr[j][i] != ch) return result
        }
        result += ch
    }
    return result
}
```

Time: O(n * m)

✅ 12. Is Subsequence

kotlin
CopyEdit

```
fun isSubsequence(s: String, t: String): Boolean {
    var i = 0
    var j = 0
    while (i < s.length && j < t.length) {
        if (s[i] == t[j]) i++
        j++
    }
    return i == s.length
}
```

Time: O(n)

✅ 13. Group Anagrams (naive)

kotlin
CopyEdit

```
fun areSameCharCount(a: String, b: String): Boolean {
    val freq = IntArray(26)
    for (ch in a) freq[ch - 'a']++
    for (ch in b) freq[ch - 'a']--
    return freq.all { it == 0 }
}

fun groupAnagrams(arr: Array<String>): List<List<String>> {
    val result = mutableListOf<MutableList<String>>()
    val visited = BooleanArray(arr.size)
    for (i in arr.indices) {
        if (visited[i]) continue
        val group = mutableListOf(arr[i])
        visited[i] = true
```

Android interview Questions

```
                visited[j] = true
            }
        }
        result.add(group)
    }
    return result
}
```

🕒 Time: O(n² * m)

✅ 14. Check if All Digits

kotlin
CopyEdit

```
fun isNumeric(s: String): Boolean {
    for (ch in s) {
        if (ch !in '0'..'9') return false
    }
    return true
}
```

🕒 Time: O(n)

✅ 15. Find All Permutations

kotlin
CopyEdit

```
fun permute(prefix: String, remaining: String, result: MutableList<String>) {
    if (remaining.isEmpty()) result.add(prefix)
    else {
        for (i in remaining.indices) {
            val next = remaining.substring(0, i) + remaining.substring(i + 1)
            permute(prefix + remaining[i], next, result)
        }
    }
}

fun stringPermutations(s: String): List<String> {
    val result = mutableListOf<String>()
    permute("", s, result)
    return result
}
```

🕒 Time: O(n!)

Android interview Questions

CopyEdit

```
fun compressString(s: String): String {
    val sb = StringBuilder()
    var count = 1
    for (i in 1 until s.length) {
        if (s[i] == s[i - 1]) count++
        else {
            sb.append(s[i - 1]).append(count)
            count = 1
        }
    }
    sb.append(s.last()).append(count)
    return sb.toString()
}
```

🕒 Time: O(n)

✅ 17. Check Unique Characters

kotlin
CopyEdit

```
fun hasUniqueChars(s: String): Boolean {
    val seen = BooleanArray(256)
    for (ch in s) {
        if (seen[ch.code]) return false
        seen[ch.code] = true
    }
    return true
}
```

🕒 Time: O(n)

✅ 18. Count Character Frequency

kotlin
CopyEdit

```
fun charCount(s: String): IntArray {
    val freq = IntArray(256)
    for (ch in s) {
        freq[ch.code]++
    }
    return freq
}
```

🕒 Time: O(n)

Android interview Questions

kotlin
CopyEdit

```kotlin
fun removeNonAlpha(s: String): String {
    val sb = StringBuilder()
    for (ch in s) {
        if ((ch in 'a'..'z') || (ch in 'A'..'Z')) {
            sb.append(ch)
        }
    }
    return sb.toString()
}
```

🕒 Time: O(n)

✅ 20. Pangram Check

kotlin
CopyEdit

```kotlin
fun isPangram(s: String): Boolean {
    val seen = BooleanArray(26)
    for (ch in s.lowercase()) {
        if (ch in 'a'..'z') {
            seen[ch - 'a'] = true
        }
    }
    return seen.all { it }
}
```

🕒 Time: O(n)

Android interview Questions

✅ 1. Linear Search

kotlin
CopyEdit

```
fun linearSearch(arr: IntArray, target: Int): Int {
    for (i in arr.indices) {
        if (arr[i] == target) return i
    }
    return -1
}
```

Time Complexity: O(n)

✅ 2. Binary Search (Iterative)

kotlin
CopyEdit

```
fun binarySearch(arr: IntArray, target: Int): Int {
    var start = 0
    var end = arr.lastIndex
    while (start <= end) {
        val mid = (start + end) / 2
        when {
            arr[mid] == target -> return mid
            arr[mid] < target -> start = mid + 1
            else -> end = mid - 1
        }
    }
    return -1
}
```

Time Complexity: O(log n) — Only for sorted arrays

✅ 3. Binary Search (Recursive)

kotlin
CopyEdit

```
fun binarySearchRec(arr: IntArray, target: Int, low: Int, high: Int): Int {
    if (low > high) return -1
    val mid = (low + high) / 2
    return when {
        arr[mid] == target -> mid
        arr[mid] < target -> binarySearchRec(arr, target, mid + 1, high)
        else -> binarySearchRec(arr, target, low, mid - 1)
    }
}
```

Android interview Questions

✅ 4. Find First Occurrence in Sorted Array

kotlin
CopyEdit

```kotlin
fun firstOccurrence(arr: IntArray, target: Int): Int {
    var low = 0
    var high = arr.lastIndex
    var result = -1
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] == target) {
            result = mid
            high = mid - 1
        } else if (arr[mid] < target) low = mid + 1
        else high = mid - 1
    }
    return result
}
```

Time Complexity: O(log n)

✅ 5. Find Last Occurrence in Sorted Array

kotlin
CopyEdit

```kotlin
fun lastOccurrence(arr: IntArray, target: Int): Int {
    var low = 0
    var high = arr.lastIndex
    var result = -1
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] == target) {
            result = mid
            low = mid + 1
        } else if (arr[mid] < target) low = mid + 1
        else high = mid - 1
    }
    return result
}
```

Time Complexity: O(log n)

✅ 6. Count of Occurrences

kotlin

Android interview Questions

```
    val last = lastOccurrence(arr, target)
    return if (first == -1) 0 else (last - first + 1)
}
```

Time Complexity: O(log n)

✅ 7. Search in Rotated Sorted Array

kotlin
CopyEdit

```
fun searchRotated(arr: IntArray, target: Int): Int {
    var low = 0
    var high = arr.lastIndex
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] == target) return mid
        if (arr[low] <= arr[mid]) {
            if (target >= arr[low] && target < arr[mid]) high = mid - 1
            else low = mid + 1
        } else {
            if (target > arr[mid] && target <= arr[high]) low = mid + 1
            else high = mid - 1
        }
    }
    return -1
}
```

Time Complexity: O(log n)

✅ 8. Find Peak Element

kotlin
CopyEdit

```
fun findPeak(arr: IntArray): Int {
    var low = 0
    var high = arr.lastIndex
    while (low < high) {
        val mid = (low + high) / 2
        if (arr[mid] < arr[mid + 1]) low = mid + 1
        else high = mid
    }
    return arr[low]
}
```

Time Complexity: O(log n)

Android interview Questions

✅ 9. Bubble Sort

kotlin
CopyEdit

```kotlin
fun bubbleSort(arr: IntArray) {
    val n = arr.size
    for (i in 0 until n) {
        for (j in 0 until n - i - 1) {
            if (arr[j] > arr[j + 1]) {
                val temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
            }
        }
    }
}
```

Time Complexity: O(n²)

✅ 10. Selection Sort

kotlin
CopyEdit

```kotlin
fun selectionSort(arr: IntArray) {
    for (i in arr.indices) {
        var minIdx = i
        for (j in i + 1 until arr.size) {
            if (arr[j] < arr[minIdx]) minIdx = j
        }
        val temp = arr[i]
        arr[i] = arr[minIdx]
        arr[minIdx] = temp
    }
}
```

Time Complexity: O(n²)

✅ 11. Insertion Sort

kotlin
CopyEdit

```kotlin
fun insertionSort(arr: IntArray) {
    for (i in 1 until arr.size) {
        val key = arr[i]
```

Android interview Questions

```
            j--
        }
        arr[j + 1] = key
    }
}
```

Time Complexity: O(n²)

✅ 12. Merge Sort

kotlin
CopyEdit

```
fun mergeSort(arr: IntArray, low: Int, high: Int) {
    if (low < high) {
        val mid = (low + high) / 2
        mergeSort(arr, low, mid)
        mergeSort(arr, mid + 1, high)
        merge(arr, low, mid, high)
    }
}

fun merge(arr: IntArray, low: Int, mid: Int, high: Int) {
    val left = arr.sliceArray(low..mid)
    val right = arr.sliceArray(mid + 1..high)
    var i = 0
    var j = 0
    var k = low
    while (i < left.size && j < right.size) {
        arr[k++] = if (left[i] < right[j]) left[i++] else right[j++]
    }
    while (i < left.size) arr[k++] = left[i++]
    while (j < right.size) arr[k++] = right[j++]
}
```

Time Complexity: O(n log n)

✅ 13. Quick Sort

kotlin
CopyEdit

```
fun quickSort(arr: IntArray, low: Int, high: Int) {
    if (low < high) {
        val pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
    }
}
```

Android interview Questions

```
    for (j in low until high) {
        if (arr[j] < pivot) {
            i++
            arr[i] = arr[j].also { arr[j] = arr[i] }
        }
    }
    arr[i + 1] = arr[high].also { arr[high] = arr[i + 1] }
    return i + 1
}
```

Time Complexity: O(n log n) average, O(n²) worst

✅ 14. Sort 0s, 1s, and 2s (Dutch National Flag)

kotlin
CopyEdit

```
fun sortColors(arr: IntArray) {
    var low = 0
    var mid = 0
    var high = arr.lastIndex
    while (mid <= high) {
        when (arr[mid]) {
            0 -> {
                arr[low] = arr[mid].also { arr[mid] = arr[low] }
                low++; mid++
            }
            1 -> mid++
            2 -> {
                arr[mid] = arr[high].also { arr[high] = arr[mid] }
                high--
            }
        }
    }
}
```

Time Complexity: O(n)

✅ 15. Check if Array is Sorted

kotlin
CopyEdit

```
fun isSorted(arr: IntArray): Boolean {
    for (i in 0 until arr.lastIndex) {
        if (arr[i] > arr[i + 1]) return false
    }
    return true
}
```

Android interview Questions

✅ 16. Find Missing Number in Sorted Array

kotlin
CopyEdit

```
fun missingNumber(arr: IntArray): Int {
    var low = 0
    var high = arr.lastIndex
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] == mid) low = mid + 1
        else high = mid - 1
    }
    return low
}
```

Time Complexity: O(log n)

✅ 17. Find Square Root using Binary Search

kotlin
CopyEdit

```
fun squareRoot(n: Int): Int {
    var low = 0
    var high = n
    var ans = 0
    while (low <= high) {
        val mid = (low + high) / 2
        if (mid * mid <= n) {
            ans = mid
            low = mid + 1
        } else high = mid - 1
    }
    return ans
}
```

Time Complexity: O(log n)

✅ 18. Find Ceiling of Number in Sorted Array

kotlin
CopyEdit

```
fun findCeil(arr: IntArray, target: Int): Int {
    var low = 0
    var high = arr.lastIndex
    var ans = -1
```

Android interview Questions

```
            ans = arr[mid]
            high = mid - 1
        } else low = mid + 1
    }
    return ans
}
```

Time Complexity: O(log n)

✅ 19. Find Floor of Number in Sorted Array

kotlin
CopyEdit

```
fun findFloor(arr: IntArray, target: Int): Int {
    var low = 0
    var high = arr.lastIndex
    var ans = -1
    while (low <= high) {
        val mid = (low + high) / 2
        if (arr[mid] <= target) {
            ans = arr[mid]
            low = mid + 1
        } else high = mid - 1
    }
    return ans
}
```

Time Complexity: O(log n)

✅ 20. Count Pairs with Given Sum (sorted array)

kotlin
CopyEdit

```
fun countPairs(arr: IntArray, sum: Int): Int {
    var i = 0
    var j = arr.lastIndex
    var count = 0
    while (i < j) {
        val s = arr[i] + arr[j]
        if (s == sum) {
            count++
            i++; j--
        } else if (s < sum) i++
        else j--
    }
    return count
}
```

Android interview Questions

✅ 1. Solid Rectangle

markdown
CopyEdit

```
*****
*****
*****
```

kotlin
CopyEdit

```kotlin
fun solidRectangle(rows: Int, cols: Int) {
    for (i in 1..rows) {
        for (j in 1..cols) {
            print("*")
        }
        println()
    }
}
```

✅ 2. Hollow Rectangle

markdown
CopyEdit

```
*****
*   *
*****
```

kotlin
CopyEdit

```kotlin
fun hollowRectangle(rows: Int, cols: Int) {
    for (i in 1..rows) {
        for (j in 1..cols) {
            if (i == 1 || i == rows || j == 1 || j == cols)
                print("*")
            else
                print(" ")
        }
        println()
    }
}
```

✅ 3. Right-Angled Triangle

markdown
CopyEdit

```
*
```

Android interview Questions

kotlin
CopyEdit
```
fun rightAngledTriangle(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) {
            print("*")
        }
        println()
    }
}
```

✅ 4. Inverted Right-Angled Triangle

markdown
CopyEdit
```
***
**
*
```

kotlin
CopyEdit
```
fun invertedTriangle(n: Int) {
    for (i in n downTo 1) {
        for (j in 1..i) {
            print("*")
        }
        println()
    }
}
```

✅ 5. Right-Aligned Triangle

markdown
CopyEdit
```
 *
 **
***
```

kotlin
CopyEdit
```
fun rightAlignedTriangle(n: Int) {
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        for (k in 1..i) print("*")
        println()
    }
}
```

Android interview Questions

```
1
12
123
```

kotlin
CopyEdit

```kotlin
fun numberTriangle(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) print(j)
        println()
    }
}
```

✅ 7. Inverted Number Triangle

CopyEdit

```
123
12
1
```

kotlin
CopyEdit

```kotlin
fun invertedNumberTriangle(n: Int) {
    for (i in n downTo 1) {
        for (j in 1..i) print(j)
        println()
    }
}
```

✅ 8. Floyd's Triangle

CopyEdit

```
1
2 3
4 5 6
```

kotlin
CopyEdit

```kotlin
fun floydsTriangle(n: Int) {
    var num = 1
    for (i in 1..n) {
        for (j in 1..i) {
            print("$num ")
            num++
        }
        println()
    }
}
```

Android interview Questions

CopyEdit
```
1
01
101
```

kotlin
CopyEdit
```kotlin
fun zeroOneTriangle(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) {
            if ((i + j) % 2 == 0) print("1") else print("0")
        }
        println()
    }
}
```

✅ 10. Pyramid

markdown
CopyEdit
```
  *
  ***
 *****
```

kotlin
CopyEdit
```kotlin
fun pyramid(n: Int) {
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        for (k in 1..(2 * i - 1)) print("*")
        println()
    }
}
```

✅ 11. Inverted Pyramid

markdown
CopyEdit
```
*****
  ***
   *
```

kotlin
CopyEdit
```kotlin
fun invertedPyramid(n: Int) {
    for (i in n downTo 1) {
        for (j in 1..(n - i)) print(" ")
        for (k in 1..(2 * i - 1)) print("*")
        println()
```

Android interview Questions

✅ 12. Diamond Pattern

markdown
CopyEdit

```
  *
  ***
 *****
  ***
   *
```

kotlin
CopyEdit

```kotlin
fun diamond(n: Int) {
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        for (k in 1..(2 * i - 1)) print("*")
        println()
    }
    for (i in (n - 1) downTo 1) {
        for (j in 1..(n - i)) print(" ")
        for (k in 1..(2 * i - 1)) print("*")
        println()
    }
}
```

✅ 13. Alphabet Triangle

css
CopyEdit

```
A
AB
ABC
```

kotlin
CopyEdit

```kotlin
fun alphabetTriangle(n: Int) {
    for (i in 1..n) {
        for (j in 0 until i) {
            print(('A' + j))
        }
        println()
    }
}
```

✅ 14. Inverted Alphabet Triangle

css

Android interview Questions

```
A
```

kotlin
CopyEdit

```kotlin
fun invertedAlphabetTriangle(n: Int) {
    for (i in n downTo 1) {
        for (j in 0 until i) {
            print(('A' + j))
        }
        println()
    }
}
```

✅ 15. Pascal's Triangle

CopyEdit

```
1
1 1
1 2 1
1 3 3 1
```

kotlin
CopyEdit

```kotlin
fun pascalTriangle(n: Int) {
    for (i in 0 until n) {
        var num = 1
        for (j in 0..i) {
            print("$num ")
            num = num * (i - j) / (j + 1)
        }
        println()
    }
}
```

✅ 16. Hollow Pyramid

markdown
CopyEdit

```
    *
   * *
  *   *
*******
```

kotlin
CopyEdit

Android interview Questions

```
        for (k in 1..(2 * i - 1)) {
            if (k == 1 || k == (2 * i - 1)) print("*") else print(" ")
        }
        println()
    }
    for (i in 1..(2 * n - 1)) print("*")
}
```

✅ 17. Left Half Diamond

markdown
CopyEdit

```
*
**
***
**
*
```

kotlin
CopyEdit

```
fun leftHalfDiamond(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) print("*")
        println()
    }
    for (i in (n - 1) downTo 1) {
        for (j in 1..i) print("*")
        println()
    }
}
```

✅ 18. Cross (X) Pattern

markdown
CopyEdit

```
*   *
 * *
  *
 * *
*   *
```

kotlin
CopyEdit

```
fun crossPattern(n: Int) {
    for (i in 1..n) {
        for (j in 1..n) {
            if (j == i || j == n - i + 1) print("*") else print(" ")
        }
        println()
    }
```

Android interview Questions

✅ 19. Number Pyramid

CopyEdit

```
  1
 1 2
1 2 3
```

kotlin
CopyEdit

```kotlin
fun numberPyramid(n: Int) {
    for (i in 1..n) {
        for (s in 1..(n - i)) print(" ")
        for (j in 1..i) print("$j ")
        println()
    }
}
```

✅ 20. Binary Number Pyramid

markdown
CopyEdit

```
   1
  0 1
 1 0 1
0 1 0 1
```

kotlin
CopyEdit

```kotlin
fun binaryPyramid(n: Int) {
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        var valToPrint = if (i % 2 == 0) 0 else 1
        for (k in 1..i) {
            print("$valToPrint ")
            valToPrint = 1 - valToPrint
        }
        println()
    }
}
```

✅ 1. Solid Rectangle

markdown
CopyEdit

```
*****
*****
*****
```

Android interview Questions

```
    for (i in 1..rows) {
        for (j in 1..cols) {
            print("*")
        }
        println()
    }
}
```

✅ 2. Hollow Rectangle

markdown
CopyEdit

```
*****
*   *
*****
```

kotlin
CopyEdit

```
fun hollowRectangle(rows: Int, cols: Int) {
    for (i in 1..rows) {
        for (j in 1..cols) {
            if (i == 1 || i == rows || j == 1 || j == cols) print("*")
            else print(" ")
        }
        println()
    }
}
```

✅ 3. Half Pyramid

markdown
CopyEdit

```
*
**
***
****
```

kotlin
CopyEdit

```
fun halfPyramid(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) print("*")
        println()
    }
}
```

Android interview Questions

markdown
CopyEdit
```
****
***
**
*
```

kotlin
CopyEdit
```kotlin
fun invertedHalfPyramid(n: Int) {
    for (i in n downTo 1) {
        for (j in 1..i) print("*")
        println()
    }
}
```

✅ 5. Right-Aligned Half Pyramid

markdown
CopyEdit
```
   *
  **
 ***
****
```

kotlin
CopyEdit
```kotlin
fun rightHalfPyramid(n: Int) {
    for (i in 1..n) {
        for (space in 1..(n - i)) print(" ")
        for (star in 1..i) print("*")
        println()
    }
}
```

✅ 6. Number Pyramid

yaml
CopyEdit
```
1
12
123
1234
```

kotlin
CopyEdit
```kotlin
fun numberPyramid(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) print(j)
        println()
```

Android interview Questions

✅ 7. Inverted Number Pyramid

yaml
CopyEdit

```
1234
123
12
1
```

kotlin
CopyEdit

```kotlin
fun invertedNumberPyramid(n: Int) {
    for (i in n downTo 1) {
        for (j in 1..i) print(j)
        println()
    }
}
```

✅ 8. Full Pyramid

markdown
CopyEdit

```
   *
  ***
 *****
*******
```

kotlin
CopyEdit

```kotlin
fun fullPyramid(n: Int) {
    for (i in 1..n) {
        for (space in 1..(n - i)) print(" ")
        for (star in 1..(2 * i - 1)) print("*")
        println()
    }
}
```

✅ 9. Inverted Full Pyramid

markdown
CopyEdit

```
*******
 *****
  ***
   *
```

kotlin

Android interview Questions

```
        for (space in 1..(n - i)) print("  ")
        for (star in 1..(2 * i - 1)) print("*")
        println()
    }
}
```

✅ 10. Pascal's Triangle

CopyEdit

```
1
1 1
1 2 1
1 3 3 1
```

kotlin
CopyEdit

```
fun pascalsTriangle(n: Int) {
    for (i in 0 until n) {
        var num = 1
        for (j in 0..i) {
            print("$num ")
            num = num * (i - j) / (j + 1)
        }
        println()
    }
}
```

✅ 11. Floyd’s Triangle

CopyEdit

```
1
2 3
4 5 6
```

kotlin
CopyEdit

```
fun floydsTriangle(n: Int) {
    var num = 1
    for (i in 1..n) {
        for (j in 1..i) {
            print("$num ")
            num++
        }
        println()
    }
}
```

Android interview Questions

```
1
0 1
1 0 1
```

kotlin
CopyEdit

```kotlin
fun zeroOneTriangle(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) {
            val value = if ((i + j) % 2 == 0) 1 else 0
            print("$value ")
        }
        println()
    }
}
```

✅ 13. Diamond Pattern

markdown
CopyEdit

```
   *
  ***
 *****
  ***
   *
```

kotlin
CopyEdit

```kotlin
fun diamond(n: Int) {
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        for (j in 1..(2 * i - 1)) print("*")
        println()
    }
    for (i in n - 1 downTo 1) {
        for (j in 1..(n - i)) print(" ")
        for (j in 1..(2 * i - 1)) print("*")
        println()
    }
}
```

✅ 14. Butterfly Pattern

markdown
CopyEdit

```
*       *
**     **
***   ***
********
```

Android interview Questions

kotlin
CopyEdit

```kotlin
fun butterfly(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) print("*")
        for (j in 1..2 * (n - i)) print(" ")
        for (j in 1..i) print("*")
        println()
    }
    for (i in n downTo 1) {
        for (j in 1..i) print("*")
        for (j in 1..2 * (n - i)) print(" ")
        for (j in 1..i) print("*")
        println()
    }
}
```

✅ 15. Hollow Pyramid

markdown
CopyEdit

```
   *
  * *
 *   *
*******
```

kotlin
CopyEdit

```kotlin
fun hollowPyramid(n: Int) {
    for (i in 1 until n) {
        for (j in 1..(n - i)) print(" ")
        for (j in 1..(2 * i - 1)) {
            if (j == 1 || j == 2 * i - 1) print("*") else print(" ")
        }
        println()
    }
    for (j in 1..(2 * n - 1)) print("*")
    println()
}
```

✅ 16. Mirror Half Pyramid

markdown
CopyEdit

```
    *
   **
  ***
 ****
```

Android interview Questions

```
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        for (j in 1..i) print("*")
        println()
    }
}
```

✅ 17. Character Triangle

css
CopyEdit

```
A
A B
A B C
```

kotlin
CopyEdit

```
fun charTriangle(n: Int) {
    for (i in 1..n) {
        var ch = 'A'
        for (j in 1..i) {
            print("$ch ")
            ch++
        }
        println()
    }
}
```

✅ 18. Repeating Character Triangle

css
CopyEdit

```
A
B B
C C C
```

kotlin
CopyEdit

```
fun repeatCharTriangle(n: Int) {
    for (i in 0 until n) {
        val ch = 'A' + i
        for (j in 0..i) {
            print("$ch ")
        }
        println()
    }
}
```

Android interview Questions

```
1
1*2
1*2*3
```

```
fun numberStarPattern(n: Int) {
    for (i in 1..n) {
        for (j in 1..i) {
            print(j)
            if (j != i) print("*")
        }
        println()
    }
}
```

✅ 20. Hill Pattern

markdown
CopyEdit

```
   *
  ***
 *****
*******
```

```
fun hillPattern(n: Int) {
    for (i in 1..n) {
        for (j in 1..(n - i)) print(" ")
        for (j in 1..(2 * i - 1)) print("*")
        println()
    }
}
```

DSA ppt

- **Fibonacci series**
  i/p=7
  o\p= 0 1 1 2 3 5 8 13

```
fun fibo(){
   val n=7
   var a=0
   var b=1
   var c=0
```

```
        c=a+b
         print(" $c")
        a=b
        b=c

    }

  }
```

- **Last term of fibbo series**

```
fun fibo(){
    val n=7
    var a=0
    var b=1
    var c=0

                                if(n==1){
                                 print(a)   return }

    for(i in 1 until n){
      c=a+b
         a=b
      b=c

    }
       print(" $c")

}
```

Q . 6 to the power 3 ===216

```
fun main() {
    val n = 2
    val p=10
     var result = 1
    for (i in 1..p) {
        result *= n


}
    println(result)
}
```

K-th Factor of n:

```
class Solution {
    fun kthFactor(n: Int, k: Int): Int {
        var count=0
        for(i in 1..n){
            if(n%i==0){
                count++
```

```
        }

return -1

    }
}
```

print the biggest digit of the number.

```
fun main() {
    var n = 43705
    var large=0
    while(n>0){
      var digit=n%10
  if(digit>large)
 {
     large=digit
   }
   n=n/10
         }
    println(large)
}
```

Digits in ascending order

```
fun main() {
    val n = 123452289
    println(asc(n))
}

fun asc(n: Int): Boolean {
    var num = n
    var lastDigit = 10  // Larger than any digit
    while (num > 0) {
        val digit = num % 10
        if (digit >= lastDigit) {
            return false
        }
        lastDigit = digit
        num /= 10
    }
    return true
}
```

Android interview Questions